Modeling Notes for Spring 2021

Douglas Robert Hundley
Mathematics Department
Whitman College

April 29, 2021

Contents

1 A Case Study in Learning 7
1.1 What is Learning? 7
1.2 Case Study: The $n-$Armed Bandit 8

2 Statistics 17
2.1 Quantities and Measures for Random Data 17
2.2 Variance and Standard Deviation 20
2.3 The Covariance Matrix 21
2.4 Exercises 22
2.5 Line of Best Fit 23

3 Another Case Study: Genetic Algorithms 25
3.1 Introduction to Genetic Algorithms 25
3.2 Example: Binary Strings 26
3.3 GA Using Real Numbers 29
3.4 Example: The Knapsack Problem 32

4 Linear Algebra 35
4.1 Representation, Basis and Dimension 35
4.2 The Four Fundamental Subspaces 41
4.3 Exercises 43
4.4 The Decomposition Theorems 45
4.5 Exercises 49
4.6 Notes about the SVD 50
4.7 Programming with the SVD 51
4.8 Generalized Inverses 52
4.9 Exercises 55

I Data Representations 57

5 The Best Basis 59
5.1 Introduction 59
5.2 The Best Basis and the Eigenvectors 62
5.3 Connecting to Principal Components Analysis (PCA) 64
5.4 Exercises 64
5.5 A Summary 65
5.6 Application: Eigenfaces 66

6 A Best Nonorthogonal Basis 73
6.1 Set up the Signal Separation Problem . . . 73
6.2 Signal Separation of Voice Data . . . 77
6.3 A Closer Look at the GSVD . . . 78

7 Local Basis and Dimension 81

8 Data Clustering 83
8.1 Introduction . . . 83
8.2 K-means clustering . . . 85
8.3 Neural Gas . . . 89
8.4 DBSCAN . . . 93
8.5 Comparisons between the algorithms . . . 98

II Functional Representations 101

9 Optimization 103
9.1 Introduction: Going from Data to Functions . . . 103
9.2 Optimization and Calculus . . . 103
9.3 Homework . . . 105
9.4 Linearization . . . 106
9.5 Nonlinear Optimization with Newton . . . 110
9.6 Gradient Descent . . . 111
9.7 Exercises . . . 114
9.8 Stochastic Gradient Descent (SGD) . . . 114
9.9 Homework . . . 116

10 k-Nearest Neighbors 117
10.1 Accuracy: Testing, Training and Validation Sets . . . 120
10.2 The Confusion Matrix . . . 121
10.3 Conclusions . . . 123
10.4 Homework . . . 124

11 Linear Neural Networks 125
11.1 A Model of Learning . . . 125
11.2 Linear Neural Nets . . . 125
11.3 Training a Network . . . 127
11.4 Hebbian Learning (On-line training) . . . 128
11.5 One step training . . . 131
11.6 Small Batch Training . . . 131
11.7 Time Series and Linear Networks . . . 132

12 Radial Basis Functions 135
12.1 Introduction . . . 135
12.2 Radial Basis Functions . . . 135
12.3 Orthogonal Least Squares . . . 140
12.4 Summary of the Radial Basis Functions . . . 141

13 Neural Networks **143**
13.1 From Biology to Construction 144
13.2 How Networks Compute 145
13.3 The Activation Function 146
13.4 Training and Error 149
13.5 Backpropagation of Error 150
13.6 Backprop Proved 152
13.7 Simple Construction of a Feed Forward Neural Net 154

14 Deep Learning **155**

III Appendices **157**

A An Introduction to Matlab **159**

B The Derivative **171**
B.1 The Derivative of f 171
B.2 Worked Examples: 174
B.3 Optimality 175
B.4 Worked Examples 177
B.5 Exercises 178

C Optimization **181**

D Matlab and Radial Basis Functions **183**

IV Bibliography **189**

V Index **193**

Chapter 1

A Case Study in Learning

1.1 What is Learning?

Here are some definitions of learning:[1]

- "Learning involves strengthening correct responses and weakening incorrect responses. Learning involves adding new information to your memory. Learning involves making sense of the presented material by attending to relevant information, mentally reorganizing it, and connecting it with what you already know."

  From eLearning and the Science of Instruction by Ruth C. Clark and Richard E. Mayer

- "Learning is the relatively permanent change in a persons knowledge or behavior due to experience. This definition has three components: 1) the duration of the change is long-term rather than short-term; 2) the locus of the change is the content and structure of knowledge in memory or the behavior of the learner; 3) the cause of the change is the learners experience in the environment rather than fatigue, motivation, drugs, physical condition or physiologic intervention."

  From Learning in Encyclopedia of Educational Research, Richard E. Mayer

In a broad sense, *learning is the process of building a "desirable" association between stimulus and response (domain and range), and is measured through resulting behavior on stimulus that has not been previously seen.*

In machine learning, problems are typically cast in one of two models: Either *supervised* or *unsupervised* learning:

- In *supervised learning*, we are given examples of proper behavior, and we want the learner (computer) to emulate and extrapolate from that behavior.
- In the other type of learning, *unsupervised learning*, no specific input-output are given. Rather an overall goal is given, and the learner (computer) must figure out how to get from the initial condition to the end goal. Here are some examples to help with the definition:
  - Suppose you have an old clunker of a car that doesn't have much of an engine. You're stuck in a valley, and so the only way out will be to go as fast as you can for a while, then let gravity take you back up the other side of the hill, then accelerate again, and so on. You hope that you can build up enough momentum to get out of the valley (that's the goal).
  - Suppose you're driving a tractor-trailer, and you need to back the trailer into a loading dock (that's your goal).
  - In a game of chess, the input would be the position of each of the chess pieces. The overall goal is to win the game.

[1] Downloaded from http://theelearningcoach.com/learning/10-definitions-learning/

In general, supervised learning is much easier than unsupervised learning. One reason is that in unsupervised learning, a lot of extra time is spent in trial-and-error exploration of the possible input space. Contrast that with supervised learning, where the "correct" behavior is explicitly given.

1.1.1 Questions for Discussion:

1. Consider the concept of *superstition*: This is a belief that one must engage in certain behaviors in order to receive a certain reward, where in reality, the reward did not depend on those behaviors. Is it possible for a computer to engage in superstitious activity? Discuss in terms of the supervised versus unsupervised learning paradigms.

2. A signal light comes on and is followed by one of two other lights. The goal is to predict which of the lights comes on given that the signal light comes on. The experimenter is free to arrange the pattern of the two response lights in any way- for example, one might come on 75% of the time.

   Let E_1, E_2 denote the event that the first (second) light comes on, and let A_1, A_2 denote the prediction that the first (second) light comes on (respectively). Let π be the probability that E_1 occurs.

   (a) If the goal is to maximize your reward through accurate predictions, what should you do in this experiment? Just give a heuristic answer- you do not have to formally justify it.

   (b) How would you program a machine to maximize it's prediction accuracy? Can you state this in mathematical terms?

   (c) What do you think happens with actual subject (human) trials?

1.2 Case Study: The $n-$Armed Bandit

The one armed bandit is slang for a slot machine, so the $n-$armed bandit can be thought of as a slot machine with n arms. Equivalently, you may think of a room with n slot machines.

The problem we're trying to solve is the classic Las Vegas quandry: How should we play the slot machines in order to maximize our returns?

Discussion Question: Is the $n-$armed bandit a case or supervised or unsupervised learning?

First, let us set up some notation: Let a be an integer between 1 and n that defines which machine we're playing. Then define the expected return:

$$Q(a) = \text{ The expected return for playing slot machine } a$$

You can also think of $Q(a)$ as the *mean* of the payoffs for slot machine a.

If we knew $Q(a)$ for each machine a, our strategy to maximize our returns would be very simple: "Play only machine a".

Of course, what makes the problem interesting is that we don't know what the any of the returns are, let alone which machine gives the maximum. That leaves us to estimate the returns, and because there will always be uncertainty associated with these estimates, we will never know if the estimates are correct. We hope to construct estimates that get better over time (and experience).

Let's first set up some notation. Let

$$Q_t(a) = \text{ Our estimation of } Q(a) \text{ at time } t.$$

so we hope that our estimates get better over time:

$$\lim_{t\to\infty} Q_t(a) = Q(a) \tag{1.1}$$

Suppose we play slot machine a a total of n_a times, with payoffs $r_1, \ldots, r_{n_a}$ (note that these values could be negative!). Then we might estimate $Q(a)$ as the mean of these values:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{n_a}}{n_a}$$

In statistical terms, we are using the sample mean to estimate the actual mean which is a reasonable thing to do as a starting point. We'll also initialize the estimates to be zero: $Q_0(a) = 0$.

We now come to the big question: What approach should we take to accomplish our goal (of maximizing our reward). The first one up is a good place to start.

1.2.1 The Greedy Algorithm

This strategy is straightforward: Always play the slot machine with the largest (estimated) payoff. If a_{t+1} is the machine we'll play at time $t+1$, then:

$$a_{t+1} = \arg\max \{Q_t(1), Q_t(2), \ldots, Q_t(n)\}$$

where "arg" refers to the argument of the maximum (which is an integer from 1 to n corresponding to the max) . If there is a tie, then choose one of them at random.

We'll need to translate this into a learning algorithm, so set's take a moment to see how we might implement the greedy algorithm in Matlab.

Programming Sidenote (Matlab/Python/R)

Here's an example, where we want to input a vector of real numbers in x, and output the indices (arguments) indicating where the maximum element is.

Matlab	Python	R
`x=[1 2 3 0 3]`	`x=np.array([1 2 3 0 3])`	`x<-c(1,2,3,0,3)`
`idx=find(x==max(x))`	`temp=np.where(x==np.amax(x))`	`idx<-which(x==max(x))`
	`idx=temp[0]`	

A couple of Python specific notes: First, we gave the command `import numpy as np`. Secondly, Python starts counting with index 0. Therefore, while in Matlab and R, `idx` will contain $3, 5$ and in Python, `idx` will contain $2, 4$.

Going back to the greedy algorithm, I think you'll see a problem- What if the estimations are wrong? Then its very possible that you'll get stuck on a suboptimal machine. This problem can be dealt with in the following way: Every once in a while, try out the other machines to see what you get. This is what we'll do in the next section.

1.2.2 The $\epsilon-$Greedy Algorithm

In this algorithm, rather than always choosing the machine with the greatest current estimate of the payout, we will choose, with probability ϵ, a machine at random.

With this strategy, as the number of trials gets larger and larger, $n_a \to \infty$ for *all* machines a, and so we will be guaranteed convergence to the proper estimates of $Q(a)$ for all a machines.

On the flip side, because we're always investigating other machines every once in a while, we'll never maximize our returns (we will always have suboptimal returns).

Using some "pseudo-code", here is what we want our algorithm to do:

```
For each time we choose a machine:

• Select an action:
```

  - `Sometimes choose a machine at random`
  - `Otherwise, select the action with greatest return.  Check for ties, and if there is a tie, pick on of them at random.`

- `Get your payoff`
- `Update the estimates` Q

`Repeat.`
The programming details so far are in separate files- One for Matlab, one for Python, and one for R.

The Softmax Action Selection

In the Softmax action selection algorithm, the idea is to construct a set of probabilities. This set will have the properties that:

- The machine (or arm) giving the highest estimated payoff will have the highest probability.
- We will choose a machine using the probabilities. For example, if the probabilities are $0.5, 0.3, 0.2$ for machines $1, 2, 3$ respectively, then machine 1 would be chosen 50% of the time, machine 2 would be chosen 30% of the time, and the last machine 20% of the time.

Therefore, this algorithm will maintain an exploration of all machines so that we will not get locked onto a suboptimal machine. Now if we have n machines with estimated payoffs recorded at time t as:

$$Q = [Q_t(1), Q_t(2), \ldots, Q_t(n)]$$

we want to construct n probabilities at time t as:

$$P = [P_t(1), P_t(2), \ldots, P_t(n)]$$

The requirements for this transformation are:

1. $P_t(k) \geq 0$ for $k = 1, 2, \ldots$ (because all probabilities are positive). Another way to say this is to say that the range of the transformation is nonnegative.
2. If $Q_t(a) < Q_t(b)$, then $P_t(a) < P_t(b)$. That is, the transformation must be strictly increasing for all domain values.
3. Finally, the sum of the probabilities must be 1.

A function that satisfies requirements 1 and 2 is the exponential function. It's range is nonnegative. It maps large negative values (large negative payoffs) to near zero probability, and it is strictly increasing. Up to this point, the transformation is:

$$\hat{P}_t(k) = \mathrm{e}^{Q_t(k)}$$

We need the probabilities to sum to 1, so we normalize the $\hat{P}_t(k)$:

$$P_t(k) = \frac{\hat{P}_t(k)}{\hat{P}_t(1) + \hat{P}_t(2) + \ldots + \hat{P}_t(n)} = \frac{\exp(Q_t(k))}{\sum_{j=1}^{n} \exp(Q_t(j))}$$

This is a popular technique worth remembering- We have what is called a Gibbs (or Boltzmann) distribution. We could stop at this point, but it is convenient to introduce a control parameter τ (sometimes this is referred to as the temperature of the distribution). Our final version of the transformation is given as:

$$P_t(k) = \frac{\exp\left(\frac{Q_t(k)}{\tau}\right)}{\sum_{j=1}^{n} \exp\left(\frac{Q_t(j)}{\tau}\right)}$$

EXERCISE: Suppose we have two probabilities, $P(1)$ and $P(2)$ (we left off the time index since it won't matter in this problem). Furthermore, suppose $P(1) > P(2)$. Compute the limits of $P(1)$ and $P(2)$ as τ goes to zero. Compute the limits as τ goes to infinity (Hint on this part: Use the definition, and divide numerator and denominator by $\exp(Q(1)/\tau)$ before taking the limit).

What we find from the previous exercise is that the effect of large τ (hot temperatures) makes all the probabilities about the same (so we would choose a machine almost at random). The effect of small *tau* (cold temperatures) makes the probability of choosing the best machine almost 1 (like the greedy algorithm).

In Matlab, these probabilities are easy to program. Let `Q` be a vector holding the current estimates of the returns, as before, and let `t=` τ, the (scalar) temperature. Then we construct a vector of probabilities using the softmax algorithm:

Matlab	Python	R
`P=exp(Q/t);`	`P=np.exp(Q/t)`	`P<-exp(Q/t)`
`P=P./sum(P);`	`P=np.divide(P,np.sum(P))`	`P=P/sum(P)`

Programming Comments

1. How to select action a with probability $p(a)$?

   We could do what we did before, and create a vector of choices with those probabilities fixed, but our probabilities change. We can also use the uniform distribution, so that if `x=rand`, and $x \leq p(1)$, use action 1. If $p(1) < x \leq p(1) + p(2)$, choose action 2. If $p(1) + p(2) < x \leq p(1) + p(2) + p(3)$, choose action 3, and so on. There is an easy way to do this, but it is not optimal (in terms of speed). We introduce two new Matlab functions, `cumsum` and `histc`.

   The function `cumsum`, which means *cumulative sum*, takes a vector x as input, and outputs a vector y so that `y=cumsum(x)` creates:

   $$y_k = \sum_{n=1}^{k} x_n = x_1 + x_2 + \ldots + x_k$$

   For example, if $x = [1, 2, 3, 4, 5]$, then `cumsum(x)` would output $[1, 3, 6, 10, 15]$

   The function `histc` (for *histogram count*) has the form: `n=histc(x,y)`, where the vector y is monotonically increasing. The elements of y form "bins" so that $n(k)$ counts the number of values in `x` that fall between the elements `y(k)` (inclusive) and `y(k+1)` (exclusive) in the vector `y`. Try a particular example, like:

   ```
   Bins=[0,1,2];
   x=[-2, 0.25, 0.75, 1, 1.3, 2];
   N=histc(x, Bins);
   ```

   `Bins` sets up the desired intervals as $[0, 1)$ and $[1, 2)$ and the last value is set up as its own interval, 2. Since -2 is outside of all the intervals, it is not counted. The next two elements of x are inside the first interval, and the next two elements are inside the second interval. Thus, the output of this code fragment is $N = [2, 2, 1]$.

   Now in our particular case, we set up the bin edges (intervals) so that they are the cumulative sums. We'll then choose a number between 0 and 1 using the (uniformly) random number $x =$ `rand`, and determine what interval it is in. This will be our action choice:

   ```
   P=[0.3, 0.1, 0.2, 0.4];
   BinEdges=[0, cumsum(P)];
   x=rand;
   Counts=histc(x,BinEdges);
   ActionChoice=find(Counts==1);
   ```

2. We have to change our parameter τ from some initial value τ_{init} (big, so that machines are chosen almost at random) to some small final value, τ_{fin}. There are an infinite number of ways of doing this. For example, a linear change from a value a to a value b in N steps would be the equation of the line going from the point $(1, a)$ to the point (N, b).

   Exercise: Give a formula for the parameter update, τ in terms of the initial value, τ_{init} and the final value, τ_{fin} if we use a linear decrease as t ranges from 1 to N.

   A more popular technique is to use the following formula, which we'll use to update many parameters. Let the initial value of the parameter be given as a, and the final value be given as b. Then the parameter p is computed as:

$$p = a \cdot \left(\frac{b}{a}\right)^{t/N} \tag{1.2}$$

   Note that when $t = 0$, $p = a$ and when $t = N$, $p = b$[2]

"Win-Stay, Lose-Shift" Strategy

The "Win-Stay, Lose-Shift" strategy discussed in terms of Harlow's monkeys and perhaps the probability matching experiments of Estes might be re-formulated here for the $n-$armed bandit experiment.

In this experiment, we interpret the strategy as: If I'm winning, make the probability of choosing that action stronger. If I'm losing, make the probability of choosing that action weaker. This brings us to the class of *pursuit* methods.

Define a^* to be the winning machine at the moment, i.e.,

$$a^* = \max_a Q_t(a)$$

The idea now is straightforward- Slightly increase the probability of choosing this winning machine, and correspondingly decrease the probability of choosing the others.

Define the probability of choosing machine a as $P(a)$ (or, if you want to explicitly include the time index, $P_t(a)$. Then given the winning machine index as a^*, we update the current probabilities by using a parameter $\beta \in [0, 1]$:

$$P_{t+1}(a^*) = P_t(a^*) + \beta \left[1 - P_t(a^*)\right]$$

and the rest of the probabilities decrease towards zero:

$$P_{t+1}(a) = P_t(a) + \beta \left[0 - P_t(a)\right]$$

Exercises with the Pursuit Strategy

1. Suppose we have three probabilities, P_1, P_2, P_3, and P_1 is the unique maximum. Show that, for any $\beta > 0$, the updated values still sum to 1.

2. Using the same values as before, show that, for any $\beta > 0$, the updated values will stay between 0 and 1- that is, If $0 \leq P_i \leq 1$ for all i before the update, then after the update, $0 \leq P_i \leq 1$.

3. Here is one way to deal with a tie (show that the updated values still sum to 1): If there are k machines with a maximum, update each via:

$$P_{t+1} = (1 - \beta) * P_t + \beta/k$$

4. Suppose that for some fixed j, P_j is always a loser (never a max). Show that the update rule guarantees that $P_j \to 0$ as $t \to \infty$. HINT: Show that $P_j(t) = (1 - \beta)^t P_j(0)$

5. Suppose that for some fixed j, P_j is always a winner (with no ties). Show that the update rule guarantees that $P_j \to 1$ as $t \to \infty$.

[2] In the C/C++ programming language, indices always start with zero, and this is leftover in this update rule. This is not a big issue, and the reader can make the appropriate change to starting with $t = 1$ if desired.

Matlab Functions `softmax` **and** `winstay`

Here are functions that will yield the softmax and win-stay, lose-shift strategies. Below each is a driver. Read through them carefully so that you understand what each does. We'll then ask you to put these into Matlab and comment on what you see.

```
function a=softmax(EstQ,tau)
% FUNCTION a=softmax(EstQ, tau)
%   Input:  Estimated payoff values in EstQ (size 1 x N,
%           where N is the number of machines
%           tau - "temperature":  High values- the probs are all
%           close to equal; Low values, becomes "greedy"
%  Output:  The machine that we should play (a number between 1 and N)

if tau==0
    fprintf('Error in the SoftMax program-\n');
    fprintf('Tau must be greater than zero\n');
    a=0;
    return
end

Temp=exp(EstQ./tau);
S1=sum(Temp);
Probs=Temp./S1;  %These are the probabilities we'll use

%Select a machine using the probabilities we just computed.
x=rand;
TotalBins=histc(x,[0,cumsum(Probs)']);
a=find(TotalBins==1);
```

Here is a driver for the softmax algorithm. Note the implementation details (e.g., how the "actual" payoffs are calculated, and what the initial and final parameter values are):

```
%Script file to run the N-armed bandit using the softmax strategy

%Initializations are Here:
NumMachines=10;
ActQ=randn(NumMachines,1);  %10 machines
NumPlay=1000;               %Play 100 times
Initialtau=10;              %Initial tau ("High in beginning")
Endingtau=0.5;
tau=10;
NumPlayed=zeros(NumMachines,1);  %Keep a running sum of the number
                                 %  of times each action is selected
ValPlayed=zeros(NumMachines,1);  %Keep a running sum of the total
                                 %  reward for each action
EstQ=zeros(NumMachines,1);
PayoffHistory=zeros(NumPlay,1);  %Keep a record of our payoffs

for i=1:NumPlay

    %Pick a machine to play:
    a=softmax(EstQ,tau);
```

```
    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    PayoffHistory(i)=Payoff;
    tau=Initialtau*(Endingtau/Initialtau)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Here is the function implementing the pursuit strategy (or "Win-Stay, Lose-Shift").

```
function [a, P]=winstay(EstQ,P,beta)
% function [a,P]=winstay(EstQ,P,beta)
%  Input:  EstQ, Estimated values of the payoffs
%          P = Probabilities of playing each machine
%          beta= parameter to adjust the probabilities, between 0 and 1
%  Output:  a = Which machine to play
%           P = Probabilities for each machine

[vals,idx]=max(EstQ);
winner=idx(1);  %Index of our "winning" machine

%Update the probabilities.  We need to do P(winner) separately.
NumMachines=length(P);
P(winner)=P(winner)+beta*(1-P(winner));

Temp=1:NumMachines;
Temp(winner)=[];  %Temp now holds the indices of all "losers"
P(Temp)=(1-beta)*P(Temp);

%Probabilities are all updated- Choose machine a w/prob P(a)
x=rand;
TotalBins=histc(x,[0,cumsum(P)']);
a=find(TotalBins==1);
```

And its corresponding driver is below. Again, be sure to read and understand what each line of the code does:

```
%Script file to run the N-armed bandit using pursuit strategy

%Initializations
NumMachines=10;
ActQ=randn(NumMachines,1);
NumPlay=2000;
Initialbeta=0.01;
Endingbeta=0.001;
beta=Initialbeta;
NumPlayed=zeros(NumMachines,1);
```

```
ValPlayed=zeros(NumMachines,1);
EstQ=zeros(NumMachines,1);
Probs=(1/NumMachines)*ones(10,1);

for i=1:NumPlay

    %Pick a machine to play:
    [a,Probs]=winstay(EstQ,Probs,beta);

    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    beta=Initialbeta*(Endingbeta/Initialbeta)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Homework: Implement these 4 pieces of code into Matlab, and comment on the performance of each. You might try changing the initial and final values of the parameters to see if the algorithms are *stable* to these changes. As you form your comments, recall our two competing goals for these algorithms:

- Estimate the values of the actual payoffs (more accurately, the mean payout for each machine).
- Maximize our rewards!

1.2.3 A Summary of Reinforcement Learning

We looked in depth at a basic problem of unsupervised learning- That of trying to find the best winning slot machine in a bank of many. This problem was unsupervised because, although we got rewards or punishments by winning or losing money, we did not know at the beginning of the problem what those payoffs would be. That is, there was no expert available to tell us if we were doing something correctly or not, *we had to infer correct behavior from directly playing the machines.*

We also saw that to solve this problem, we had to do a lot of *trial and error* learning- that's typical in unsupervised learning. Because an expert is not there to tell us the operating parameters, we have to spend time exploring the possibilities.

We learned some techniques for translating learning theory into mathematics, and in the process, we learned some commands in Matlab. We don't expect you to be an expert programmer - this should be a fairly gentle introduction to programming. At this stage, you should be able to read some Matlab code and interpret the output of an algorithm. Later on, we'll give you more opportunities to produce your own pieces of code.

In summary, we looked at the greedy algorithm, the $\epsilon-$greedy algorithm, the softmax strategy, and the pursuit strategy. You might consider how closely (if at all) these algorithms would reproduce human or animal behavior if given the same task.

There are many more topics in Reinforcement Learning to consider, we presented only a short introduction to the topic.

Chapter 2

Statistics

2.1 Quantities and Measures for Random Data

The most basic way to characterize a numerical data set is through one number- the mean (or median or mode).

- The **sample mean** for a discrete set of m numbers, $x_1, \ldots, x_m$ is given by:

$$\bar{x} = \frac{1}{m} \sum_{k=1}^{m} x_k$$

- The **mean of a set of m vectors in $\mathbb{R}^n$**:

  Suppose we have m vectors in $\mathbb{R}^n$. We can similarly define the (sample) mean by replacing the scalar x_k with the kth vector:

$$\bar{\boldsymbol{x}} = \frac{1}{m} \sum_{k=1}^{m} \boldsymbol{x}^{(k)}$$

  The jth element of the sample mean vector is just the sample mean of the (scalar) data in the jth dimension of your collection of vectors.

- Note that we can also define the mean for a collection of $m \times n$ matrices, as well. For example, if I have a collection of k photos that are each $m \times n$ pixels, then I can compute the mean photo by summing the matrices together, then dividing by k.

- In fact, matrices have different properties that we can consider- A matrix can be thought of as a collection of column vectors, or as a collection of row vectors, or simply a collection of numbers.

  Similarly, we can compute a mean of over the columns (and getting a column), or the mean over all rows (and get a row), or we can compute the mean over all the numerical values of the matrix, which is called the **grand mean**.

- Computing the mean in Matlab, Python and R:

  - Matlab:

    * If $\boldsymbol{x}$ is a row or column vector (example):

```
x=[1,2,3,4,5];
mean(x)
```

    * If X is an $m \times n$ matrix (example):

```
X=[1,2,3; 4,5,6];
mean(X)       %Returns a row vector as default
m=mean(X,1);  %Returns a row vector 1 x n
m=mean(X,2);  %Returns a column vector m x 1
m=mean(mean(X)); % Returns the grand mean (a scalar)
```

– Python: First, import *numpy*: `import numpy as np`

  ∗ If $\boldsymbol{x}$ is a row or column vector- two methods are shown below:

```
x=np.array([1,2,3,4,5])  #Example vector
np.mean(x)  #Returns a scalar
x.mean(0)   #Returns an array- a scalar or a vector
```

  ∗ If X is an $m \times n$ array

```
X=np.array([[1,2,3],[4,5,6]])
X.mean(0)  # Output: array([2.5, 3.5, 4.5])
X.mean(1)  # Output: array([2., 5.])
X.mean()   # Output the grand mean: 3.5
```

  Alternatively, for the row, column means respectively:

```
Xr=np.mean(X,axis=0)
Xm=np.mean(X,axis=1)
```

– R

  ∗ If $\boldsymbol{x}$ is a row or column vector: then

```
x<-c(1,2,3,4,5)
np.mean(x)  #Returns a scalar
x.mean(0)   #Returns an array- a scalar or a vector
```

  ∗ If X is an $m \times n$ array

```
x<-array(1:6,c(2,3))  #x is a 2 x 3 matrix
colMeans(x)       #Returns: 1.5 3.5 5.5
rowMeans(x)       #Returns: 3 4
```

  Alternatively,

```
apply(x,1,mean)   #Returns: 3 4
apply(x,2,mean)   #Returns: 1.5 3.5 5.5
```

A note about language: Should "row mean" be the mean found by summing the rows together, and producing a row, or should "row mean" be the sum through the rows, producing a column? I will typically mean the former, but I see that R uses the latter (the command `colMeans` produces the mean down the columns and returns a row, for example).

Just be sure you're consistent with whichever method you want to define.

- The *Median* is a number so that exactly half the data is above that number, and half the data is below that number. Although the median does not have to be unique, we follow the definitions below if we are given a finite sample:

  If there are an odd number of data points, the median is the middle point. If there is an even number of data points, then there are two numbers in the middle- the median is the average of these.

  The syntax for the median works in very much the same way as the mean.

- The *Mode* is the value taken the most number of times. In the case of ties, the data is multi-modal.

  We typically would not use the mode unless there is a special reason to do so.

2.1.1 Matlab note, Linear Algebra

We'll be subtracting a row vector from each row of a matrix, and similarly, we'll subtract a column vector from each column of a matrix. You'll note that, if A is a matrix, $\boldsymbol{r}$ is a row, and $\boldsymbol{c}$ is a column, then writing something like:

$$A - \boldsymbol{r} \qquad A - \boldsymbol{c}$$

would not be defined in linear algebra, and for older versions of Matlab, this was the case as well. This changed several years ago, so that "Matrix - Row" is assumed to be row subtraction for each row of the matrix, and "Matrix - Column" is assumed to be carried out column-wise on the matrix. We'll see this below.

2.1.2 Centering and Double Centering Data

Let matrix A be $m \times n$, which may be considered n points in $\mathbb{R}^m$ (this looks at the data column-wise) or m points in $\mathbb{R}^n$ (looking at the data row-wise). If we wish to look at A both ways, a double-centering may be appropriate.

The result of the double-centering will be that (in Matlab), we determine $\hat{A}$ so that

$$\texttt{mean}(\hat{A}, 1) = 0, \qquad \texttt{mean}(\hat{A}, 2) = 0$$

There are a couple of ways to do this. Here is onw algorithm, where the means are computed first.

Algorithm for Double Centering

- Given matrix A:
  - Compute the mean of the rows. Call this row $\boldsymbol{r}$.
  - Compute the mean of the columns. Call this column $\boldsymbol{c}$.
  - Compute the grand mean. Call this scalar g.
- Output the matrix: $A - \boldsymbol{r} - \boldsymbol{c} + g$.

Here is the implementation in Matlab, Python and R:

Matlab	Python	R
`A=[1,2,3;4,5,6];`	`A=np.array([[1,2,3],[4,5,6]])`	`A<-array(1:6,c(2,3))`
`r=mean(A,1);`	`r=A.mean(0)`	`r=apply(A,2,mean)`
`c=mean(A,2);`	`c=A.mean(1)`	`c=apply(A,1,mean)`
	`c=c[:,np.newaxis]`	
`g=mean(mean(A));`	`g=A.mean()`	`g=mean(apply(A,1,mean))`
`B=A-r-c+g`	`B=A-r-c+g`	`B1=sweep(A,2,r)`
		`B2=sweep(B1,1,c)`
		`B=B2+g`

Python Note about Vector Subtraction

In the Python code, we needed to "reshape" the size of the column representing the mean across A. In the first line, `c=A.mean(1)`, we see that the shape of c is $(2,)$. After the next line, the shape of c is $(2, 1)$. We didn't need to do that for r, although we probably should have- That is, the current shape of r is $(3,)$, but we really wanted a row vector, so we could reshape it as `r=r[np.newaxis,:]` so that the shape is $(1, 3)$. Why? If you leave c and r as defined in the code above, what happens with `c-r`? You get a matrix- That is,

$$c - r - \begin{bmatrix} 2 \\ 5 \end{bmatrix} - [2.5, 3.5, 4.5] = \begin{bmatrix} -0.5 & -1.5 & -2.5 \\ 2.5 & 1.5 & 0.5 \end{bmatrix}$$

So we need to be careful when we think we're subtracting vectors. Now back to the text...

As a final note, double centering is only suitable if it is reasonable that the $m \times n$ matrix may be data in either $\mathbb{R}^n$ or $\mathbb{R}^m$. For example, you probably would not double center a matrix that is 5000×2- Treat this as 5000 points in $\mathbb{R}^2$, so that the mean is in $\mathbb{R}^2$.

2.2 Variance and Standard Deviation

The number that is used to describe the spread of the data about its mean is the *variance*. As with the mean, we rarely know the underlying distribution, so again we'll focus on the sample variance.

Let $\{x_1, \ldots, x_m\}$ be m real numbers, and $\bar{x}$ its sample mean. Then the **sample variance** is:

$$s^2 = \frac{1}{m-1} \sum_{k=1}^{m} (x_k - \bar{x})^2$$

If we think of the data as a vector of length m, then this formula becomes:

$$s^2 = \frac{1}{m-1} \|\boldsymbol{x} - \bar{x}\|^2$$

The **standard deviation** is the square root of the variance, so the standard deviation is s.

Quick Example

Let's take some template data to look at what the variance (and standard deviation) measure: Consider the data:

$$-\frac{2}{n}, -\frac{1}{n}, 0, \frac{1}{n}, \frac{2}{n}$$

If n is large, our data is tightly packed together about the mean, 0. If n is small, the data are spread out. The variance and standard deviation of this sample is:

$$s^2 = \frac{1}{4}\left(\frac{4+1+0+1+4}{n^2}\right) = \frac{5}{2}\frac{1}{n^2}, \qquad s = \sqrt{\frac{5}{2}}\,\frac{1}{n}$$

and this is in agreement with our heuristic: If n is large, our data is tightly packed about the mean, and the standard deviation is small. If n is small, our data is loosely distributed about the mean, and the standard deviation is large. Another way to look at the standard deviation is in linear algebra terms: If the data is put into a vector of length m (call it $\boldsymbol{x}$), then the (sample) standard deviation can be computed as:

$$s = \frac{\|\boldsymbol{x} - \bar{x}\|}{\sqrt{m-1}}$$

2.2.1 Covariance and Correlation Coefficients

If we have two data sets, sometimes we would like to compare them to see how they relate to each other. In this case, it is important that the two data sets be ordered so that x_1 is being compared to y_1, then x_2 is compared to y_2, and so on.

Definition: Let $X = \{x_1, \ldots, x_n\}, Y = \{y_1, \ldots, y_n\}$ be two ordered data sets with means m_x, m_y respectively. Then the *sample covariance* of the data sets is given by:

$$\mathrm{Cov}(X, Y) = s_{xy}^2 = \frac{1}{n-1} \sum_{k=1}^{n} (x_k - m_x)(y_k - m_y)$$

There are exercises at the end of the chapter that will reinforce the notation and give you some methods for manipulating the covariance. In the meantime, it is easy to remember this formula if you think of the following:

If X and Y have mean zero, and we think of X and Y as vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, then the covariance is just the dot product between the vectors, divided by $n-1$:

$$\text{Cov}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n-1}\boldsymbol{x}^T\boldsymbol{y}$$

We can then interpret what it means for X, Y to have a covariance of zero: $\boldsymbol{x}$ is "orthogonal" to $\boldsymbol{y}$. Continuing with this analogy, if we normalized by the size of $\boldsymbol{x}$ and the size of $\boldsymbol{y}$, we'd get the cosine of the angle between them. This is the definition of the correlation coefficient, and gives the relationship between the covariance and correlation coefficient:

Definition: The **correlation coefficient** between the data ordered in vector $\boldsymbol{x}$ and data in $\boldsymbol{y}$ is given by:

$$r_{xy} = \frac{s_{xy}^2}{s_x s_y} = \frac{\sum_{k=1}^{n}(x_k - m_x)(y_k - m_y)}{\sqrt{\sum_{k=1}^{n}(x_k - m_x)^2 \cdot \sum_{k=1}^{n}(y_k - m_y)^2}}$$

If the data in $\boldsymbol{x}$ and $\boldsymbol{y}$ have been mean subtracted, then the formula is reminiscent of something from linear algebra:

$$r_{xy} = \frac{\boldsymbol{x}^T\boldsymbol{y}}{\|\boldsymbol{x}\| \cdot \|\boldsymbol{y}\|} = \cos(\theta)$$

This works out so nicely because we have a $\frac{1}{n-1}$ in both the numerator and denominator, so they cancel each other out.

We also see immediately that r_{xy} can only take on the real numbers between -1 and 1. Some interesting values of r_{xy}:

If r_{xy} is:	Then the data is:
1	Perfectly correlated ($\theta = 0$)
0	Uncorrelated ($\theta = \frac{\pi}{2}$)
-1	Perfectly (negatively) correlated ($\theta = \pi$)

One last comment before we leave this section: The covariance s_{xy}^2 and correlation coefficient r_{xy} only look for *linear* relationships between data sets!

For example, we know that $\sin(x)$ and $\cos(x)$, either as functions, or as data points sampled at equally spaced intervals, will be uncorrelated, but, because $\sin^2(x) + \cos^2(x) = 1$, we see that $\sin^2(x)$ and $\cos^2(x)$ are perfectly correlated.

This difference is the difference between the words "correlated" and "statistically independent". Statistical independence (not defined here) and correlations are not the same thing. We will look at this difference closely in a later section.

2.3 The Covariance Matrix

Suppose we have a collection of n columns, where each column represents data in one dimension. And suppose each column has p elements. The $p \times n$ matrix X can be thought of as either p points in $\mathbb{R}^n$ or n points in $\mathbb{R}^p$. Thinking of have p points in dimension i and p points in dimension j, we can compute the variance between those dimensions.

Continuing, we can compute the covariance between all pairings of the n dimensions resulting in an $n \times n$ matrix (note that the diagonal entries would be the covariance of a set of data with itself- which is the regular variance). Such a matrix is known as the covariance matrix.

In the formulas below, we'll assume that X has beende*mean-subtracted* (subtract the row representing the mean from all rows of X). The $(i, j)^{\text{th}}$ entry in the covariance matrix is then defined as the covariance between the ith and j^{th} dimensions:

$$s_{ij}^2 = \frac{1}{p-1}\sum_{k=1}^{p} X(k,i) \cdot X(k,j)$$

Computing this for all i, j will result in an $n \times n$ symmetric matrix, C, for which $C_{ij} = s_{ij}^2$.

In the exercises, you'll show that an alternative way of computing the covariance matrix is by using what we'll refer to as its definition below.

Definition: Let X denote a matrix of data, so that, if X is $p \times n$, then we have p data points in $\mathbb{R}^n$. Furthermore, we assume that the data in X has been mean subtracted (so the mean in $\mathbb{R}^n$ is the zero vector). Then the $n \times n$ *covariance matrix* associated with X is given by:

$$C = \frac{1}{p-1} X^T X$$

In the language of your choice, it is straightforward to compute the covariance matrix- but be sure to keep in mind the dimensions.

Matlab	Python	R
`X=rand(10,3);`	`X=np.random.rand(10,3)`	`X<-matrix(runif(30),nrow=10)`
`C=cov(X);`	`C=np.cov(X.T,bias=False)`	`C<-cov(X)`

You might note that in Python, the default matrix arrangement is reversed, and the default number to divide by is n rather than $n-1$, unless you include the "bias" option.

2.4 Exercises

1. Compute the covariance between the following data sets:

$$\begin{array}{c|ccccccc} x & -1.0 & -0.7 & -0.4 & -0.1 & 0.2 & 0.5 & 0.8 \\ \hline y & -1.3 & -0.7 & -0.1 & 0.5 & 1.1 & 1.7 & 2.3 \end{array} \tag{2.1}$$

2. Let's explore some of the things mentioned in the text. Use a computer program to verify the following:
   (a) "If $\boldsymbol{t}$ is a vector of equally spaced points, the $\sin(t)$ and $\cos(t)$ (computer notation) will be uncorrelated". Show this by example using Matlab, Python or R.
   (b) Continuing, show that $\sin^2(t)$ and $\cos^2(t)$ are perfectly correlated (again, using Matlab, Python or R).
   (c) Take the vector $\boldsymbol{t}$, and let $\boldsymbol{y} = 2\boldsymbol{t} - 5$. Show that the vectors $\boldsymbol{t}$ and $\boldsymbol{y}$ have a correlation of 1.
   (d) Redo the previous experiment, but use any negative slope. What is the correlation coefficient?
3. Let $\boldsymbol{x}$ be a vector of data with mean $\bar{x}$, and let a, b be scalars.
   (a) Show, using the definition of the mean, that the mean of $a\boldsymbol{x}$ is $a\bar{x}$.
   (b) Similarly, find a formula for the mean of $a\boldsymbol{x} + b$ in terms of the mean of $\boldsymbol{x}$.
4. Let $\boldsymbol{x}$ be a vector of data with variance s_x^2, and let a, b be scalars.

(a) Show, using the definition of variance, that the variance of $\boldsymbol{ax}$ is $a^2 s_x^2$. You might start with:

$$s^2_{(ax)} = \frac{1}{m-1}\sum_{i=1}^{m}(ax_i - \overline{ax_i})^2$$

(b) Similarly, find a formula for the variance of $a\boldsymbol{x} + b$ in terms of the variance of $\boldsymbol{x}$.

The exercises below explore the notion that the correlation tries to find linear relationships between data.

5. Show that, for data in vectors $\boldsymbol{x}$, $\boldsymbol{y}$ and a real scalar a,

$$\mathrm{Cov}(ax, y) = a\mathrm{Cov}(x, y) = \mathrm{Cov}(x, ay)$$

6. For a and b fixed scalars, and data in vector $\boldsymbol{x}$ find a formula for the $\mathrm{Cov}(x, ax + b)$ in terms of the variance of $\boldsymbol{x}$.

7. For a and b fixed scalars, and data in vector $\boldsymbol{x}$, let $\boldsymbol{y} = a\boldsymbol{x} + b$, find the correlation coefficient r_{xy}^2 and simplify as much as possible. What do you get?

8. Let X be a $p \times n$ matrix of data, where we n columns of p data points (you may assume each column has zero mean). Show that the $(i, j)^{\mathrm{th}}$ entry of $\dfrac{1}{p-1}X^T X$ is the covariance between the ith and j^{th} columns of X. HINT: It might be convenient to write X in terms of its columns,

$$X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]$$

Also show that $\frac{1}{p-1}X^T X$ is a symmetric matrix.

2.5 Line of Best Fit

In this section, we examine the simplest case of fitting data to a function. We are given p pairs of data (t is for "target", we'll use y for something else):

$$(x_1, t_1), (x_2, t_2), \ldots (x_p, t_p)$$

We wish to find a line through the data. That is, we want to find scalars m, b so that

$$mx_i + b = t_i$$

for each pair (x_i, t_i). Of course, if the data actually was on a line, we would not need p points- only two are needed.

Therefore we assume that there is something going on so that the data is not exactly on the line- for each point, we now have an error. We are distinguishing now between the point on the line:

$$y_i = mx_i + b$$

and the *desired* value t_i. Now the error at the ith point is defined as:

$$(t_i - y_i)^2 = (t_i - (mx_i + b))^2$$

and the overall error is the **sum of squares** error (summed over the p points):

$$E(m, b) = \sum_{k=1}^{p}(t_k - (mx_k + b))^2$$

We have now translated our problem into a Calculus problem- Find the minimum of $E(m, b)$. Here are some exercises to lead you to the solution:

Exercises with the Error

1. E is a function of m and b, so the minimum value occurs where

$$\frac{\partial E}{\partial m} = 0 \qquad \frac{\partial E}{\partial b} = 0$$

Show that this leads to the system of equations: (the summation index is 1 to p)

$$\begin{aligned} m\sum x_k^2 + b\sum x_k &= \sum x_k t_k \\ m\sum x_k + bn &= \sum t_k \end{aligned}$$

2. With linear algebra, the line of best fit becomes the matrix-vector equation below that has **no solution**. Therefore, we are looking for the least squares solution.

$$\begin{array}{ll} mx_1 + b & = t_1 \\ mx_2 + b & = t_2 \\ mx_3 + b & = t_3 \\ \vdots & \vdots \\ mx_p + b & = t_p \end{array} \quad \Rightarrow \quad \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ \vdots & \vdots \\ x_p & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_p \end{bmatrix} \quad \Rightarrow \quad A\boldsymbol{c} = \boldsymbol{t}$$

In this case, the error is the norm of the difference between the matrix product $A\boldsymbol{c} = \boldsymbol{y}$ and the known target vector $\boldsymbol{t}$, but usually we square that difference to get the "least squares" solution. In other words, we try to find $\boldsymbol{c}$ that minimizes:

$$E(\boldsymbol{c}) = \|\boldsymbol{t} - \boldsymbol{y}\|^2 = \|\boldsymbol{t} - A\boldsymbol{c}\|^2$$

For now, we can solve this problem using the **normal equations**. That is, we will multiply both sides of our equation by A^T to get:

$$A\boldsymbol{c} = \boldsymbol{t} \quad \Rightarrow \quad A^T A\boldsymbol{c} = A^T\boldsymbol{t}$$

Originally, A was $p \times 2$, so now $A^T A$ is 2×2, which we can invert. EXERCISE: Show that this system of two equations in two variables is the same as the system we obtained by setting the partial derivatives to zero.

3. (Toy problem) Find the line of best fit through the data:

x	-1.0	-0.7	-0.4	-0.1	0.2	0.5	0.8
y	-1.3	-0.7	-0.1	0.5	1.1	1.7	2.3

(2.2)

A word of caution:

The line of best fit is **mathematically a unique answer**. That is, given a specific set of data, we get one answer using our technique. However, what we have not considered is whether or not the data is actually linear!

Students of statistics will recognize that this is the issue that you spend a good amount of time studying-tests for goodness of fit exist, but would take us too far afield for now.

Chapter 3

Another Case Study: Genetic Algorithms

3.1 Introduction to Genetic Algorithms

The section on Genetic Algorithms (GA) appears here because it is closely related to the problem of unsupervised learning. Much of what follows was done in collaboration with a student, Jenna Carr, who was working on her senior project.

It is probably easiest to think of a GA as a technique that will optimize some "fitness function", which is what we typically think of when we talk of biological evolution. Researchers use vocabulary that has its roots in biology, but we will try to keep the vocab neutral.

Let's start things off by considering how evolution works in its simplest form. We begin with some *population* that has *chromosomes*, and these chromosomes defines how well an individual matches its environment. By mating, chromosomes from different individuals are somehow combined to make new individuals, and a new generation appears.

Here are components that are common to almost all forms of a GA:

- A population of individuals, each with chromosomes.

  A key step is in how one translates a chromosome into a numerical string. We also need to decide what will constitute a valid string, or valid chromosome.

  For example, some practitioners will use binary strings. This works well for some problems, but in other problems we may want to use a continuum rather than a discrete set of points.

- A fitness function.

  This function uses chromosomes to measure how well an individual is "performing" in a given environment. We will need to be able to compute a real value for any given (valid) chromosome.

  The overall goal of the GA is to find member(s) of the population that optimize the fitness function.

- A process by which we can select which individuals will reproduce.

- A process by which chromosomes combine to create new chromosomes.

  This should also include some random mutation.

There are many different algorithms, and most differ on how one interprets the four previous items. Before we get too abstract, we'll work through a very simple example.

3.2 Example: Binary Strings

Suppose our problem is to maximize the number of 1's in a bit string of length 20. There is an easy solution to this- The optimal string would be a string of 1's, but let's see how our four pieces fit together in this problem:

1. The population: The population consists of strings of length 20, where each element of the string is a 0 or a 1. We also need the size of the population- In this case, we set it arbitrarily to 100.

2. The fitness function: The fitness function takes in a binary string of length 20, and outputs the number of 1's in the string.

3. A process by which individuals are selected for reproduction:

   In this example, we'll select individuals at random. Unfortunately, this is not specific enough, since there are many ways to make a "random" selection. In this case, if individual j has fitness value $f(j)$, then consider the set of probabilities:

   $$P_1 = \frac{f(1)}{\sum_{j=1}^{100} f(j)}, P_2 = \frac{f(2)}{\sum_{j=1}^{100} f(j)}, \ldots, P_{100} = \frac{f(100)}{\sum_{j=1}^{100} f(j)}$$

   In this example, these are indeed probabilities since they are non-negative and sum to 1. Therefore, we will define the selection process as: Select individual j with probability P_j. We should consider this problem separately, and write a separate function that will perform this function (and we will). This is actually a fairly standard way of selection.

   Here is the Matlab code we'll use. You might notice that it comes from code we used in the $n-$armed bandit problem, where we had to select machine a using probability P_a. Here, we'll call the function `RandChooseN`.

```
function Action=RandChooseN(P,N)
% function Action=RandChooseN(P,N)
%   Choose N numbers from 1 to length(P) using the
%   probabilities in P.  For example, if P=[0.1,0.9],
%   we choose "1" 10% of the time, and "2" 90% of
%   the time.   Selection is done WITH replacement,
%   so, for example, if N=3, we could return [2, 2, 2]

%Set up the bins
BinEdges=[0, cumsum(P(:)')];
Action=zeros(1,N);

for i=1:N
    x=rand;
    Counts=histc(x,BinEdges);
    Action(i)=find(Counts==1);
end
```

4. A process by which chromosomes combine to create new chromosomes:

   Chromosomes will be combined in two ways: Using crossover, then using a random mutation. We'll define the crossover process first.

   If we have two "parents", "ma" and "pa", each defined by a string of 20 characters, then in crossover we choose a crossover point that is an integer between 1 and 20. Let's call the crossover value x_p.

The two offspring will then be two strings of length 20. The first uses the first x_p characters from "ma" and the remaining characters from the corresponding values in "pa".

Similarly, the second offspring will use the initial x_p characters from "pa" and the remainder from "ma".

Therefore, the offspring 1 would have its front part from "ma" and back part from "pa" (split at x_p). Offspring 2 is the opposite, its front part from "pa" and back part from "ma".

The crossover point may be the same for all parents, or it can be randomly selected for each parent. In the code that follows, the crossover points were selected at random for each pair of parents.

Furthermore, keeping with our analogy of evolution, we also include the possibility (usually a very small one) that mutations may occur, and they occur randomly.

There are many other ways one could perform crossover and mutation- The only caution here is that one must ensure that the offspring are still valid chromosome strings.

3.2.1 Matlab Code for Example 1

Here is the Matlab code that implements the algorithms from the text. Save it in Matlab and run it several times and compare the outputs.

```
%% Genetic Algorithm Example
%
%  Problem:  Define the population as binary strings of length 20,
%  with the fitness function:  Sum of 1's.  Use a population of
%  100 with certain probabilities defined below to see how long
%  it takes to reach a string of all 1's.

%%  Setup the GA parameters

ff=inline('sum(x,2)'); % objective function

maxit=200;        % max number of iterations (for stopping)
maxcost=9999999; %Maximum allowable cost (for stopping)
popsize=100; % set population size (it is constant)
mutrate=0.001; % set mutation rate (a small probability)

lenx=20; % Length of the chromosome (20 here)

%% Create the initial population
pop=round(rand(popsize,lenx)); % random population of 1s
                               % and 0s (using default, 100x20)

% Initialize cost and other items to set up the main loop
cost=feval(ff,pop); % calculates population cost using ff

[cost,ind]=sort(cost,'descend'); % max element in first entry
pop=pop(ind,:); % sorts population with max cost first

probs=cost/sum(cost);  %Simple normalization for probabilities.

% We'll be tracking the following quantities for our plot:
maxc(1)=max(cost); % minc contains min of population
meanc(1)=mean(cost); % meanc contains mean of population
```

```
%% MAIN LOOP

iga=0;  % Initalize the variable used in the loop below.
        %  This will keep track of how many iterations we've
        %  used and will get us out of the loop at the max

while iga<maxit

    iga=iga+1; % increments generation counter

% Choose mates

    M=ceil(popsize/2); % number of matings
    ma=RandChooseN(probs,M); % mate #1
    pa=RandChooseN(probs,M); % mate #2
% ma and pa contain the *indices* of the chromosomes that will mate

% Select crossover values for each set of parents
    xp=randi([1,lenx],1,M);  % M integers from 1 to lenx
                             % In this code, crossover is different
                             %  for each set of parents.

    Temp=pop;  %Just temporary storage as we perform crossover

%Crossover: One offspring will be stored in the odd indices,
%           the other in the even indices.
    for k=1:M
      pop(2*k-1,:)=[Temp(ma(k),1:xp(k)) Temp(pa(k),xp(k)+1:20)];
      pop(2*k,:)=[Temp(pa(k),1:xp(k)) Temp(ma(k),xp(k)+1:20)];
    end

% Mutate the population
    nmut=ceil((popsize-1)*lenx*mutrate); % total number of mutations
    mrow=ceil(rand(1,nmut)*(popsize-1))+1; % row to mutate
    mcol=ceil(rand(1,nmut)*lenx); % column to mutate
    for ii=1:nmut
        pop(mrow(ii),mcol(ii))=abs(pop(mrow(ii),mcol(ii))-1); % toggles bits
    end % ii

%% The population is re-evaluated for cost
    cost=feval(ff,pop); % calculates population cost using ff
    [cost,ind]=sort(cost,'descend'); % max element in first entry
    pop=pop(ind,:); % sorts population with max cost first

    probs=cost/sum(cost);  %Re-set probabilities

% We keep track of some values for graphical output:
    maxc(iga+1)=max(cost);
    meanc(iga+1)=mean(cost);
```

```
%% Stopping criteria.  The double bar: || is an "or"
    if iga>maxit || cost(1)>maxcost
        break
    end

end %iga


%% Displays the output
day=clock;
disp(datestr(datenum(day(1),day(2),day(3),day(4),day(5),day(6)),0))
%disp(['optimized function is ' ff])
format short g
disp(['popsize = ' num2str(popsize) ' mutrate = ' num2str(mutrate)]);
disp(['#generations=' num2str(iga) ' best cost=' num2str(cost(1))]);
fprintf('best solution\n%s\n',mat2str(int8(pop(1,:))));
figure(1)
iters=0:length(maxc)-1;
plot(1:(iga+1),maxc,1:(iga+1),meanc);
xlabel('generation');ylabel('cost');
```

In this example, we have seen how to implement one GA on a population of binary strings. In the next section, we will consider changes if the string consists of real numbers.

3.3 GA Using Real Numbers

Consider the following problem: Find the minimum value of f over the domain $0 \le x \le 10$, $0 \le y \le 10$, and

$$f(x, y) = x\sin(4x) + 1.1y\sin(2y)$$

If one attempts to use "classical" techniques like gradient descent, then you rapidly get stuck in a local minimum (or a local max). The graph of f is shown in Figure 3.1.

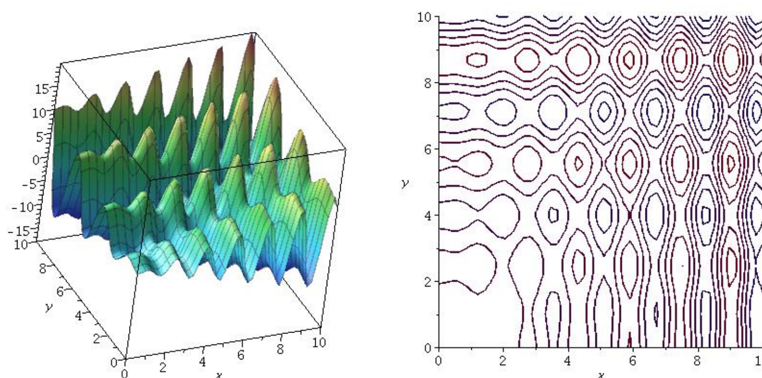


Figure 3.1: The surface and contour map for the function we will try to minimize, $f(x, y) = x\sin(4x) + 1.1y\sin(2y)$.

Now we'll work through the steps for defining the genetic algorithm:

1. Define the population. We will have 12 individuals (that number was randomly selected). Each chromosome will be an ordered pair of real numbers:

   $$[x, y] \qquad 0 \le x \le 10, 0 \le y \le 10$$

   The population will be initialized randomly.

2. Define the fitness function. The fitness function is given to us already:

   $$f(x, y) = x \sin(4x) + 1.1y \sin(2y)$$

3. A process by which individuals are selected for reproduction:

   One interesting way of putting together the probability distribution is to use *rank order*. For example, suppose we have N things in order from "best" to "last". Then we choose item j with probability

   $$\frac{N - (j - 1)}{\sum_{i=1}^{N} i}$$

   with maximum probability $N/\sum i$ and minimum probability $1/\sum i$.

   For example, if we have 4 things, the probability distribution would be:

   $$\frac{4}{10}, \quad \frac{3}{10}, \quad \frac{2}{10}, \quad \frac{1}{10}$$

   We'll implement this in the Matlab script.

4. Crossover and Mutation:

   This is where things get a little trickier. Here is one method that we implemented in Matlab below.

   - Select the crossover point at random (in this case, the x coordinate or the y coordinate).
   - The coordinate z_{ma}, z_{pa} selected for crossover will then be updated by selecting $0 < \beta < 1$:

     $$z_{ma_{new}} = (1 - \beta)z_{ma} + \beta z_{pa}$$

     $$z_{pa_{new}} = (1 - \beta)z_{pa} + \beta z_{ma}$$

   - The other coordinates will remain.
   - Since there are additional constraints on the coordinates, these will also be checked. That is, if the new x coordinate is greater than 10, it will be chopped to 10. Similarly, if a coordinate is less than zero, it will be chopped to zero.
   - If selected for mutation, a random number (uniform) will be selected between 0 and 10.

Matlab Script for the Optimization

```
%% Script File:  Optimization and GA

% Initialize the population:
Pop=10*rand(12,2);

% Stopping criteria
maxit=200;  %Max number of iterations
mincost=-99999999;  %Minimum cost

% Parameters:
```

```
popsize=12;
mutrate=0.05;  %Mutation rate
popKept=0.5;  %Fraction of the population to keep
keep=floor(popKept*popsize);  %How many individuals are kept
M=ceil((popsize-keep)/2); % number of matings; 2 mates create 2 offspring
crossprob=round(rand(maxit,1)); %0= x-coord, 1= y-coord
nmut=ceil((popsize-1)*2*mutrate);  %Number of mutations

% Fitness function:
ff='testfunction';  %The fitness function is in the file testfunction.m

% Probability distribution (won't change in this example)
probs=(keep:-1:1)/sum(1:keep);  %Probability is rank ordering

%% Initialize the population:

cost=feval(ff,Pop);  %Initial costs
[cost,idx]=sort(cost);  % Default sort is from small to large
Pop=Pop(idx,:);
minc(1)=min(cost);  %Minimum cost, for plotting later
meanc(1)=mean(cost);  %Mean cost for this population (for plotting later)


%% Main loop
iga=0;
while iga<maxit

    iga=iga+1;

    % Pair up and mate:
    ma=RandChooseN(probs,M);
    pa=RandChooseN(probs,M);

    % Set up crossover and mutation:
    idx2=keep+1:popsize;
    beta=rand;
    PopMa=Pop(ma,:);  PopPa=Pop(pa,:);
    if crossprob(iga)==0  %Crossover the x-coordinate
        for j=1:M
            Pop(idx2(2*j-1),1)=(1-beta)*PopMa(j,1)+beta*PopPa(j,1);
            Pop(idx2(2*j-1),2)=PopMa(j,2);
            Pop(idx2(2*j),1)=(1-beta)*PopPa(j,1)+beta*PopMa(j,1);
            Pop(idx2(2*j),2)=PopPa(j,2);
        end
    else  %Crossover the y-coordinate
        for j=1:M
            Pop(idx2(2*j-1),1)=PopMa(j,1);
            Pop(idx2(2*j-1),2)=(1-beta)*PopMa(j,2)+beta*PopPa(j,2);
            Pop(idx2(2*j),1)=PopPa(j,1);
            Pop(idx2(2*j),2)=(1-beta)*PopPa(j,2)+beta*PopMa(j,2);
        end
    end
```

```
    % Mutation

    mrow=sort(ceil(rand(1,nmut)*(popsize-1))+1);
    mcol=ceil(rand(1,nmut)*2);
    for ii=1:nmut
        Pop(mrow(ii),mcol(ii))=10*rand;
    end

    % New cost, set up for next iteration:
    cost=feval(ff,Pop);  %Initial costs
    [cost,idx]=sort(cost);  % Default sort is from small to large
    Pop=Pop(idx,:);
    minc(iga+1)=min(cost);  %Minimum cost, for plotting later
    meanc(iga+1)=mean(cost);  %Mean cost for this population (for plotting later)


    % Stopping criteria
    if iga>maxit || cost(1)<mincost
        break
    end

end % End of the while loop

%% Display the results
figure(1)
iters=0:length(minc)-1;
plot(iters,minc,iters,meanc,'-');
xlabel('generation');ylabel('cost');
```

3.4 Example: The Knapsack Problem

Suppose you want to go on a camping trip. You're planning on carrying no more than 20 kg of supplies. The problem is that you cannot possibly take all the supplies that you may want. An additional worry is that not every is kilogram is equal- For example, taking the tent should be much more important than taking a novel.

Therefore, you set up a scale on which you rate the relative importance of each item. Here is your list in Table 3.1.

Now we need to translate this problem into a GA (and there are many ways you might do it). Let's see if we can start.

One way to define the fitness function may be the following. The fitness value will be the sum of the values, if the weight is less than (or equal to) 20 kg. Otherwise, the fitness will take a value of -1. We will then try to maximize the fitness function.

One way to set things up is to make three strings of length 12 (for the number of items we could pack). One vector would store the values, one vector would store the weights, and the third vector could be binary, with 1 meaning "include this", and 0 meaning "do not include this".

Since we have 12 items total, we might define a chromosome as a vector with 1 (include this item), or 0 (do not include).

Exercises with the Knapsack Problem

We're going to go through the first example of a GA, the example with binary strings of length 20, and modify it to try to solve the knapsack problem.

Here are some suggestions for changes:

1. The objective function won't be so simple this time. Change the line:

   ```
   ff=inline(sum(x,2)); % objective function
   ```

   so that we can write the objective function as an M-file (like the second example).

   In the objective function, if X is a matrix that is 20×12 of zeros and ones, and `val` is a 1×12 vector of the "values" of each item, then what does the following command do? (You might look up the `repmat` command and what the `sum` command does if you use it like: `sum(A,2)`).

   ```
   sum(X.*repmat(val, numpop, 1), 2)
   ```

2. Some of the other parameters:
   - Change the maximum number of iterations to 250
   - We'll have a population size of 20, and we'll keep half. (The second example shows this).
   - The mutation rate will be about 10%

3. We'll keep the crossover technique and the way of selecting mates the same as for the binary strings.

Item	Value	Weight
bug repellent	12	2
camp stove	5	4
canteen (full)	10	7
clothes	11	5
dried food	50	3
first aid kit	15	3
flashlight	6	2
novel	4	2
rain gear	5	2
sleeping bag	25	3
tent	20	11
water filter	30	1

Table 3.1: Camping Supplies, with Weight and Value

Chapter 4

Linear Algebra

When you work with numerical data, it is naturally organized as a (possibly multidimensional) matrix. Therefore, linear algebra provides the natural context for working with data, and the theory of linear algebra can also give us methods for performing an analysis of that data.

First up is a little review.

It can be argued that all of linear algebra can be understood using the *four fundamental subspaces* associated with a matrix. Because they form the foundation on which we later work, we want an explicit method for analyzing these subspaces- That method will be the *Singular Value Decomposition* (SVD).

Finally, the SVD will give us the tools for performing linear regression analysis, projection analysis, and Principle Components Analysis (PCA).

Whenever we are analyzing new data sets, we can look to the SVD and PCA as the first tools we use to see if we can extract "features" from the data (more on that later).

4.1 Representation, Basis and Dimension

Let us quickly review some notation and basic ideas from linear algebra. First and foremost, we have to define the vector space that we're working with. For example, our "vectors" could be column vectors with n elements in them. Our "vectors" could actually be matrices that are all $m \times n$. The vector space defines the background to which all of our other analysis takes place, so it's an important consideration.

We will typically define our vector space V by constructing a **basis** for it, which is a **linearly independent** and **spanning** set of vectors, and the **dimension** of V is the number of its basis vectors.

Vector Space $\mathbb{R}^n$

Suppose the vector space is $\mathbb{R}^n$, and let H be a subspace of $\mathbb{R}^n$ with basis vectors in set $\mathcal{B}$. Since we're in $\mathbb{R}^n$, we can organize these vectors as columns in a matrix B (so B would be $n \times k$):

$$\mathcal{B} = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}, \qquad B = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k].$$

Then every vector in H can be written as a linear combination of the basis vectors. That is, if $\boldsymbol{x} \in H$,

$$\boldsymbol{x} = c_1\boldsymbol{v}_1 + \ldots + c_k\boldsymbol{v}_k$$

An important observation is that the linear combination you see above can be written in matrix-vector form:

$$\boldsymbol{x} = c_1\boldsymbol{v}_1 + \ldots + c_k\boldsymbol{v}_k = B[\boldsymbol{x}]_{\mathcal{B}} \tag{4.1}$$

where $(c_1, c_2, \ldots, c_k)^T = [\boldsymbol{x}]_{\mathcal{B}}$ is called the **coordinates of $\boldsymbol{x}$** with respect to the basis $\mathcal{B}$.

Now, consider that every vector in H can be identified with a point in $\mathbb{R}^k$, and that defines a function that we'll call the **coordinate mapping**:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n \longleftrightarrow [\boldsymbol{x}]_{\mathcal{B}} = \begin{bmatrix} c_1 \\ \vdots \\ c_k \end{bmatrix} \in \mathbb{R}^k$$

If k is small (with respect to n) we think of $\boldsymbol{c}$ is the **low dimensional representation** of the vector $\boldsymbol{x}$, and that H is *isomorphic* to $\mathbb{R}^k$ (Isormorphic meaning one to one and onto linear map is the isormorphism)

Example 4.1.1. If $H = \text{span}\,(\boldsymbol{v}_1, \boldsymbol{v}_2)$ where $\boldsymbol{v}_{1,2} \in \mathbb{R}^5$, then H is *isomorphic* to the plane $\mathbb{R}^2$ - but is not *equal* to the plane. The isomorphism is the coordinate mapping (call it F). For example, if $\mathbf{x} \in H$, then

$$\mathbf{x} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2$$

and $F(\mathbf{x}) = (c_1, c_2) \in \mathbb{R}^2$.

Computing the Coordinates

Generally, finding the coordinates of $\boldsymbol{x}$ with respect to an arbitrary basis (as columns of a matrix V) means that we have to solve the matrix equation for $[\boldsymbol{x}]_{\mathcal{B}}$:

$$\mathbf{x} = B[\boldsymbol{x}]_{\mathcal{B}}.$$

However, if the columns of B are actually orthogonal, then it is very simple to compute the coordinates. We'll start with our vector equation: $\boldsymbol{x} = c_1\boldsymbol{v}_1 + \cdots + c_k\boldsymbol{v}_k$. Now take the inner product of both sides with $\boldsymbol{v}_j$:

$$\boldsymbol{x} \cdot \boldsymbol{v}_j = c_1\boldsymbol{v}_1 \cdot \boldsymbol{v}_j + \cdots + c_j\boldsymbol{v}_j \cdot \boldsymbol{v}_j + \cdots + c_k\boldsymbol{v}_k \cdot \boldsymbol{v}_j$$

All the dot products are 0 (due to orthogonality) except for the dot product with $\boldsymbol{x}$ and $\boldsymbol{v}_j$ leading to the formula:

$$\boldsymbol{x} \cdot \boldsymbol{v}_j = 0 + 0 + \ldots + c_j\boldsymbol{v}_j \cdot \boldsymbol{v}_j + \cdots + 0 \quad \Rightarrow \quad c_j = \frac{\boldsymbol{x} \cdot \boldsymbol{v}_j}{\boldsymbol{v}_j \cdot \boldsymbol{v}_j}$$

And we remember that this is the scalar projection of $\boldsymbol{x}$ onto $\boldsymbol{v}_j$, which we've seen in Calc III and linear algebra:

$$\text{Proj}_v(\mathbf{u}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}\mathbf{v}$$

Therefore, we can think of the linear combination as the following, which simplifies if we use **orthonormal basis vectors**:

$$\begin{aligned} \mathbf{x} &= \text{Proj}_{v_1}(\mathbf{x}) + \text{Proj}_{v_2}(\mathbf{x}) + \cdots + \text{Proj}_{v_k}(\mathbf{x}) \\ &= (\mathbf{x} \cdot \mathbf{v}_1)\mathbf{v}_1 + (\mathbf{x} \cdot \mathbf{v}_2)\mathbf{v}_2 + \cdots + (\mathbf{x} \cdot \mathbf{v}_k)\mathbf{v}_k \end{aligned} \tag{4.2}$$

IMPORTANT NOTE: In the event that $\mathbf{x}$ is NOT in H, then Equation 4.2 gives the (orthogonal) **projection of $\mathbf{x}$** into H.

The important conclusion we want to make is the following: Out of all possible bases we might choose for a subspace of a vector space, the "best" one is one that is orthonormal, since the coordinates of a vector are easy to compute using the inner product. In fact, if the space is a subspace of $\mathbb{R}^n$ and the (orthonormal) basis is stored as the columns of a matrix B ($n \times k$ in our previous example), then the coordinates of any $\mathbf{x}$ can be simply computed as:

$$[\mathbf{x}]_{\mathcal{B}} = B^T\mathbf{x}$$

Of course, you might be noting that in that case,

$$\mathbf{x} = B[\mathbf{x}]_{\mathcal{B}} = BB^T\mathbf{x}$$

This matrix, BB^T, is a very interesting one- One which we examine more closely in the next section.

Projections

Consider the following example. If a matrix $U = [\mathbf{u}_1, \ldots, \mathbf{u}_k]$ has orthonormal columns (so if U is $n \times k$, then that requires $k \leq n$), then U^TU is $k \times k$, and can be computed as:

$$U^TU = \begin{bmatrix} \mathbf{u}_1^T \\ \vdots \\ \mathbf{u}_k \end{bmatrix} [\mathbf{u}_1, \ldots, \mathbf{u}_k] = \begin{bmatrix} \mathbf{u}_1^T\mathbf{u}_1 & \mathbf{u}_1^T\mathbf{u}_2 & \cdots & \mathbf{u}_1^T\mathbf{u}_k \\ \mathbf{u}_2^T\mathbf{u}_1 & \mathbf{u}_2^T\mathbf{u}_2 & \cdots & \mathbf{u}_2^T\mathbf{u}_k \\ \vdots & & & \vdots \\ \mathbf{u}_k^T\mathbf{u}_1 & \mathbf{u}_k^T\mathbf{u}_2 & \cdots & \mathbf{u}_k^T\mathbf{u}_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = I_k$$

But UU^T (which is $n \times n$) is NOT the identity if $k \neq n$ (If $k = n$, then the previous computation proves that the inverse is the transpose).

Here is a computation one might make for UU^T (these are OUTER products):

$$UU^T = [\mathbf{u}_1, \ldots, \mathbf{u}_k] \begin{bmatrix} \mathbf{u}_1^T \\ \vdots \\ \mathbf{u}_k \end{bmatrix} = \mathbf{u}_1\mathbf{u}_1^T + \mathbf{u}_2\mathbf{u}_2^T + \cdots + \mathbf{u}_k\mathbf{u}_k^T$$

Example 4.1.2. Consider the following computations:

$$U = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad U^TU = 1 \qquad UU^T = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

If UU^T is not the identity, what is it? Consider the following computation:

$$UU^T\mathbf{x} = \mathbf{u}_1\mathbf{u}_1^T\mathbf{x} + \mathbf{u}_2\mathbf{u}_2^T\mathbf{x} + \cdots + \mathbf{u}_k\mathbf{u}_k^T\mathbf{x}$$

$$= \mathbf{u}_1(\mathbf{u}_1^T\mathbf{x}) + \mathbf{u}_2(\mathbf{u}_2^T\mathbf{x}) + \cdots + \mathbf{u}_k(\mathbf{u}_k^T\mathbf{x})$$

which we recognize as the projection of $\mathbf{x}$ into the space spanned by the orthonormal vectors of U. In the previous section, we called this matrix B, but it is common notation that a matrix with orthonormal vectors is typically denoted by U.

Just to repeat then: If we have a set of orthonormal vectors in a matrix U forming a basis for some subspace H, then if $\mathbf{x} \in H$,

$$\mathbf{x} = U[\mathbf{x}]_U = U(U^T\mathbf{x}) = (UU^T)\mathbf{x}$$

However, if $\mathbf{x}$ is not completely contained in H, then

$$\text{Proj}_H(\mathbf{x}) = UU^T\mathbf{x}$$

And of course, if $n > k$ and U is $n \times k$, then $U^TU = I$.

Notes about Programming

Programming in Matlab

- Find a random matrix with orthonormal columns

  SOLUTION: Use the "QR" decomposition of a matrix A. That is, decompose matrix A as a product of matrix Q (with orthonormal columns) times matrix R.

```
X=randn(9,6);
[Q,R]=qr(X,0);
Q'*Q           % Should be 6 x 6 identity matrix.
```

- Take a matrix X and "normalize" it by making each column have norm 1.

  SOLUTION: Find a row representing the norm of each column, then divide each row of the matrix by that row.

```
X=[1,0;0,1;1,0]
RowNorms=sqrt(sum(X.*X));
C=X./RowNorms
```

- Let $\boldsymbol{x}$ be a random vector in $\mathbb{R}^9$. We're going to project this into the subspace spanned by the first three columns of the matrix Q that we constructed previously.

  - First, what are the coordinates? In linear algebra, the coordinates (a vector in $\mathbb{R}^3$) are: $Q^T\boldsymbol{x}$, where this Q is 9×3.
  - Next, project this vector into the subspace: In linear algebra notation, $QQ^T\boldsymbol{x}$

  Here it is in Matlab, using the Q from the QR decomposition.

```
x=rand(9,1);
Coords=Q(:,1:3)'*x;
Projx=Q(:,1:3)*Coords;
Check=Q*(Q'*Projx);  % Don't compute QQ^T first!
norm(Check-Projx);  %Should be close to zero
```

Programming in Python

- Find a random matrix with orthonormal columns.

  SOLUTION: Use the "QR" decomposition of a matrix A. That is, decompose matrix A as a product of matrix Q (with orthonormal columns) times matrix R.

```
A=np.random.randn(9,6)
q,r=np.linalg.qr(A)
q.shape              #Answer is (9,6)
D=np.matmul(q.T,q)   #Should be 6 x 6 identity
```

- Take a matrix X and "normalize" it by making each column have norm 1.

  SOLUTION: Find a row representing the norm of each column, then divide each row of the matrix by that row.

```
import numpy as np
import numpy.linalg

X=np.array([[1, 0],[0,1],[1,0]])
RowNorms=np.linalg.norm(X,axis=0)
C=X/RowNorms[np.newaxis,:]
Output:
array([[0.70710678, 0.        ],
       [0.        , 1.        ],
       [0.70710678, 0.        ]])
```

- Let $\boldsymbol{x}$ be a random vector in $\mathbb{R}^9$. We're going to project this into the subspace spanned by the first three columns of the matrix Q that we constructed previously.

- First, what are the coordinates? In linear algebra, the coordinates (a vector in $\mathbb{R}^3$) are: $Q^T\boldsymbol{x}$, where this Q is 9×3.
- Next, project this vector into the subspace: In linear algebra notation, $QQ^T\boldsymbol{x}$

```
import numpy as np
import numpy.linalg

x=np.random.rand(9,1)  #Random vector in R^9
A=np.random.rand(9,3)  #We'll go with a 3-d subspace in R^9
Q,R=np.linalg.qr(A)

Q.shape   #This checks the dimensions- This is 9 x 3

Coords=np.matmul(Q.T,x)  #This is a 3 x 1 vector, Q^Tx
xProjected=np.matmul(Q,Coords)  #xProjected is 9 x 1

#Optional: Check by projecting the projection (shouldn't change)
Check=np.matmul(Q,np.matmul(Q.T,xProjected))
np.linalg.norm(xProjected-Check)  #Returns something like 2 x 10^(-16)
```

Programming in R

- Find a random matrix with orthonormal columns

  SOLUTION: Use the "QR" decomposition of a matrix A. That is, decompose matrix A as a product of matrix Q (with orthonormal columns) times matrix R.

```
A<-matrix(rnorm(54),nrow=9)   # rnorm are random, normal dist.
X=qr(A)                       # X is a QR-object
Q<-qr.Q(X)                    # Extract the matrix Q
dim(Q)                        # Check dimensions on Q (returns (9,6))
H<- t(Q) %*% Q                # Compute Q^TQ to get 6 x 6 identity
```

- Take a matrix X and "normalize" it by making each column have norm 1.

  SOLUTION: Find a row representing the norm of each column, then divide each row of the matrix by that row.

```
X<-matrix(c(1,1,0,0,1,0),3,2)
N<-sqrt(colSums(X^2)
C<-sweep(X,2,N,FUN='/')
```

- Let $\boldsymbol{x}$ be a random vector in $\mathbb{R}^9$. We're going to project this into the subspace spanned by the first three columns of the matrix Q that we constructed previously.

  - First, what are the coordinates? In linear algebra, the coordinates (a vector in $\mathbb{R}^3$) are: $Q^T\boldsymbol{x}$, where this Q is 9×3.
  - Next, project this vector into the subspace: In linear algebra notation, $QQ^T\boldsymbol{x}$

```
> x<-matrix(runif(9),ncol=1)
> A<-matrix(runif(27),ncol=3)
> X<-qr(A)
> Q<-qr.Q(X)            #This is a bit awkward
```

```
> Coords<-t(Q)%*%x
> xProjected<-Q %*% Coords
> Check=Q %*% (t(Q) %*% xProjected)
> y=xProjected-Check
> sqrt(sum(y^2))
[1] 5.360492e-16
```

Exercises

1. Let the subspace H be formed by the span of the vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$ given below. Given the point $\boldsymbol{x}_1, \boldsymbol{x}_2$ below, find which one belongs to H, and if it does, give its coordinates. (NOTE: The basis vectors are NOT orthonormal)

$$\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix} \quad \boldsymbol{v}_2 = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \quad \boldsymbol{x}_1 = \begin{bmatrix} 7 \\ 4 \\ 0 \end{bmatrix} \quad \boldsymbol{x}_2 = \begin{bmatrix} 4 \\ 3 \\ -1 \end{bmatrix}$$

2. Show that the plane H defined by:

$$H = \left\{ \alpha_1 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} \text{ such that } \alpha_1, \alpha_2 \in \mathbb{R} \right\}$$

is isormorphic to $\mathbb{R}^2$.

3. Let the subspace G be the plane defined below, and consider the vector $\boldsymbol{x}$, where:

$$G = \left\{ \alpha_1 \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix} + \alpha_2 \begin{bmatrix} 3 \\ -1 \\ 0 \end{bmatrix} \text{ such that } \alpha_1, \alpha_2 \in \mathbb{R} \right\} \qquad \boldsymbol{x} = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}$$

   (a) Find the matrix (UU^T in our notes) that takes an arbitrary vector and projects it (orthogonally) to the plane G.

   (b) Find the orthogonal projection of the given $\boldsymbol{x}$ onto the plane G.

   (c) Find the distance from the plane G to the vector $\boldsymbol{x}$.

4. If the low dimensional representation of a vector $\boldsymbol{x}$ is $[9, -1]^T$ and the basis vectors are $[1, 0, 1]^T$ and $[3, 1, 1]^T$, then what was the original vector $\boldsymbol{x}$? (HINT: it is easy to compute it directly)

5. If the vector $\boldsymbol{x} = [10, 4, 2]^T$ and the basis vectors are $[1, 0, 1]^T$ and $[3, 1, 1]^T$, then what is the low dimensional representation for $\boldsymbol{x}$?

6. Let $\boldsymbol{a} = [-1, 3]^T$. Find a square matrix P so that $P\boldsymbol{x}$ is the orthogonal projection of $\boldsymbol{x}$ onto the span of $\boldsymbol{a}$.

7. Refer to one of the programming languages (Matlab/Python/R), and reproduce finding an arbitrary 10×4 matrix with orthonormal columns. Use a random $\boldsymbol{x} \in \mathbb{R}^{10}$, and first find the coordinates of $\boldsymbol{x}$ with respect to the four columns in Q, then compute the orthogonal projection of $\boldsymbol{x}$ into the subspace spanned by the first four columns of Q.

8. To prove that we have an *orthogonal* projection, the vector $\text{Proj}_u(\boldsymbol{x}) - \boldsymbol{x}$ should be orthogonal to $\boldsymbol{u}$. Use this definition to show that our earlier formula was correct- that is,

$$\text{Proj}_u(\boldsymbol{x}) = \frac{\boldsymbol{x} \cdot \boldsymbol{u}}{\boldsymbol{u} \cdot \boldsymbol{u}} \boldsymbol{u}$$

is the orthogonal projection of $\mathbf{x}$ onto $\mathbf{u}$.

9. Continuing with the last exercise, show that $UU^T\mathbf{x}$ is the *orthogonal* projection of $\mathbf{x}$ into the space spanned by the columns of U by showing that $(UU^T\boldsymbol{x} - \boldsymbol{x})$ is orthogonal to $\mathbf{u}_i$ for any $i = 1, 2, \cdots, k$.

4.2 The Four Fundamental Subspaces

Given any $m \times n$ matrix A, we consider the mapping $A : \mathbb{R}^n \to \mathbb{R}^m$ by:

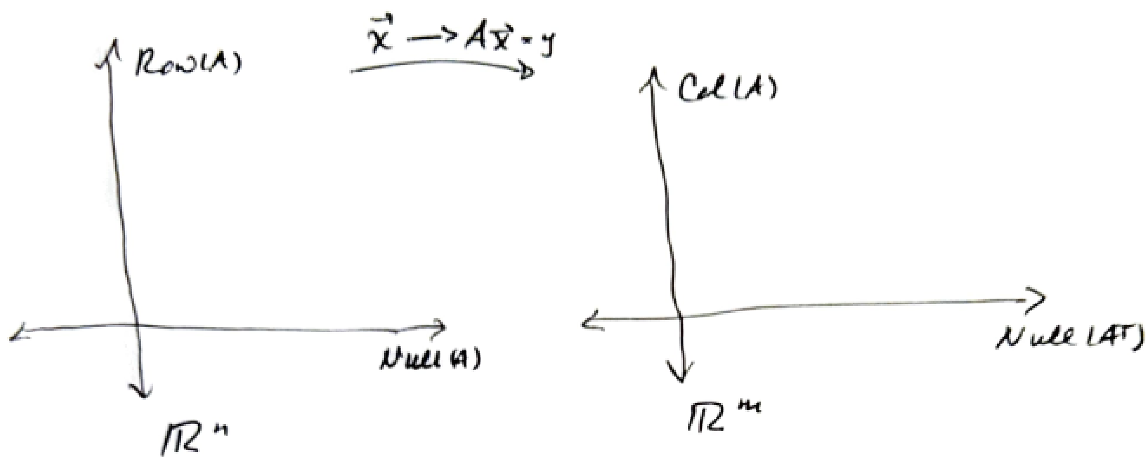
$$\boldsymbol{x} \to A\boldsymbol{x} = \boldsymbol{y}$$

The four subspaces allow us to completely understand the domain and range of the mapping. We will first define them, then look at some examples.

Definition 4.2.1. The Four Fundamental Subspaces

- The **row space** of A is a subspace of $\mathbb{R}^n$ formed by taking all possible linear combinations of the rows of A. Formally,
$$\text{Row}(A) = \left\{\boldsymbol{x} \in \mathbb{R}^n \,|\, x = A^T\boldsymbol{y} \;\; y \in \mathbb{R}^m\right\}$$
- The **null space** of A is a subspace of $\mathbb{R}^n$ formed by
$$\text{Null}(A) = \{\boldsymbol{x} \in \mathbb{R}^n \,|\, A\boldsymbol{x} = \mathbf{0}\}$$
- The **column space** of A is a subspace of $\mathbb{R}^m$ formed by taking all possible linear combinations of the columns of A.
$$\text{Col}(A) = \{\boldsymbol{y} \in \mathbb{R}^m \,|\, y = A\boldsymbol{x} \quad \in \mathbb{R}^n\}$$
The column space is also the image of the mapping. Notice that $A\boldsymbol{x}$ is simply a linear combination of the columns of A:
$$A\boldsymbol{x} = x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \cdots + x_n\boldsymbol{a}_n$$
- Finally, we define the **null space** of A^T can be defined in the obvious way (see the Exercises).

The fundamental subspaces subdivide the domain and range of the mapping in a particularly nice way. Below we give a "cartoon" of the relationship between the four subspaces. On the left is the domain of the matrix mapping, and it represents $\mathbb{R}^n$. On the right is the codomain, and it represents $\mathbb{R}^m$. The spaces can apparently be split into two subspaces each. In the domain, these are the Row and Null spaces. In the codomain, these are the Column space and the null space of A^T (we don't use this one much).



In the diagram, notice that the axes are representing subspaces. The picture suggests that the two spaces that split our domain and range are actually orthogonal subspaces- and that is true.

Theorem 4.2.1. *Let A be an $m \times n$ matrix. Then*

- *The nullspace of A is orthogonal to the row space of A*
- *The nullspace of A^T is orthogonal to the columnspace of A*

Proof: We'll prove the first statement, the second statement is almost identical to the first. To prove the first statement, we have to show that if we take any vector $\mathbf{x}$ from nullspace of A and any vector $\mathbf{y}$ from the row space of A, then $\mathbf{x} \cdot \mathbf{y} = 0$.

Alternatively, if we can show that $\mathbf{x}$ is orthogonal to each and every row of A, then we're done as well (since $\mathbf{y}$ is a linear combination of the rows of A).

In fact, now we see a strategy: Write out what it means for $\mathbf{x}$ to be in the nullspace using the rows of A. For ease of notation, let $\mathbf{a}_j$ denote the jth **row** of A, which will have size $1 \times n$. Then:

$$A\mathbf{x} = \mathbf{0} \quad \Rightarrow \quad \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_m \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1\mathbf{x} \\ \mathbf{a}_2\mathbf{x} \\ \vdots \\ \mathbf{a}_m\mathbf{x} \end{bmatrix} = \mathbf{0}$$

Therefore, the dot product between any row of A and $\mathbf{x}$ is zero, so that $\mathbf{x}$ is orthogonal to every row of A. Therefore, $\mathbf{x}$ must be orthogonal to any linear combination of the rows of A, so that $\mathbf{x}$ is orthogonal to the row space of A. $\Box$

Before going further, let us recall how to construct a basis for the column space, row space and nullspace of a matrix A. We'll do it with a particular matrix:

Example 4.2.1. Construct a basis for the column space, row space and nullspace of the matrix A below that is row equivalent to the matrix beside it, RREF(A):

$$A = \begin{bmatrix} 2 & 0 & -2 & 2 \\ -2 & 5 & 7 & 3 \\ 3 & -5 & -8 & -2 \end{bmatrix} \qquad \text{RREF}(A) = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The first two columns of the original matrix form a basis for the columnspace (which is a subspace of $\mathbb{R}^3$):

$$\text{Col}(A) = \text{span}\left\{ \begin{bmatrix} 2 \\ -2 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ -2 \\ 3 \end{bmatrix} \right\}$$

A basis for the row space is found by using the row reduced rows corresponding to the pivots (and is a subspace of $\mathbb{R}^4$). You should also verify that you can find a basis for the null space of A, given below (also a subspace of $\mathbb{R}^4$). If you're having any difficulties here, be sure to look it up in a linear algebra text:

$$\text{Row}(A) = \text{span}\left\{ \begin{bmatrix} 1 \\ 0 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right\} \qquad \text{Null}(A) = \text{span}\left\{ \begin{bmatrix} 1 \\ -1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \\ 0 \\ 1 \end{bmatrix} \right\}$$

We will often refer to the dimensions of the four subspaces. We recall that there is a term for the dimension of the column space- That is, the rank.

Definition 4.2.2. The *rank* of a matrix A is the number of independent columns of A.

In our previous example, the rank of A is 2. Also from our example, we see that the rank is the dimension of the column space, and that this is the same as the dimension of the row space (all three numbers correspond to the number of pivots in the row reduced form of A). Finally, a handy theorem for counting is the following.

The Rank Theorem. Let the $m \times n$ matrix A have rank r. Then

$$r + \dim(\text{Null}(A)) = n$$

This theorem says that the number of pivot columns plus the other columns (which correspond to free variables) is equal to the total number of columns.

Example 4.2.2. The Dimensions of the Subspaces.
Given a matrix A that is $m \times n$ with rank k, then the dimensions of the four subspaces are shown below.

- $\dim(\text{Row}(A)) = k$
- $\dim(\text{Null}(A)) = n - k$
- $\dim(\text{Col}(A)) = k$
- $\dim\left(\text{Null}(A^T)\right) = m - k$

There are some interesting implications of these theorems to matrices of data- For example, suppose A is $m \times n$. With no other information, we do not know whether we should consider this matrix as n points in $\mathbb{R}^m$, or m points in $\mathbb{R}^n$. In one sense, it doesn't matter! The theorems we've discussed shows that the dimension of the columnspace is equal to the dimension of the rowspace. Later on, we'll find out that if we can find a basis for the columnspace, it is easy to find a basis for the rowspace. We'll need some more machinery first.

4.3 Exercises

In the exercises below, recall that the usual norm for a vector is the Euclidean norm, or the $2-norm$, which is defined as:

$$\|\mathbf{x}\| = \sqrt{x_1^1 + x_2^2 + \cdots x_n^2} = \sqrt{\mathbf{x}^T\mathbf{x}}$$

1. Show that $\text{Null}(A^T) \perp \text{Col}(A)$. Hint: You may use what we already proved.
2. If A is $m \times n$, how big can the rank of A possibly be?
3. Show that multiplication by an orthogonal matrix preserves lengths: $\|\mathbb{Q}\boldsymbol{x}\|_2 = \|\boldsymbol{x}\|_2$ (Hint: Use properties of inner products). Conclude that multiplication by $\mathbb{Q}$ represents a rigid rotation.
4. Prove the Pythagorean Theorem by induction: Given a set of n orthogonal vectors $\{\boldsymbol{x}_i\}$

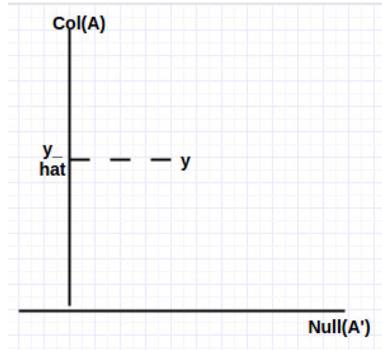
   $$\|\sum_{i=1}^{n} \boldsymbol{x}_i\|^2 = \sum_{i=1}^{n} \|\boldsymbol{x}_i\|^2$$

   The case where $n = 1$ is trivial, so you might look at $n = 2$ first. Try starting with

   $$\|\mathbf{x} + \mathbf{y}\|^2 = (\mathbf{x} + \mathbf{y})^T(\mathbf{x} + \mathbf{y}) = \cdots$$

   and then simplify to get $\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2$. Now try the induction step on your own.
5. Let A be an $m \times n$ matrix where $m > n$, and let A have rank n. Let $\boldsymbol{y}, \hat{\boldsymbol{y}} \in \mathbb{R}^m$, such that $\hat{\boldsymbol{y}}$ is the orthogonal projection of $\boldsymbol{y}$ onto the column space of A. We want a formula for the matrix $\mathbb{P} : \mathbb{R}^m \to \mathbb{R}^m$ so that $\mathbb{P}\boldsymbol{y} = \hat{\boldsymbol{y}}$.

   The following image shows the relevant subspaces:



(a) Why is the projector not $\mathbb{P} = AA^T$?

(b) Since $\hat{\boldsymbol{y}} - \boldsymbol{y}$ is orthogonal to the column space of A, show that

$$A^T(\hat{\boldsymbol{y}} - \boldsymbol{y}) = \mathbf{0} \tag{4.3}$$

(c) Show that there exists $\boldsymbol{x} \in \mathbb{R}^n$ so that Equation (4.3) can be written as:

$$A^T(A\boldsymbol{x} - \boldsymbol{y}) = 0 \tag{4.4}$$

(d) Argue that A^TA (which is $n \times n$) is invertible, so that Equation (13.2) implies that

$$\boldsymbol{x} = \left(A^TA\right)^{-1} A^T\boldsymbol{y}$$

(e) Finally, show that this implies that

$$\mathbb{P} = A\left(A^TA\right)^{-1} A^T$$

Note: If A has rank $k \neq n$, then we will need something different, since A^TA will not be full rank. The missing piece is the singular value decomposition, to be discussed later.

6. The Orthogonal Decomposition Theorem: if $\boldsymbol{x} \in \mathbb{R}^n$ and W is a (non-zero) subspace of $\mathbb{R}^n$, then $\boldsymbol{x}$ can be written *uniquely* as

$$\boldsymbol{x} = \boldsymbol{w} + \boldsymbol{z}$$

where $\boldsymbol{w} \in W$ and $\boldsymbol{z} \in W^\perp$.

To prove this, let $\{\boldsymbol{u}_i\}_{i=1}^p$ be an orthonormal basis for W, define $\boldsymbol{w} = (\boldsymbol{x} \cdot \boldsymbol{u}_1)\boldsymbol{u}_1 + \ldots + (\boldsymbol{x} \cdot \boldsymbol{u}_p)\boldsymbol{u}_p$, and define $\boldsymbol{z} = \boldsymbol{x} - \boldsymbol{w}$. Then:

(a) Show that $\boldsymbol{z} \in W^\perp$ by showing that it is orthogonal to every $\boldsymbol{u}_i$.

(b) To show that the decomposition is unique, suppose it is not. That is, there are two decompositions:

$$\boldsymbol{x} = \boldsymbol{w}_1 + \boldsymbol{z}_1, \quad \boldsymbol{x} = \boldsymbol{w}_2 + \boldsymbol{z}_2$$

Show this implies that $\boldsymbol{w}_1 - \boldsymbol{w}_2 = \boldsymbol{z}_2 - \boldsymbol{z}_1$, and that this vector is in both W and $W^\perp$. What can we conclude from this?

7. Use the previous exercises to prove the **The Best Approximation Theorem** If W is a subspace of $\mathbb{R}^n$ and $\boldsymbol{x} \in \mathbb{R}^n$, then the point closest to $\boldsymbol{x}$ in W is the orthogonal projection of $\boldsymbol{x}$ into W.

4.4 The Decomposition Theorems

The matrix factorization that arises from an eigenvector/eigenvalue decomposition is useful in many applications, so we'll briefly review it here and build from it until we get to our main goal, the Singular Value Decomposition.

4.4.1 The Eigenvector/Eigenvalue Decomposition

First we have a basic definition:

Let A be an $n \times n$ matrix. If there exists a scalar λ and non-zero vector $\boldsymbol{v}$ so that

$$A\boldsymbol{v} = \lambda \boldsymbol{v}$$

then we say that λ is an eigenvalue and $\boldsymbol{v}$ is an associated eigenvector.

An equivalent formulation of the problem is to solve $A\mathbf{v} - \mathbf{v} = \mathbf{0}$, or, factoring $\mathbf{v}$ out,

$$(A - \lambda I)\mathbf{v} = \mathbf{0}$$

This equation always has the zero solution (letting $\mathbf{v} = \mathbf{0}$), however, we need to have non-trivial solutions, and the only way that will happen is if $A - \lambda I$ is non-invertible, or equivalently,

$$\det(A - \lambda I) = 0$$

which, when multiplied out, is a polynomial equation in λ that is called the **characteristic equation**.

Therefore, we find the eigenvalues first, then for each λ, there is an associated subspace- The null space of $A - \lambda I$, or the **eigenspace** associated with λ, denoted by E_λ.

The way to finish the problem is to give a "nice" basis for the eigenspace- If you're working by hand, try one with integer values. If you're on the computer, it is often convenient to make them unit vectors.

Some vocabulary associated with eigenvalues: Solving the characteristic equation will mean that we can have repeated solutions. The number of repetitions is the *algebraic multiplicity* of λ. On the other hand, for each λ, we find the eigenspace which will have a certain dimension- The dimension of the eigenspace is the *geometric multiplicity* of λ.

Examples:

1. Compute the eigenvalues and eigenvectors for the 2×2 identity matrix.

   SOLUTION: The eigenvalue is 1 (a double root), so the algebraic multiplicity of 1 is 2.

   On the other hand, if we take $A - \lambda I$, we simply get the zero matrix, which implies that every vector in $\mathbb{R}^2$ is an eigenvector. Therefore, we can take any basis of $\mathbb{R}^2$ is a basis for E_1, and the geometric multiplicity is 2.

2. Consider the matrix

   $$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

   Again, the eigenvalue 1 is a double eigenvalue (so the algebraic multiplicity is 2), but solving $(A - \lambda I)\boldsymbol{v} = 0$ gives us:

   $$2v_2 = 0 \quad \Rightarrow \quad v_2 = 0$$

   That means v_1 is free, and the basis for E_1 is $[1, 0]^T$. Therefore, the algebraic multiplicity is 1.

Definition: A matrix for which the algebraic and geometric multiplicities are not equal is called *defective*.

There is a nice theorem relating eigenvalues:
Theorem: If X is square and invertible, then A and $X^{-1}AX$ have the same eigenvalues.
Sometimes this method of characterizing eigenvalues in terms of the determinant and trace of a matrix:

$$\det(A) = \Pi_{i=1}^{n}\lambda_i \quad \text{trace}(A) = \sum_{i=1}^{\infty}\lambda_i$$

Symmetric Matrices and the Spectral Theorem

There are some difficulties working with eigenvalues and eigenvectors of a general matrix. For one thing, they are only defined for square matrices, and even when they are defined, we may get real or complex eigenvalues.

If a matrix is symmetric, beautiful things happen with the eigenvalues and eigenvectors, and it is summarized below in the Spectral Theorem.

The Spectral Theorem: If A is an $n \times n$ symmetric matrix, then:

1. A has n real eigenvalues (counting multiplicity).
2. For each distinct λ, the algebraic and geometric multiplicities are the same.
3. The eigenspaces are mutually orthogonal- both for distinct eigenvalues, and we'll take each E_λ to have an orthonormal basis.
4. A is orthogonally diagonalizeable, with $D = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$. That is, if V is the matrix whose columns are the (othornormal) eigenvectors of A, then

$$A = VDV^T$$

Some remarks about the Spectral Theorem:

- If a matrix is real and symmetric, the Spectral Theorem says that its eigenvectors form an orthonormal basis for $\mathbb{R}^n$.
- The first part is somewhat difficult to prove in that we would have to bring in more machinery than we would like. If you would like to see a proof, it comes from the *Schur Decomposition*, which is given, for example, in "Matrix Computations" by Golub and Van Loan.

The following is a proof of the third part. Supply justification for each step: Let $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ be eigenvectors from distinct eigenvalues, λ_1, λ_2. We show that $\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = 0$:

$$\lambda_1 \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = (A\boldsymbol{v}_1)^T \boldsymbol{v}_2 = \boldsymbol{v}_1^T A^T \boldsymbol{v}_2 = \boldsymbol{v}_1^T A \boldsymbol{v}_2 = \lambda_2 \boldsymbol{v}_1 \cdot \boldsymbol{v}_2$$

Now, $(\lambda_1 - \lambda_2)\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = 0$.

The Spectral Decomposition: Since A is orthogonally diagonalizable, then

$$A = (\boldsymbol{q}_1\ \boldsymbol{q}_2\ \ldots\ \boldsymbol{q}_n) \begin{pmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{pmatrix} \begin{pmatrix} \boldsymbol{q}_1^T \\ \boldsymbol{q}_2^T \\ \vdots \\ \boldsymbol{q}_n^T \end{pmatrix}$$

so that:

$$A = (\lambda_1 \boldsymbol{q}_1 \ \lambda_2 \boldsymbol{q}_2 \ \ldots \ \lambda_n \boldsymbol{q}_n) \begin{pmatrix} \boldsymbol{q}_1^T \\ \boldsymbol{q}_2^T \\ \vdots \\ \boldsymbol{q}_n^T \end{pmatrix}$$

so finally:

$$A = \lambda_1 \boldsymbol{q}_1 \boldsymbol{q}_1^T + \lambda_2 \boldsymbol{q}_2 \boldsymbol{q}_2^T + \ldots + \lambda_n \boldsymbol{q}_n \boldsymbol{q}_n^T$$

That is, A is a sum of n rank one matrices, each of which is a projection matrix.

Exercises

1. Prove that if X is invertible and matrix A is square, then A and $X^{-1}AX$ have the same eigenvalues.

2. **Programming Exercise:** Verify the spectral decomposition for a symmetric matrix. Here are examples in Matlab, Python and R. First, we'll construct a random symmetric 6×6 matrix, then we'll compute the eigenvalues and eigenvectors.

   - Matlab:

   ```
   %Construct a random, symmetric, 6 x 6 matrix:
   N=6;
   Atemp=rand(N,N);
   A=(Atemp+Atemp')/2;  %A is symmetric

   %Compute the eigenvalues of A:
   [Q,L]=eig(A);   %NOTE:  A = Q L Q'
                   %L is a diagonal matrix

   %Now form the "spectral sum"
   S=zeros(6,6);
   for i=1:6
     S=S+L(i,i)*Q(:,i)*Q(:,i)';
   end

   max(max(S-A))
   ```

   Note that the maximum of $S - A$ should be a very small number! (By the spectral decomposition theorem).

   - Python:

   ```
   import numpy as np
   import numpy.linalg

   N=6;
   Atemp=np.random.rand(N,N)
   A=(Atemp+Atemp.T)/2       #Makes A symmetric

   D,V=numpy.linalg.eig(A)
   S=np.zeros((6,6))

   for i in range(0,6):
       S=S+D[i]*np.outer(V[:,i],V[:,i])
   ```

```
print(numpy.linalg.norm(A-S,'fro'))
```

- R:

```
A<-matrix(runif(36),ncol=6)
A<-(A+t(A))/2 #Makes A symmetric
P<-eigen(A)  #Creates a structure holding info
D<-P$values
V<-P$vectors
S=matrix(0,nrow=6,ncol=6)
for (i in 1:6){
  S=S+D[i]*( V[,i] %*% t(V[,i]) )
}
print(norm(A-S,"F"))
```

4.4.2 The Singular Value Decomposition

There is a special matrix factorization that is extremely useful, both in applications and in proving theorems. This is mainly due to two facts, which we shall investigate in this section: (1) We can use this factorization on *any* matrix, (2) The factorization defines explicitly the rank of the matrix, and gives **orthonormal bases** for all **four fundamental subspaces** of A.

In what follows, assume that A is an $m \times n$ matrix (so A maps $\mathbb{R}^n$ to $\mathbb{R}^m$ and is not necessarily square). We'll build the factorization by using the Spectral Theorem twice.

Step 1. Although A itself is not symmetric, A^TA is $n \times n$ and symmetric, so the Spectral Theorem applies, and we define an $n \times n$ orthonormal matrix V and diagonal matrix D_1 (with eigenvalues λ along the diagonal): using that theorem:

$$A^TA = VDV^T$$

We will assume that the eigenvalues are ordered from largest to smallest.

Step 2. Similarly, the $m \times m$ matrix AA^T is also symmetric, and so applying the Spectral Theorem again defines an $m \times m$ orthonormal matrix U and diagonal matrix D_2 such that

$$AA^T = UD_2U^T$$

We will assume that the eigenvalues are ordered from largest to smallest.

Step 3. Later, we will show that the non-zero diagonal elements of D_1 and D_2 are identical. Assuming the rank of A to be k, we'll define **the singular values** of A as:

$$\sigma_i = \begin{cases} \sqrt{\lambda_i} & \text{if } 1 \le i \le k \\ 0 & \text{for the remaining values} \end{cases}$$

where λ_i is the ith diagonal element of D_1 or D_2. The $m \times n$ matrix Σ is defined to be the $m \times n$ diagonal matrix with singular values along the diagonal (ordered from largest to smallest).

Theorem **The Singular Value Decomposition (SVD)** Let A be any $m \times n$ matrix of rank k. Then we can factor the matrix A as the following product:

$$A = U\Sigma V^T$$

where U is an orthogonal $m \times m$ matrix, Σ is a diagonal $m \times n$ matrix, and V is an orthogonal $n \times n$ matrix. The columns of U are called the *left singular vectors* and the columns of V are called the *right singular vectors*. Further, there are exactly k non-zero singular values of A.

Before we get to the exercises, there is a different way of expressing the SVD that is similar to our spectral decomposition. Recall that the rank is k:

$$A = U\Sigma V^T = \sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^T + \sigma_2 \boldsymbol{u}_2 \boldsymbol{v}_2^T + \cdots + \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k^T$$

Please note that this is a sum of k matrices, not scalars- That is, we're computing outer products, $\sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T$ is an $m \times n$ matrix.

4.5 Exercises

In the exercises below, we'll look more closely at the SVD and the relationships between the vectors in U and V. As you're working through these, be sure to keep in mind the dimensions of the objects you're working with.

For all of the exercises, assume that A is $m \times n$ with rank k, and the SVD of A is $U\Sigma V^T$.

1. Show that if λ is an eigenvalue of $A^T A$, then λ is also an eigenvalue of AA^T.

   Hint: If λ is an eigenvalue of $A^T A$ with eigenvector $\mathbf{v}$, then $A^T A\mathbf{v} = \lambda\mathbf{v}$. Now what equation must be true if λ is an eigenvalue of AA^T?

2. There is a beautiful relationship between the column vectors of U and V. We will assume that A is $m \times n$ with rank k, and $A = U\Sigma V^T$ is the SVD of A. Then:

   $$A\boldsymbol{v}_i = \sigma_i \boldsymbol{u}_i \qquad \text{and} \qquad A^T \boldsymbol{u}_i = \sigma_i \boldsymbol{v}_i$$

   NOTE: The importance of this is that, if you know the left singular vectors, then you can compute the right singular vectors.

   HINT: You might start with $A\boldsymbol{v}_i$, then recall that $A = U\Sigma V^T$, which can be written as a sum of k matrices. The proof of the second equation is similar.

3. We said that $\sigma_i = \sqrt{\lambda_i}$. Here we prove that $\lambda_i \geq 0$, so that the singular values are always real:

   Show that $\lambda_i \geq 0$ for $i = 1..n$ by showing that $\|A\boldsymbol{v}_i\|^2 = \lambda_i$. HINT: As a starting point, you might rewrite

   $$\|A\boldsymbol{v}_i\|^2 = (A\boldsymbol{v})^T A\boldsymbol{v}$$

4. Prove that, if $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ are distinct eigenvectors of $A^T A$, then their corresponding images, $A\boldsymbol{v}_i$ and $A\boldsymbol{v}_j$, are orthogonal.

5. Prove that, if $\boldsymbol{x} = \alpha_1 \boldsymbol{v}_1 + \ldots \alpha_n \boldsymbol{v}_n$, then $\|A\boldsymbol{x}\|^2 = \alpha_1^2 \lambda_1 + \ldots + \alpha_n^2 \lambda_n$. (You might check the hint in #3).

6. Let W be the subspace generated by the basis $\{\boldsymbol{v}_j\}_{j=k+1}^n$, where $\boldsymbol{v}_j$ are the eigenvectors associated with the *zero* eigenvalues of $A^T A$ (therefore, we are assuming that the first k eigenvalues are NOT zero). Show that $W = \text{Null}(A)$.

   Hint: To show this, take an arbitrary vector $\boldsymbol{x}$ from W. Rather than showing directly that $A\boldsymbol{x} = \mathbf{0}$, instead show that the magnitude of $A\boldsymbol{x}$ is zero. We also need to show that if we take any vector from the nullspace of A, then it is also in W.

7. Prove that if the rank of $A^T A$ is r, then so is the rank of A.

   Hint: How does the previous exercise help?

8. Compute the SVD by hand of the following matrices:

   $$A_1 = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

   (Hint: $A^T A$ and AA^T are symmetric matrices.)

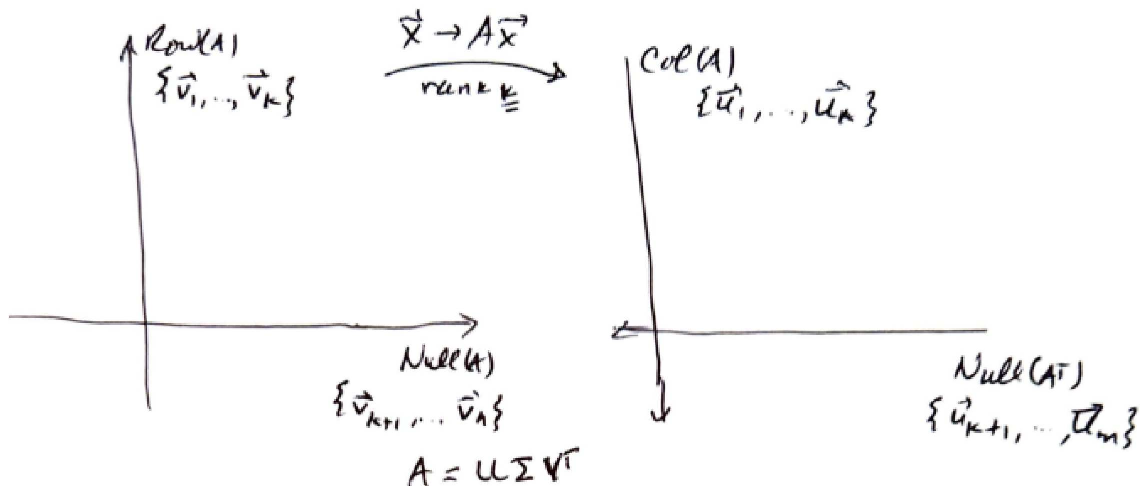


Figure 4.1: The SVD of A (`[U,S,V]=svd(A)`) completely and explicitly describes the 4 fundamental subspaces associated with the matrix, as shown. We have a one to one correspondence between the rowspace and columnspace of A, the remaining $\boldsymbol{v}$'s map to zero, and the remaining $\boldsymbol{u}$'s map to zero (under A^T).

4.6 Notes about the SVD

The SVD computes a basis for all 4 fundamental subspaces for matrix A. To be more specific, let $A = U\Sigma V^T$ be the SVD which has rank k. Be sure that the singular values are ordered from highest to lowest. Then:

1. A basis for the columnspace of A, Col(A) is $\{\boldsymbol{u}_i\}_{i=1}^k$
2. A basis for nullspace of A, Null(A) is $\{\boldsymbol{v}_i\}_{i=k+1}^n$
3. A basis for the rowspace of A, Row(A) is $\{\boldsymbol{v}_i\}_{i=1}^k$
4. A basis for the nullspace of A^T, Null(A^T) is $\{\boldsymbol{u}_i\}_{i=k+1}^m$

Those are the basis vectors, but does σ_i also have some geometric interpretation? Yes- Recall that $A\mathbf{v}_i = \sigma_i \boldsymbol{u}_i$. Therefore, we can think of each σ_i as either a stretching or contraction factor along each $\boldsymbol{v}_i$ (which turns into $\boldsymbol{u}_i$.

The SVD is one of the premier tools of linear algebra primarily because it allows us to completely reveal everything we need to know about a matrix mapping: The rank, the basis of the nullspace, a basis for the column space, the basis for the nullspace of A^T, and of the row space. See Figure 4.1.

Lastly remembering that matrix multiplication is function composition, and multiplication by an orthogonal matrix represents a "rotation", the SVD provides a means of decomposing **any** linear mapping into two "rotations" and a scaling. This will become important later when we try to deduce a mapping matrix from data.

4.6.1 The Reduced (or "economy-size") SVD

If you have a matrix A that is 50000×3, you should NOT ask for the complete SVD decomposition unless you really, really mean it.

Why? Think about the sizes of your matrices- The matrix U would be 50000×50000, which is very large, and most of which is not needed. Indeed, the rank of your matrix is at most 3, so there are only at most three columns of your matrix that are really needed (unless again, you specifically require the basis for

the nullspace of A^T). Instead, what you want to compute is called the Reduced SVD (or the economy-size SVD).

Definition: The Reduced SVD

Let A be $m \times n$ with rank k. Then we can write:

$$A = \widetilde{U}\widetilde{\Sigma}\widetilde{V}^T$$

where $\tilde{U}$ is an $m \times k$ matrix with orthogonal columns, $\tilde{\Sigma}$ is an $k \times k$ *square* matrix, and $\tilde{V}$ is an $n \times k$ matrix.

Most of the time we'll only need the reduced SVD- The only difference is that we're stripping away the two nullspaces from the set of four fundamental subspaces. We'll see an example comparing the full with the reduced SVD below.

Remarks on some language: The reader will see several ways of referring to the "non-full" SVD. For example, there is reduced SVD, truncated SVD, and thin SVD to name three. I typically refer to the reduced SVD by using the rank of the matrix k to define matrix sizes (as above). If $m > n$, and we use n to define the sizes, then the decomposition is referred to as the "thin SVD" (Golub and Van Loan). However, computer software rarely will compute the rank of the matrix unless requested, so they actually compute the thin SVD, but will refer to it as the reduced SVD. The moral of the story is to be sure you know which version it is that you're using!

4.7 Programming with the SVD

Here, we'll perform some computations with a small, 5×3 matrix to illustrate some of the formulas we've been working with. We'll make the matrix have rank 2, then we'll compare the full and reduced SVD formulas. For this example, we'll take

$$A = \begin{bmatrix} -1 & 2 & 4 \\ 1 & 0 & -2 \\ 0 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

Now, the full SVD would look like $U\Sigma V^T$, where U, Σ, and V are below (resp):

$$\begin{bmatrix} 0.78 & 0.35 & -0.38 & -0.35 & -0.06 \\ -0.28 & -0.45 & -0.82 & -0.19 & -0.03 \\ 0.25 & -0.05 & -0.23 & 0.63 & 0.69 \\ 0.46 & -0.60 & 0.11 & 0.38 & -0.52 \\ 0.21 & -0.55 & 0.34 & -0.54 & 0.49 \end{bmatrix} \begin{bmatrix} 5.68 & 0 & 0 \\ 0 & 3.42 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.07 & -0.57 & -0.82 \\ 0.63 & -0.66 & 0.41 \\ 0.77 & 0.49 & -0.41 \end{bmatrix}$$

The reduced SVD would strip away the nullspaces, leaving us with $\tilde{U}, \tilde{\Sigma}, \tilde{V}$ below (resp):

$$\begin{bmatrix} 0.78 & 0.35 \\ -0.28 & -0.45 \\ 0.25 & -0.05 \\ 0.46 & -0.60 \\ 0.21 & -0.55 \end{bmatrix} \begin{bmatrix} 5.68 & 0 \\ 0 & 3.42 \end{bmatrix} \begin{bmatrix} -0.07 & -0.57 \\ 0.63 & -0.66 \\ 0.77 & 0.49 \end{bmatrix}$$

Secondly, let's look at the decomposition we discussed a few pages ago. That was:

$$A = \sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^T + \sigma_2 \boldsymbol{u}_2 \boldsymbol{v}_2^T + \cdots + \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k^T$$

In our case, $k = 2$, and we'll compute this using our three computer languages as a check.

Matlab and the SVD

The full command is `[U,S,V]=svd(A)`, where $A = USV^T$. For the reduced SVD, add a zero in the list of arguments, `[U,S,V]=svd(A,0)`. Then here's the script using our example matrix A:

```
A=[-1 2 4;1 0 -2;0 1 1;1 3 1;1 2 0];
[U,S,V]=svd(A);
Temp=S(1,1)*U(:,1)*V(:,1)'+S(2,2)*U(:,2)*V(:,2)';
norm(A-Temp,'fro')
% We could compute "Temp" as:
Temp2=U(:,1:2)*S(1:2,1:2)*V(:,1:2)';
norm(A-Temp2,'fro')
```

Side note: The `'fro'` part of the norm is short for "Frobenius". The Frobenius norm of a matrix is like treating the whole matrix as one big vector, then applying the Euclidean norm. That is, take every element, square it, then sum those and take the square root of the result.

Python and the SVD

Quick note: After working with Python and matrices for a bit, I see that in Python 3.5 and later, the `@` symbol is used for matrix multiplication, so I'll use that below rather than `np.matmul`.

As a side note, if we want columns indexed as i to j from array A, use the notation `A[:,i:j+1]`. So the first two columns would be `A[:,0:2]`, where columns 0 and 1 are extracted.

```
import numpy as np

A=np.array([[-1,2,4],[1,0,-2],[0,1,1],[1,3,1],[1,2,0]])
U,S,VT=np.linalg.svd(A,full_matrices=0)
Temp=S[0]*U[:,:1] @ VT[:1,:] + S[1]*U[:,1:2] @ VT[1:2,:]
# We could compute Temp as:
Temp2=U[:,0:2] @ np.diag(S[:2]) @ VT[0:2,:]
```

R and the SVD

```
A<-cbind(c(-1,1,0,1,1),c(2,0,1,3,2),c(4,-2,1,1,0))

A.svd<-svd(A)  #Creates a structure holding info
U<-A.svd$u
D<-A.svd$d #Note that R uses UDV^T instead of USV^T
V<-A.svd$v
Temp<-D[1] * U[,1] %*% t(V[,1]) + D[2]* U[,2] %*% t(V[,2])
norm(A-Temp)
Temp2<-U[,1:2] %*% diag(D[1:2]) %*% t(V[,1:2])
norm(A-Temp2)
```

Next, we'll be looking at two applications- One in image processing, and one in computing a generalized inverse.

4.8 Generalized Inverses

Let a matrix A be $m \times n$ with rank k. In this general case, A does not have an inverse (A is not even square). Is there a way of restricting the domain and range of the mapping $\boldsymbol{y} = A\boldsymbol{x}$ so that the **restricted map** is invertible? And if so, how do we compute that inverse?

We'll find that the "inverse function" will be called the **Moore-Penrose Pseudo-Inverse**, and is relatively easy to compute using the SVD of matrix A. Indeed, let's consider the matrix equation:

$$A\boldsymbol{x} = \boldsymbol{b}$$

Replacing A by the reduced SVD (reduced to $m \times k, k \times k, n \times k$ matrices), we have:

$$U\Sigma V^T\boldsymbol{x} = \boldsymbol{b}$$

Since U is $m \times k$ with orthonormal columns, U^TU is the $k \times k$ identity, so we multiply both sides by U^T:

$$U^TU\Sigma V^T\boldsymbol{x} = U^T\boldsymbol{b} \quad \Rightarrow \quad \Sigma V^T = U^T\boldsymbol{b}$$

The matrix Σ is $k \times k$ with singular values along the diagonal. Because the rank is k, we have k non-zero singular values, so Σ is invertible. In the exercises, we'll show that the inverse is found by taking the reciprocal of each diagonal element- We'll see how that's computed in the programming section. Multiply both sides by the inverse:

$$\Sigma^{-1}\Sigma V^T\boldsymbol{x} = \Sigma^{-1}U^T\boldsymbol{b} \quad \Rightarrow \quad V^T\boldsymbol{c} = \Sigma^{-1}U^T\boldsymbol{x}$$

Finally, multiply both sides by V. Remember, $VV^T \neq I$, however, $VV^T\boldsymbol{x}$ is the projection of $\boldsymbol{x}$ into the rowspace of A. We'll call that $\tilde{\boldsymbol{x}}$.

$$\tilde{\boldsymbol{x}} = V\Sigma^{-1}U^T\boldsymbol{b}$$

This matrix product is the pseudo-inverse, denoted by dagger notation

$$A^\dagger = V\Sigma^{-1}U^T$$

You might recall that A is $m \times n$, and the pseudo-inverse $A^\dagger$ is $n \times m$. Just to be clear, this "inverse" will take a general $\boldsymbol{b} \in \mathbb{R}^m$ and map it back to an $\boldsymbol{x}$ in the rowspace of A. If you check, A will map $\boldsymbol{x}$ to the the projection of $\boldsymbol{b}$ to the columnspace of A (unless it is in the columnspace already). That's what we mean when we say that this mapping has been restricted to be between the $k-$dimensional rowspace and columnspace. **Restricted to these two subspaces**, the mapping $\boldsymbol{x} \to A\boldsymbol{x}$ is 1-1 and onto, and invertible!

Finding the rank of A

What we haven't discussed yet is the determination of the rank k. Theoretically, the value of k is the number of non-zero singular values (or non-zero eigenvalues of AA^T or A^TA).

The problem is one that is not theoretical, but computational. We need to determine if a number is zero, or if it is not- Seems simple, but we seldom see *exactly zero* in our computations- In the computations, what typically happens is that the singular values *approach* zero, and then we need to decide on a cut-off: how close to zero is zero?

The cut-off is typically arbitrary and problem-dependent. For example, if there is a clear drop between numbers (like $\sigma_2 = 8$ and $\sigma_3 = 0.01$), then we might go ahead and take the rank to be 2.

Often we look at the eigenvalues of AA^T or A^TA rather than the singular values- The reason is that later we'll see that the eigenvalues actually represent some statistical properties of the data (related to variance). In order to be consistent, rather than looking at the eigenvalues themselves, we'll look at the normalized eigenvalues:

$$\frac{\lambda_1}{\sum_i \lambda_i}, \frac{\lambda_2}{\sum_i \lambda_i}, \ldots, \frac{\lambda_n}{\sum_i \lambda_i}$$

These sum to 1, like probabilities. One idea is to take the rank k to be that scaled eigenvalue such that the sum of the first k values is at least, say, 0.99.

There are other ways as well. In fact, for large matrices, Gavish and Donoho[1] published some work on finding the optimal rank.

[1]Gavish and Donoho, "*The Optimal Hard Threshold for Singular Values is* $4/\sqrt{3}$", IEEE Transactions on Information Theory, v 60, Issue 8, 2014, pg 5040 - 5053

Programming Examples in Matlab, Python and R

The problem we'll be solving on the computer is to find the solution to the following matrix equation by explicitly computing the pseudo-inverse.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 4 \\ -1 \\ 2 \\ 1 \end{bmatrix}$$

In the code below, we will first construct the SVD of A, and then inspect the diagonal values of S to determine the rank (in this case, the rank is 2). We then compute the pseudo-inverse, `Ap`, and find the (least squares) solution to the equation above.

Finally, we verify our theory by comparing $A\mathbf{x}$ and the projection of $\boldsymbol{b}$ into the columnspace of A (they should be the same). Similarly, we compare our solution $\boldsymbol{x}$ with the projection of $\boldsymbol{x}$ into the rowspace (the projection of the projection does not change, so these should be the same vector).

We should note that each of these languages has the pseudo-inverse built in (probably `pinv`), but before we use it, we want to be sure we understand how it is constructed- The (possibly) non-trivial part of the construction is the determination of the rank of A.

- Matlab:

```
A=[1 2 3;4 5 6;7 8  9;10 11 12];
b=[4;-1;2;1];
[U,S,V]=svd(A);
S
Ap=V(:,1:2)*diag(1./diag(S(1:2,1:2)))*U(:,1:2)';  %Pseudo-inverse
xsoln=Ap*b                                        %The "solution" to our system.
A*xsoln                                           %This is what Ax actually is.
U(:,1:2)*U(:,1:2)'*b                              %See if Ax is the proj of b into
                                                  %   col(A).
V(:,1:2)*V(:,1:2)'*xsoln                          %See if x is the same as proj of x
                                                  %   into row(A).
```

- Python:

```
import numpy as np

A=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
b=np.array([[4],[-1],[2],[1]])
U,S,VT=np.linalg.svd(A,full_matrices=0)
print(S)      # In Python, S is not a diagonal matrix

Ap= VT[0:2,:].T @ np.diag(1/S[0:2]) @ U[:,0:2].T
xsoln=Ap @ b
print(A @ xsoln)                    #This vector and the next should be the same.
print(U[:,0:2] @ U[:,0:2].T @ b)
print(xsoln)                        #This vector and the next should be the same.
print(VT[0:2,:].T @ VT[0:2,:] @ xsoln)
```

- R:

```
A<-cbind(c(1,4,7,10),c(2,5,8,11),c(3,6,9,12))
```

```
b=cbind(c(4,-1,2,1))

A.svd<-svd(A)  #Creates a structure holding info
U<-A.svd$u
S<-A.svd$d #Note that R uses UDV^T instead of USV^T
V<-A.svd$v

Ap=V[,1:2] %*% diag(1/S[1:2]) %*% t(U[,1:2])
xsoln=Ap %*% b

A%*%xsoln                        #This vector and the next should be equal
U[,1:2] %*% t(U[,1:2]) %*% b
V[,1:2] %*% t(V[,1:2]) %*% xsoln  #This vector and the next should be equal
xsoln
```

4.9 Exercises

Before working through the exercises, be sure you've tried out the SVD and pseudo-inverse code examples so you have the template files ready for the homework.

1. In each of the programming languages, we built Σ^{-1} by replacing each diagonal element with its reciprocal. Because we had rank 2, we only used the first two elements.

   What happens if you forget that, and compute (in each language, respectively):

   ```
   Matlab:  U*diag(1./diag(S))*V'        Python:  VT.T @ np.diag(1/S) @ U.T
   In R:  V%*% diag(1/S)%*% t(U)
   ```

   Compare these to your previous pseudo-inverse. Notice anything? What happened?

2. Some sources say that, if A is full rank (let's assume that $m > n$ with rank n), then the pseudo-inverse can be computed as the following. We want to verify this using the SVD.

   $$A^{\dagger} = (A^T A)^{-1} A^T$$

   Hint: Start with $A = U\Sigma V^T$, and note that $\Sigma^{-1}\Sigma^{-1} = \Sigma^{-2}$, and $\Sigma^{-2}\Sigma = \Sigma^{-1}$, since Σ is a diagonal matrix.

3. Let A be $m \times n$ with rank k, so that $A^{\dagger} = U\Sigma^{-1}V^T$ is from the rank k SVD of A.

   (a) Show that the pseudo-inverse of the pseudo-inverse is the matrix A: $(A^{\dagger})^{\dagger} = A$

   (b) Simplify the expression $AA^{\dagger}$ using the SVD.

   (c) Simplify the expression $A^{\dagger}A$ using the SVD.

   (d) Given $A\boldsymbol{x} = \boldsymbol{b}$, solve for $\boldsymbol{x}$ by first multiplying both sides by $A^{\dagger}$, and use your previous simplification to simplify the equation.

4. Consider

   $$\begin{bmatrix} 2 & 1 & -1 \\ 3 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

   (a) Before solving this problem, what are the dimensions of the four fundamental subspaces?

   (b) Use Matlab, Python or R to compute the SVD of the matrix A, and solve the problem by computing the pseudoinverse of A explicitly.

(c) Check your answer explicitly and verify that $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ are in the rowspace and columnspace, similar to the example.

5. Consider

$$\begin{bmatrix} 2 & 1 & -1 & 3 \\ -1 & 0 & 1 & -2 \\ 7 & 2 & -5 & 12 \\ -3 & -2 & 0 & -4 \\ 4 & 1 & -3 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x4 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 0 \\ -2 \\ 6 \end{bmatrix}$$

(a) Find the dimensions of the four fundamental subspaces by using the SVD of A (in Matlab, Python or R).

(b) Solve the problem.

(c) Check your answer explicitly and verify that $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ are in the rowspace and columnspace.

Part I

Data Representations

Chapter 5

The Best Basis

5.1 Introduction

The problem we want to consider is this: We're given p points in $\mathbb{R}^n$. Find the "best" $k-$dimensional basis for the data.

There are a couple of things that will make our job easier:

- We will assume that the data has been mean-subtracted, so that the mean is zero (in $\mathbb{R}^n$).
- The basis is orthonormal (each basis vector is in $\mathbb{R}^n$).
- To find the "best" basis will require an error function. We will then minimize it.

At the end of this section, you'll see that the best $k-$dimensional basis for your data (regardless of k) is given by the first k **eigenvectors of the covariance matrix**, which are typically computed using the Singular Value Decomposition (SVD).

5.1.1 The Covariance Matrix, Revisited

Suppose we have p data points in $\mathbb{R}^n$, $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$, and they are organized column-wise in an $n \times p$ matrix X.

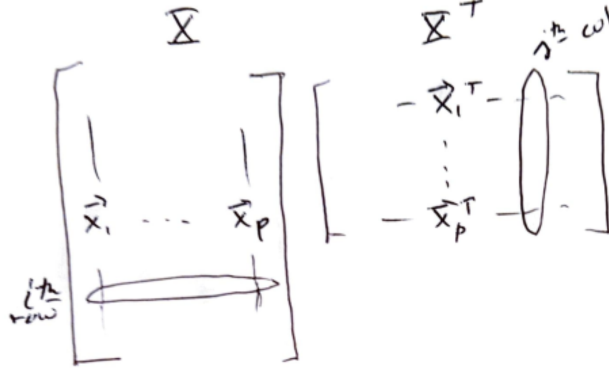
As we recall, the $n \times n$ covariance matrix for data in $\mathbb{R}^n$ measures the covariance between the data in coordinate i and the data in coordinate j. Using the $n \times p$ matrix X, then define $\bar{\boldsymbol{x}} \in \mathbb{R}^n$ as the mean, then the $(i,j)^{\text{th}}$ entry of the covariance matrix is given by the following, where we're taking the covariance between the ith and j^{th} row of X.

$$C_{ij} = \frac{1}{p-1} \sum_{k=1}^{p} (x_{ik} - \bar{x}_i)(x_{jk} - \bar{x}_j)$$

We will typically assume the mean is zero, so be sure and mean-subtract your data matrix before finding a basis for your data! With zero mean,

$$C_{ij} = \frac{1}{p-1} \sum_{k=1}^{p} x_{ik} x_{jk}$$

If we think of this computation in terms of the matrix X as in the figure below, we see that C_{ij} can be computed using a dot product between row i of X and column j of X^T:



Therefore, we see that the $n \times n$ matrix C can be computed one of two equivalent ways:

$$C = \frac{1}{p-1} XX^T \qquad \text{or} \qquad C = \frac{1}{p-1} \sum_{k=1}^{n} \boldsymbol{x}_k \boldsymbol{x}_k^T$$

Finally, we recognize that the covariance matrix is symmetric, so the **Spectral Theorem** applies. In particular, there is an orthonormal matrix P and a diagonal matrix D so that

$$C = PDP^T$$

where the columns of P form the eigenvectors associated with the diagonal elements of D (which are typically written largest to smallest).

To connect this to the SVD of the data matrix X, if X is $n \times p$ (so that data is stored column-wise), and we write the **reduced** SVD as:

$$X = U\Sigma V^T$$

Then

$$C = \frac{1}{p-1} XX^T = \frac{1}{p-1} U\Sigma V^T V \Sigma^T U^T = U \left(\frac{1}{p-1} \Sigma^2 \right) U^T$$

We see a relationship between the singular values of X, σ_i, and the eigenvalues of the covariance matrix, $\hat{\lambda}_i$:

$$\frac{1}{p-1} \sigma_i^2 = \hat{\lambda}_i$$

So far, we have defined a data matrix X, and we've looked at its covariance matrix C, and we've discovered that the eigenvectors of the covariance matrix are the left singular vectors of the data matrix (when the data is written column-wise and has been mean subtracted).

We'll be getting back to the best basis in a moment, but first we want to make a few more observations.

Projections and the Mean

Suppose you have your p data points in $\mathbb{R}^n$ that have *not* been mean subtracted, and you have a vector $\boldsymbol{u}$ onto which we want to project the data.

First, if we project one point $\boldsymbol{x}$ onto our (unit) vector $\boldsymbol{u}$, then the projection is $(\boldsymbol{x}^T\boldsymbol{u})\boldsymbol{u}$, and the scalar projection is the number $(\boldsymbol{x}^T\boldsymbol{u})$. Similarly, projecting all the data gives us p real numbers (the scalar projections):

$$\{\boldsymbol{u}^T\boldsymbol{x}_1, \boldsymbol{u}^T\boldsymbol{x}_2, \ldots, \boldsymbol{u}^T\boldsymbol{x}_p\}$$

so the mean of the projected data is given by

$$\frac{1}{p}\left(\boldsymbol{u}^T\boldsymbol{x}_1 + \boldsymbol{u}^T\boldsymbol{x}_2 + \ldots + \boldsymbol{u}^T\boldsymbol{x}_p\right) = \boldsymbol{u}^T \frac{(\boldsymbol{x}_1 + \boldsymbol{x}_2 + \ldots + \boldsymbol{x}_p)}{p} = \boldsymbol{u}^T\bar{\boldsymbol{x}}$$

Therefore, **the mean of the projection is the projection of the mean**. In particular, if the mean of a data set is zero, and the data is projected to a vector (or subspace), then the new mean is also zero.

Projections and the Variance

In the last section, we saw how the mean and the projection interacted. In this section, let's see how the variance is affected. We'll keep our previous general data set, p points in $\mathbb{R}^n$, and we'll suppose that **the data has zero mean** in $\mathbb{R}^n$. By what we showed, the scalar projections would also have zero mean.

If we project the p vectors onto an arbitrary unit vector $\boldsymbol{u}$, and consider the **scalar projections**, then the resulting variance (in the direction of $\boldsymbol{u}$ will be:

$$S_u^2 = \frac{1}{p-1}\sum_{k=1}^{p}(\boldsymbol{u}^T\boldsymbol{x}_k)^2 = \frac{1}{p-1}\sum_{k=1}^{p}\boldsymbol{u}^T\boldsymbol{x}_k\boldsymbol{x}_k^T\boldsymbol{u} = \boldsymbol{u}^T\left(\frac{1}{p-1}\sum_{k=1}^{p}\boldsymbol{x}_k{\boldsymbol{x}_k}^T\right)\boldsymbol{u} = \boldsymbol{u}^T C\boldsymbol{u}$$

This is actually a very key quantity, and will come up in the next section. We will look at this quantity more closely in a bit, but let's look at what happens in one **special case**: Suppose that $\boldsymbol{u}$ is an eigenvector of the covariance C corresponding to the first eigenvalue, $\hat{\lambda}_1$ using our previous notation. Then:

$$\boldsymbol{u}^T C\boldsymbol{u} = \boldsymbol{u}^T\hat{\lambda}_1\boldsymbol{u} = \hat{\lambda}_1$$

Therefore, if we project all the data to the first eigenvector of C, the new variance will be the first eigenvalue of C.

Reconstruction Error and the Basis

Given a specific vector $\boldsymbol{x}\in\mathbb{R}^n$ and an arbitrary orthonormal basis, $\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \ldots, \boldsymbol{\phi}_n$, we can write

$$\boldsymbol{x} = (\boldsymbol{\phi}_1^T\boldsymbol{x})\boldsymbol{\phi}_1 + (\boldsymbol{\phi}_2^T\boldsymbol{x})\boldsymbol{\phi}_2 + \ldots + (\boldsymbol{\phi}_n^T\boldsymbol{x})\boldsymbol{\phi}_n$$

so that the magnitude of $\boldsymbol{x}$ can be written as:

$$\|\boldsymbol{x}\|^2 = (\boldsymbol{\phi}_1^T\boldsymbol{x})^2 + (\boldsymbol{\phi}_2^T\boldsymbol{x})^2 + \ldots + (\boldsymbol{\phi}_n^T\boldsymbol{x})^2.$$

We can use the same algebraic manipulation that we used in the last section to rewrite this as:

$$\|\boldsymbol{x}\|^2 = \boldsymbol{\phi}_1^T\boldsymbol{x}\boldsymbol{x}^T\boldsymbol{\phi}_1 + \boldsymbol{\phi}_2^T\boldsymbol{x}\boldsymbol{x}^T\boldsymbol{\phi}_2 + \ldots + \boldsymbol{\phi}_n^T\boldsymbol{x}\boldsymbol{x}^T\boldsymbol{\phi}_n.$$

We can break this up and define the error using one vector $\boldsymbol{\phi}_1$:

$$\|\boldsymbol{x}\|^2 = \boldsymbol{\phi}_1^T\boldsymbol{x}\boldsymbol{x}^T\boldsymbol{\phi}_1 + \|\boldsymbol{x}_{\text{err}}\|^2$$

Now do this for all p data points. For any single vector $\boldsymbol{\phi}_1$, we sum these together:

$$\begin{aligned}
\|\boldsymbol{x}_1\|^2 &= \boldsymbol{\phi}_1^T\boldsymbol{x}_1\boldsymbol{x}_1^T\boldsymbol{\phi}_1 + \|\boldsymbol{x}_{\text{err}}^{(1)}\|^2\\
+\|\boldsymbol{x}_2\|^2 &= \boldsymbol{\phi}_1^T\boldsymbol{x}_2{\boldsymbol{x}_2}^T\boldsymbol{\phi}_1 + \|\boldsymbol{x}_{\text{err}}^{(2)}\|^2\\
&\vdots\\
\|\boldsymbol{x}_p\|^2 &= \boldsymbol{\phi}_1^T\boldsymbol{x}_p{\boldsymbol{x}_p}^T\boldsymbol{\phi}_1 + \|\boldsymbol{x}_{\text{err}}^{(p)}\|^2
\end{aligned}$$

$$\sum_{k=1}^{p}\|\boldsymbol{x}_k\|^2 = \boldsymbol{\phi}_1^T\left(\sum_{k=1}^{p}\boldsymbol{x}_k{\boldsymbol{x}_k}^T\right)\boldsymbol{\phi}_1 + \sum_{k=1}^{p}\|\boldsymbol{x}_{\text{err}}^{(k)}\|^2$$

We can multiply everything by $1/(p-1)$ to make things work. That is,

$$\frac{1}{p-1}\sum_{k=1}^{p}\|\boldsymbol{x}_k\|^2 = \boldsymbol{\phi}_1^T C\boldsymbol{\phi}_1 + \frac{1}{p-1}\sum_{k=1}^{p}\|\boldsymbol{x}_{\text{err}}^{(k)}\|^2. \tag{5.1}$$

In light of this equation, let us now define an error function using an arbitrary orthonormal basis, $\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_n$. The error we get when using a one dimensional representation of our data is given by

$$E(\boldsymbol{\phi}_2, \ldots, \boldsymbol{\phi}_n) = \frac{1}{p-1} \sum_{n=1}^{p} \|\boldsymbol{x}_{\text{err}}^{(n)}\|^2.$$

Notice that the left side of Equation 5.1 is constant (it is the sum of the magnitudes of all the known data). Therefore, **minimizing** the error function (the second term) is equivalent to **maximizing** the first term, $\boldsymbol{\phi}_1^T C \boldsymbol{\phi}_1$.

Here now is our algorithm for find the "best" basis:

1. Find the unit vector $\boldsymbol{\phi}_1$ so that $\boldsymbol{\phi}_1^T C \boldsymbol{\phi}_1$ is maximized.
2. We "project out" this vector so that the ith data point now becomes:

$$\underline{\boldsymbol{x}}^{(i)} = \boldsymbol{x}^{(i)} - \text{Proj}_{\boldsymbol{\phi}_1}(\boldsymbol{x}^{(i)})$$

3. Re-compute C.
4. Repeat from Step 1 until we have enough basis vectors.

In practice, we will not need to do this- there is an easier way!

5.2 The Best Basis and the Eigenvectors

We have shown that finding the best basis reduces to maximizing the quantity:

$$\max_{\boldsymbol{\phi} \neq \mathbf{0}} \frac{\boldsymbol{\phi}^T C \boldsymbol{\phi}}{\boldsymbol{\phi}^T \boldsymbol{\phi}}$$

where we divide by the magnitude (squared) to enforce the fact that we want a unit vector, and we want to stay away from the zero vector.

We know that the eigenvectors of the covariance matrix form an orthonormal basis for $\mathbb{R}^n$, so we can write any vector as a linear combination of them:

$$\boldsymbol{\phi} = c_1 \boldsymbol{v}_1 + c_2 \boldsymbol{v}_2 + \cdots + c_n \boldsymbol{v}_n = V\boldsymbol{c}$$

Using the eigenvector-eigenvalue factorization $C = VDV^T$, we can now write the numerator as:

$$\boldsymbol{\phi}^T C \boldsymbol{\phi} = (V\boldsymbol{c})^T (VDV^T)(V\boldsymbol{c}) = \boldsymbol{c}^T (V^T V) D (V^T V) \boldsymbol{c} = \boldsymbol{c}^T D \boldsymbol{c} = c_1^2 \lambda_1 + c_2^2 \lambda_2 + \cdots + c_n^2 \lambda_n$$

Similarly, the denominator is:

$$\boldsymbol{\phi}^T \boldsymbol{\phi} = c_1^2 + c_2^2 + \cdots + c_n^2$$

Let's look at the coefficients of our expansion now. For λ_i, the coefficient in front is

$$\rho_i = \frac{c_i^2}{c_1^2 + c_2^2 + \cdots + c_n^2}$$

where $\rho_i \geq 0$ and $\sum_{i=1}^n \rho_i = 1$ (like a probability distribution). Let's summarize where we are. We now see that maximizing $\boldsymbol{\phi}^T C \boldsymbol{\phi}$ is equivalent to choosing $\rho_1, \rho_2, \cdots, \rho_n$ so that each $\rho_i \geq 0$ and they sum to 1, to maximize the quantity:

$$\rho_1 \lambda_1 + \rho_2 \lambda_2 + \cdots + \rho_n \lambda_n$$

It is easy to see that, if the $\lambda_i \geq 0$ and are ordered from largest to smallest, then:

$$\lambda_n \leq \rho_1\lambda_1 + \rho_2\lambda_2 + \cdots + \rho_n\lambda_n \leq \lambda_1.$$

To maximize our given quantity, we set $c_1 = 1$ and the rest of the coefficients to zero. This leads us to our main conclusion. The vector $\boldsymbol{\phi}$ that maximizes the quantity $\max_{\boldsymbol{\phi}\neq\mathbf{0}} \dfrac{\boldsymbol{\phi}^T C\boldsymbol{\phi}}{\boldsymbol{\phi}^T\boldsymbol{\phi}}$ is given by $\boldsymbol{v}_1$, the eigenvector corresponding to the largest eigenvalue of C. This is summarized by the theorem below:

The Best Basis Theorem

Given p points in $\mathbb{R}^n$, the *best $k-$dimensional basis* is found by taking the first k eigenvectors of the covariance matrix C. Equivalently, given the data in an $n \times p$ matrix X, the best $k-$dimensional basis is found by taking the first k columns of the U, the left singular vectors of the SVD of X. Further, this is the "best" basis for $k = 1, 2, \ldots, r$, where r is the rank of X.

Speaking of Rank...

We discussed this briefly in an earlier section, but it is worth thinking about again.

It is useful to look at the rank as that number of basis vectors required to preserve some percentage of the variance in the data. From our previous section on the covariance matrix, we had a relationship between the eigenvalues of the covariance matrix, $\hat{\lambda}_i$ and the singular values of X:

$$\frac{1}{p-1}\sigma_i^2 = \hat{\lambda}_i$$

so normalizing the set of eigenvalues is equivalent to doing it to the squared singular values:

$$\lambda_i = \frac{\hat{\lambda}_i}{\sum_{j=1}^n \hat{\lambda}_j} = \frac{\frac{1}{p-1}\sigma_i^2}{\sum_{j=1}^n \frac{1}{p-1}\sigma_j^2} = \frac{\sigma_i^2}{\sum_{j=1}^n \sigma_j^2}$$

Now the λ_i are positive and sum to 1. The idea is to keep enough dimensions r so that

$$\sum_{i=1}^{r} \lambda_i \geq \tau \qquad \text{but} \qquad \sum_{i=1}^{r-1} \lambda_i < \tau.$$

In this case, we would say that it takes r dimensions to explain or encapsulate τ percent of the variance in the data.

What should τ be? This is problem dependent. In some very noisy problems, you may only want to keep $\tau \approx 0.6$, while with very little noise, you might take $\tau \approx 0.99$.

5.2.1 The Dimensionality Reduction Step

Once we have our k basis vectors, what do we do with them? First, we create our low dimensional representation of the data. Initially, the data represents p points in $\mathbb{R}^n$, and we want to reduce that to p points in $\mathbb{R}^k$- These are the coordinates of each point using our $k-$dimensional basis. That is, if $U\Sigma V^T$ is the svd of X (mean subtracted), then the k dimensional data is created by the following, where U is $n \times k$, X is $n \times p$, and the low-dimensional representation X_{coords} is $k \times p$.

$$X_{\text{coords}} = U^T X$$

Especially if $k = 2$ or $k = 3$, we can then plot the low dimensional points in the plane or in 3-d. The "reconstruction" of the data is the representation back in $\mathbb{R}^n$ using the k basis vectors.

$$X_{\text{recon}} = UX_{\text{coords}} = UU^T X$$

Remember that earlier we said that $U^T U = I$, but UU^T is the projection matrix taking data in $\mathbb{R}^n$ and projecting it into the column space of U (so $X \neq UU^T X$ unless the columns of X are already contained within the column space of U).

5.3 Connecting to Principal Components Analysis (PCA)

In PCA, the principal components are defined to be a sequence of k direction vectors, where the ith vector is the direction of a line that best fits the data while being orthogonal to the first $i-1$ vectors[1].

You can see that the principal components of set of data are then equivalent to the k basis vectors we've constructed (the first k eigenvectors of the covariance matrix). While PCA and the best basis are the same, you will typically hear the language of statistics used in PCA, while we use the language of linear algebra in constructing the best basis.

5.4 Exercises

Before doing the computer problems below, you should write down (using linear algebra notation) what computations you want to make. If you have questions (especially with the coding), I'm happy to help.

1. Suppose we have p data points in $\mathbb{R}^n$. Show that the variance of the data, projected to a standard basis vector $\boldsymbol{e}_i$, returns the usual variance for the data in that dimension. (I want you to look back at the computations we made for this in the text, "Projections and the Variance").

2. Suppose we have two o.n. vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$. Given our p points in $\mathbb{R}^n$, compute the covariance between the data projected to $\boldsymbol{u}$ and the data projected to $\boldsymbol{v}$, and (i) show that the result is

$$\boldsymbol{u}^T C \boldsymbol{v}$$

   (ii) In the special case that $\boldsymbol{u}, \boldsymbol{v}$ are eigenvectors of the covariance matrix, how does this quantity simplify?

3. Suppose we have 4 points in $\mathbb{R}^3$ as organized in the matrix X (left and below), and let $\boldsymbol{\phi}_1 = (1/\sqrt{3})[1, 1, 1]^T$. Use a computer (Octave/Matlab, Python or R) to compute the three quantities given in the formula to the right and below. In your script, be sure you're actually computing the covariance matrix and each quantity separately.

$$X = \begin{bmatrix} 1 & 2 & -1 & 3 \\ 0 & 0 & 1 & 1 \\ -1 & 1 & 2 & 1 \end{bmatrix}, \qquad \frac{1}{p-1}\sum_{k=1}^{p} \|\boldsymbol{x}_k\|^2 = \boldsymbol{\phi}_1^T C \boldsymbol{\phi}_1 + \frac{1}{p-1}\sum_{k=1}^{p} \|\boldsymbol{x}_{\text{err}}^{(k)}\|^2.$$

4. Using the data (and vector $\boldsymbol{\phi}_1$) in the previous exercise, compuationally verify our statements: The projection of the mean is the mean of the projection, and the variance of the data projected to $\boldsymbol{\phi}_1$ is $\boldsymbol{\phi}_1^T C \boldsymbol{\phi}_1$.

5. Verify numerically that the variance of the projected data to the first best basis vector (first one) is given by the first eigenvalue of the covariance matrix. (Careful- if you use the `eig` command, the eigenvalues are not ordered).

6. Continuing with the data from Problem 3, if we retained two of the basis vectors, how much variance (as a percentage) is "explained" by them? (This refers to the discussion in the text about how to compute the rank).

7. Load the clown data, we obtain a matrix X that is 200×320. Treat this as 320 vectors in $\mathbb{R}^{200}$.
   (a) Double center the data in X (call the result X_m).
   (b) Find the best two dimensional basis for the vectors in X_m, then project the data to two dimensions and plot the result.
   (Question to think about, you don't need to answer: Did you expect a pattern or not?)
   (c) Reconstruct the data back in $\mathbb{R}^{200}$, and show the result as an image (don't add the means back in).

[1]Wikipedia, pulled March 2021

5.5 A Summary

Before we continue, let's summarize the process for PCA (or best basis), and let's discuss what you can get out of the PCA. First, the process- It's short.

Principal Components Analysis

Let X store p points, each in $\mathbb{R}^n$ so X is $n \times p$. To find the best basis (in $\mathbb{R}^n$):

1. Compute the mean so that $\bar{\boldsymbol{x}} \in \mathbb{R}^n$.
2. Mean subtract the data, put in the $n \times p$ matrix X_m.
3. For the resulting data, take the SVD:
$$X_m = U\Sigma V^T$$
4. The best basis (or principal components) are the first k columns of U.

Once this is done, what can we do with it? Here is a list of common tasks.

- Visualize the mean and eigenvectors.

  The eigenvectors of the covariance matrix (the columns of U) typically carry very interesting information- What do they look like? In the application below, the eigenvectors can be visualized as photos, but you can also simply plot an eigenvector if that has meaning (plot the vector index along the horizontal axis, and the vector values along the vertical axis. In Matlab, if $\boldsymbol{v}$ is a vector, this would be `plot(v)`.

- Determine the rank.

  We talked about this earlier- The rank of a matrix is critical to know if you'll be constructing the pseudo-inverse, or if you simply want to reduce the dimensionality of the data (see the next item). One way to determine the rank is to look for large breaks in the plot of the singular values (the diagonal elements of Σ). A second way is to construct the eigenvalues of the covariance matrix, and retain enough dimensions so that the resulting space encapsulates a certain percentage of the variance.

- Reduce the dimensionality of the data.

  This is the primary goal of most applications of PCA. If X_m is $n \times p$ (p points in $\mathbb{R}^n$, with mean subtracted), then the following is the $k \times p$, or $k-$dimensional, representation of the p points- matrix L (for low dimensional) is $k \times p$:
$$L = U(:, 1:k)^T X_m$$
  The Matlab-esque notation means to take all rows, but only the first k columns of U.

- Visualize the data that has been reduced in dimension.

  It's often a good idea to plot the first rows of L in $\mathbb{R}^2$ just to be sure it gives you what you expect. There might be even more interesting information using other basis vectors down further in the decomposition (corresponding to lower rows of L)- We'll actually see an example below.

- Project new data using the new basis.

  If D is $m \times n$ (m points in $\mathbb{R}^n$), and D_m is the matrix with the mean of X subtracted from the columns, then the projected data in matrix P is also $m \times n$ and is given by:
$$P = U(:, 1:k)U(:, 1:k)^T D_m$$
  You can then compare P to D_m (or add the mean back to both matrices) to see if the new data is well represented by the basis.

Before we leave this discussion for an application, have you considered what the matrix V (from the SVD of X) represents? It represents a scaled version of the coordinates (or low dimensional representation). How? If $X_m = U\Sigma V^T$ is the reduced SVD, then

$$L = U^T X_m = U^T(U\Sigma V^T) = \Sigma V^T$$

And remember that Σ is just a diagonal matrix ($k \times k$) and V is $p \times k$. Multiplying by the diagonal just scales each column of V, so V is a scaled version of the $k-$dimensional representation of the data!

5.6 Application: Eigenfaces

Consider the set of all photos of some fixed width and length (say $a \times b$) and these are photos of faces. To be even more specific, let's set the coordinates of the eyes to be the same in all photos.

As you might imagine, the dimension of this "face space" should be quite large, since we're representing all possible faces here. However, should it fill up $\mathbb{R}^{ab}$? If you were to take an arbitary vector in $\mathbb{R}^{ab}$ and visualize it as an $a \times b$ photo, what would you see? The image would probably be just noise- the set of "faces" is then (hopefully) only a "small" dimensional subspace of $\mathbb{R}^{ab}$, and that gives us some reason to expect that we can find a smallish number of template faces so that every face is a linear combination of the templates.

What are these template faces? They are basis vectors in $\mathbb{R}^{ab}$, meaning that each basis vector can be visualized as an $a \times b$ image. We know that these basis vectors are the eigenvectors of the covariance matrix of the photo data, so we'll call them **eigenfaces**.

In out next example, we'll explore face space a bit and see some interesting things.

5.6.1 Project Example: The Yale Database

The data we'll look at is a portion of the Extended Yale Database below linked below if you want to have a look (good as of March 2021):

Extended Yale Database

Further, this little project is adapted from Brunton and Kutz' text on "Data Driven Science and Engineering", which is at `databookuw.com`. The had a very cute example!

As a side note, it's important to distinguish between an $m \times n$ matrix (which as m rows and n columns) versus an $m \times n$ image, which has a width of m pixels and a height of n pixels (not my fault!). The images below are said to be 168×192, but are typically stored in a 192×168 matrix (for future reference, $168 \times 192 = 32256$).

Discussion of the Data

We'll begin with a discussion of the dataset `allFaces2.mat`, which will be available from our class website. Here is a discussion of the datasets you'll see stored in this file.

- `faces` is 32256×2410 matrix, representing 2410 photos, each of which is 192×168 (as a matrix).
- Constants $n = 192$ and $m = 168$.
- `nfaces` is a vector with 38 elements, each representing one of the persons posing for photos. The datain `faces` is stored in order with all the poses for the first person first, then the second person, and so on. The vector `nfaces` stores the number of poses each person has. For example, the first person has 64 poses, so `nfaces(1)` is 64.
- `Dog` is an image of the same size as the photos of the people, but is the grayscale photo of a dog.

Figure 5.1: The first 36 of the 38 persons in the file are shown to the left. To the right, we show a sample of 36 poses for the first person (out of 64 included in the data file). Each photo is 168×192 (as an image).

Visualizing a Sample of Data

In Figure 5.1, the first 36 of the 38 persons in the file are shown to the left. To the right, we show a sample of 36 poses for the first person (out of 64 included in the data file).

Project Outline

We're going to find the best basis for our face data. Since we have so many photos, we're going to try "training" the PCA on just the photos from the first 36 people, leaving the last two people out to see how well we have captured "face space". Continuing, here's what we'll do on the computer:

1. Load the data. The "training" data will be the first 2282 columns.
2. Find the mean, then mean-subtract the data. Visualize the mean.
3. Find the SVD.
4. Look at the singular values to see if there is any clear break.
5. Look at some sample eigenfaces (columns of U).
6. Project persons 2 and 7 into the plane spanned by the 5th and 6th basis vector, and plot the result.
7. Look at reconstructions of person 37 using dimension: $50, 150, 500, 1000$.
8. Look at whether or not we can reconstruct the dog's face using a basis from people!

Project Implementation: Matlab/Octave

Side Remark: This data will most likely be too large to run on Octave-online. However, if you have Octave set up on your home PC, it should work. For the homework, we'll be using a smaller data set.

```
%% Eigenface Example using Yale Database

%%Step 1: Load the data and create the training subset using 36 people.
%   The total number of columns needed is: sum(nfaces(1:36))

load allFaces2.mat
trainingFaces=faces(:,1:sum(nfaces(1:36)));

%% Step 2: Find the mean, mean-subtract the data.  Visualize the mean
%    as a photo!

avgFace=mean(trainingFaces,2);
Xm=trainingFaces-avgFace;

figure(1)
imagesc(reshape(avgFace,n,m));
colormap(gray); axis off; axis square

%% Step 3: Compute the SVD.
[U,S,V]=svd(Xm,'econ');

%% Step 4: Look at the singular values; usually a log plot for high dimensions.
plot(log(diag(S)));  % At approx 2262, we see a big drop- this is the rank.

%% Step 5:  Let's look at some eigenfaces!  Here are the first four:
figure(2)

for j=1:4
    subplot(2,2,j)
    imagesc(reshape(U(:,j),n,m));
    colormap(gray); axis off; axis square;
end

%% Step 6:  Project the poses from persons 2 and 7 into the plane spanned by
%          basis vectors 5 and 6.  Don't worry about the formulas used to determine
%          the columns.  They're included here in case you want to change 2 and 7
%          to other people.
P1=2;  P2=7;  Bas1=5; Bas2=6;
P1cols=sum(nfaces(1:P1-1))+1:sum(nfaces(1:P1));
P2cols=sum(nfaces(1:P2-1))+1:sum(nfaces(1:P2));
Data=faces(:,[P1cols,P2cols]) - avgFace;

Coords=U(:,[Bas1,Bas2])'*Data;  % This is two dimensional data
figure(3)
plot(Coords(1,1:nfaces(P1)),Coords(2,1:nfaces(P1)),'ro');
hold on
plot(Coords(1,nfaces(P1)+1:end),Coords(2,nfaces(P1)+1:end),'k^');
hold off

%% Step 6A: Optional subplot with the two faces and positive/negative
%   5th basis vector.
```

```
subplot(2,2,1)
imagesc(reshape(faces(:,65),n,m));
colormap(gray); axis off; axis square;
subplot(2,2,2)
imagesc(reshape(U(:,5),n,m));
colormap(gray); axis off; axis square;
subplot(2,2,3)
imagesc(reshape(faces(:,381),n,m));
colormap(gray); axis off; axis square;
subplot(2,2,4)
imagesc(reshape(-U(:,5),n,m));
colormap(gray); axis off; axis square;

%% Step 7: Look at partial reconstructions of person 37 using rank 50, 150, 500, 1000
Data=faces(:,sum(nfaces(1:36))+1);
Data=Data-avgFace;
kdim=[50,150,500,1000];

figure(4)
for j=1:4
    subplot(2,2,j)
    Recon=U(:,1:kdim(j))*(U(:,1:kdim(j))'*Data);
    imagesc(reshape(Recon+avgFace,n,m));
    colormap(gray); axis off; axis square
end

%% Step 8:  Repeat, for the photo of a dog!
Data=Dog;
Data=Data-avgFace;
kdim=[50,100,400,600,1600];

figure(5)
subplot(2,3,1)
imagesc(reshape(Dog,n,m));
colormap(gray); axis off; axis equal

for j=1:5
    subplot(2,3,j+1)
    Recon=U(:,1:kdim(j))*(U(:,1:kdim(j))'*Data);
    imagesc(reshape(Recon+avgFace,n,m));
    colormap(gray); axis off; axis equal
end
```

Project Implementation: Python

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.io

# Step 0: set some parameters, load the original Matlab data
plt.rcParams['figure.figsize'] = [8, 8]
plt.rcParams.update({'font.size': 18})
```

```
mat_contents = scipy.io.loadmat('allFaces2.mat')
faces = mat_contents['faces']
m = int(mat_contents['m'])
n = int(mat_contents['n'])
Dog = mat_contents['Dog']
nfaces = np.ndarray.flatten(mat_contents['nfaces'])

#%%
# Step 1: Load the data
trainingFaces = faces[:,:np.sum(nfaces[:36])]

#%%
# Step 2: Find the mean, visualize it.
avgFace = np.mean(trainingFaces,axis=1) # size n*m by 1
Xm=trainingFaces-avgFace[:,np.newaxis];

plt.figure()
plt.imshow(np.reshape(avgFace,(m,n)).T)
plt.set_cmap('gray')
plt.title('Mean Face')
plt.axis('off')
plt.show()

#%%
# Step 3: Compute the SVD
U, S, VT = np.linalg.svd(Xm,full_matrices=0)

#%%
# Step 4: Plot the singular values.
plt.figure()
plt.plot(np.log(S))

#%%
# Step 5:  Let's look at some eigenfaces (first 4, pos and neg)

fig,axs = plt.subplots(2,2)
axs=axs.ravel()
for i in range(4):
    axs[i].imshow(np.reshape(U[:,i],(m,n)).T)

fig,axs = plt.subplots(2,2)
axs=axs.ravel()
for i in range(4):
    axs[i].imshow(np.reshape(-U[:,i],(m,n)).T)
    axs[i].set_title(str(i))


#%%
## Step 6:  Project person 2 and 7 onto basis vecs 5 and 6

P1 = 2 # Person number 2
```

```
P2 = 7 # Person number 7

P1data = faces[:,np.sum(nfaces[:(P1-1)]):np.sum(nfaces[:P1])]
P2data = faces[:,np.sum(nfaces[:(P2-1)]):np.sum(nfaces[:P2])]

P1data = P1data - avgFace[:,np.newaxis]
P2data = P2data - avgFace[:,np.newaxis]

# Project onto PCA modes 5 and 6
PCACoordsP1 = U[:,4:6].T @ P1data
PCACoordsP2 = U[:,4:6].T @ P2data

plt.figure()

plt.plot(PCACoordsP1[0,:],PCACoordsP1[1,:],'d',color='k',label='Person 2')
plt.plot(PCACoordsP2[0,:],PCACoordsP2[1,:],'^',color='r',label='Person 7')

plt.legend()
plt.show()


# %%
## Step 7:  Show eigenface reconstruction of image that was omitted from test set


plt.figure()
testFace = faces[:,np.sum(nfaces[:36])] # First face of person 37
plt.imshow(np.reshape(testFace,(m,n)).T)
plt.set_cmap('gray')
plt.title('Original Image')
plt.axis('off')
plt.show()


#%% Step 8: Reconstruct a dog from photos of people!
#
# Vector "Dog" needs to be reshaped for some reason...
Dog=np.reshape(Dog,(32256,))
testFace = Dog - avgFace
k_list = [50, 200, 500, 1000, 1600]

plt.figure()

fig,axs = plt.subplots(2,3)
axs=axs.ravel()

axs[0].imshow(np.reshape(Dog,(m,n)).T)
axs[0].axis('off')
for i in range(5):
    reconFace = avgFace + U[:,:k_list[i]]  @ ( U[:,:k_list[i]].T @ testFace)
    axs[i+1].imshow(np.reshape(reconFace,(m,n)).T)
    axs[i+1].axis('off')
```

Project: Eigenfaces

Description of the data: `Math350Homework.mat`

- Matrix X that is 24138 26 (26 photos, each with 162 rows and 149 columns)
- Constants $m = 162$ and $n = 149$.

If you want to visualize the photos, you would reshape the corresponding column vector. For example, to visualize the first four faces and put them together in one figure, we would type:

```
for jj=1:4
   subplot(2,2,jj)
   imagesc(reshape(X(:,jj),m,n));
   axis off; axis equal; colormap(gray)
   end
```

Our project is to use the provided code and previous example to do some analysis of this data set.

1. Compute the mean vector and represent it as a face (by plotting it).
2. Compute the first four eigenfaces (and represent them as faces). Put them together in one plot. In a second plot, show what the "photographic negatives" of the eigenfaces look like.
3. What might be a good approximation to the rank of the matrix X (if our goal is to make the faces recognizable, about 74% of the variance).
4. Plot the reconstruction of a randomly selected face using rank 2, 5, 10 and 15. Plot these (as photos) together in one figure.
5. Plot the data using the best two dimensional representation using a new figure. Plot the first 13 data points as asterisks and the remaining 13 as triangles.

Chapter 6

A Best Nonorthogonal Basis

In this section, we examine a particular question whose solution will involve getting an optimal *nonorthogonal* basis, which is quite contrary to our earlier chapters- in fact, the reader should ask why we would ever want to use a non-orthogonal basis when it would be quite easy (using Gram-Schmidt) to construct an orthogonal version of the same basis.

To answer this question, consider the synthetic data example in Figure 6.1. Here there is a definite "natural" basis appearing in the data- and the basis vectors are not orthogonal. While the data was synthetic, we do get similar types of data appearing in the problem of *Blind Signal Separation.* Consider the following tasks:

1. We have a patient that is pregnant. Our overall goal is to listen to the fetus heartbeat, but when we try, the sound of the mother's heartbeat is mixed with the heartbeat of the fetus. Symmetrically, if we were to try to listen to the heartbeat of the mother, we would also hear the heartbeat of the fetus. Is it possible to gather these sounds on microphones and manipulate the data so that the mother's (or fetus) heartbeat has been isolated?

2. We have two microphones placed at random, but distinct, places in a room. We also have two people speaking in the room (the placement of the people is distinct from the placement of the microphones- we do not assume that each microphone is placed in front of each speaker). Is it possible to manipulate the two mixtures of voices so that we can isolate each speaker's voice?

The answer lies in a fairly new technique called *Independent Component Analysis* (ICA) (versus what we studied earlier, *principal components analysis*, or PCA). In ICA, we assume that we have some underlying, statistically independent, processes and that we are observing mixtures of these processes. Our goal is to separate the mixtures.

This problem is also known as *Blind Signal Separation*, where we assume some unknown mixture of signals, and we attempt at separating them.

This process (the problem and the solution) can also be framed in other terms- we will focus on the geometric meaning of the problem, and will solve it using the techniques of linear algebra.

6.1 Set up the Signal Separation Problem

We will assume that there exists a "clean" separation of our observed mixture of two signals (we will be explicit in what we mean by that momentarily, and we will discuss the more general case in a moment). These signals, as time series, are two columns of a matrix S, so that $S \in \mathbb{R}^{p\times 2}$, where p is the length of the sample.

We will further assume that the mixtures we are observing are *linear* mixtures, so that the mixtures we observe may be modeled as:

$$\boldsymbol{x}_1 = \alpha_1 \boldsymbol{s}_1 + \alpha_2 \boldsymbol{s}_2$$

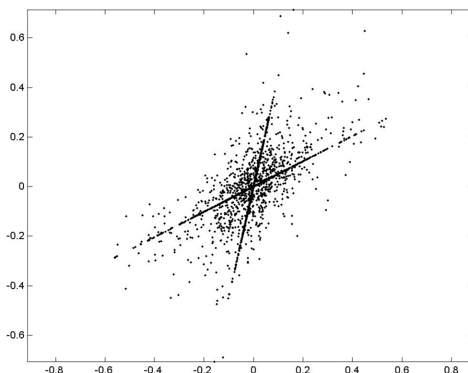


Figure 6.1: A synthetic data example of a naturally emerging set of basis vectors from data- These are not orthogonal.

$$\boldsymbol{x}_2 = \alpha_3 \boldsymbol{s}_1 + \alpha_4 \boldsymbol{s}_2$$

so that $\boldsymbol{x}_1$ is the observed mixture in microphone 1, and $\boldsymbol{x}_2$ is the observed mixture in microphone 2. In linear algebra terms, we can state the problem as follows:

Given $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ as columns of $X \in \mathbb{R}^{p\times 2}$, solve the following equation for $A \in \mathbb{R}^{2\times 2}$ and $S \in \mathbb{R}^{p\times 2}$:

$$X = SA$$

We assume that the rows of X have been mean-subtracted.

If you look at this equation, something should be occurring to you- this is not a well defined problem! There are an infinite number of solutions for A, S. In fact, one solution would be to let A be the 2×2 identity matrix, and $S = X$.

We could also give a solution in terms of the SVD of X, which is what we would do in Principal Component Analysis:

$$X = U_x \Sigma_x V_x^T$$

so that $S = U_x$, and $A = \Sigma_x V_x^T$. In this case the data is "separated" in the sense that the columns of U_x are orthogonal (or *uncorrelated*). Figure 6.2 shows the result of this operation on our synthetic data. It also shows that the desired basis vectors are still rotated.

This process, while not the desired one, is a good first step, but somehow we need the signals to be *independent*, and not just uncorrelated.

Alternatively, let us consider the SVD of the unknown mixing matrix A, $A = U_m \Sigma_m V_m^T$. The problem will be solved if we knew this matrix, as S could be computed by using the inverse of A (we assume that A is full rank- see the exercises for a discussion). Let's try some sample computations now by taking our original equation and substituting the SVD of A:

$$X = SA \Rightarrow X = SU_m \Sigma_m V_m^T$$

Now compute the 2×2 covariance of X:

$$X^T X = V_m \Sigma_m U_m^T S^T S U_m \Sigma_m V_m^T$$

Now, if the rows of S are statistically independent, then they certainly should be uncorrelated. We will assume therefore that SS^T is some scalar multiple of the identity:

$$SS^T = cI_{2\times 2}$$

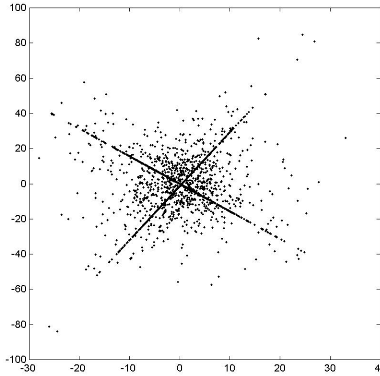


Figure 6.2: The synthetic data set after the SVD transformation as described in the text. In this case, the principal components analysis has left the desired basis vectors in a rotated position. We require one more rotation (multiplication by an orthogonal matrix) to get the desirable results.

which also assumes that the variances of the signals are the same. In particular, we'll assume that the signals in S have been scaled so that $S^T S = I$. In the exercises, we will examine this assumption in more detail.

Using this, we see that:

$$X^T X = V_m \Sigma_m^2 V_m^T = V_x \Sigma_x V_x^T$$

This tells us that V_m and Σ_m are recoverable from the SVD of X: If

$$X = U_x \Sigma_x V_x^T$$

then

$$\Sigma_m = \Sigma_x^{1/2}, \qquad V_m = V_x$$

If we were to stop here and take:

$$Y = X V_x \Sigma_x^{-1/2} = S U_m \Sigma_m V_m^T V_x \Sigma_x^{-1/2} = S U_m$$

we obtain the standard PCA solution. However, the signals are still rotated. We cannot perform another covariance computation on Y, since now:

$$Y^T Y = U_m^T S^T S U_m = I_{2\times 2}$$

How can we compute U_m? There is some justification for what we're about to do- Let's do it first and then discuss it.

Define dA to be the difference matrix for the matrix A. That is, if A has been organized so that it comprises p samples of k time series of data, then let A be $p \times k$. We compute the difference as:

$$dA = A(2:p,:) - A(1:p-1,:)$$

so that dA is now $p-1 \times k$ and

$$(dA)_{ij} = A_{i+1,j} - A_{i,j}$$

If the data in A were the sample of some differentiable function, then dA is an approximation to the derivative using $\Delta t = 1$.

Note that the following matrix equation holds:

$$X = SA \quad \Rightarrow \quad dX = dS\, A$$

so that

$$dX^T dX = A^T\, dS^T dS\, A$$

More particularly, let

$$Y = XV_x\Sigma_x^{-1/2}, \text{ or } dY = dXV_x\Sigma_x^{-1/2} = dS\, U_m$$

so that $dY^T dY = U_m^T dS^T dS\, U_m$. We now make our second assumption: While $S^T S = I$, we assume that $dS^T dS \neq I$, but is diagonal. In this case, we can recover U_m from the SVD of dY:

$$dY = U_y \Sigma_y V_y^T$$

and $U_m = V_y^T$. Note that the singular vectors are transposed when making this computation!

This finishes our problem, since we have now computed the SVD of the mixing matrix A. The clean signal is now found by taking:

$$S = YV_y$$

We are approximating the mixing matrix by:

$$A \approx V_y^T \Sigma_x^{1/2} V_x = U_m \Sigma_m V_m^T$$

so that the approximate inverse is found via:

$$A^{-1} = V_m \Sigma_m^{-1} U_m^T = V_x^T \Sigma_x^{-1/2} V_y$$

This process of two SVDs, one on the matrix of data X, and another on the data dY can be brought together as a single command. In fact, this process is equivalent to using the *Generalized Singular Value Decomposition*, which will sometimes go under the name of *Quotient Singular Value Decomposition* (GSVD or QSVD, respectively). This simplifies the coding so that you only have to use the following Matlab commands. Let X be the $p \times k$ matrix of k mixtures of signals (this is transposed from our earlier notation). Then signal separation is simply:

```
dX=diff(X);
[U,V,B,C,S]=gsvd(X,dX,0);
```

where the clean mixtures are in the columns of U.

We'll try out both versions in the examples below, then in the next section we'll define the GSVD.

Example

In this example, we will take three columns of data. The first two columns will comprise a circle and the last will be white noise (that is, random data from a normal distribution). We will then multiply this by 3×3 mixing matrix A, which was originally taken so that the elements are from a normal random variable then subsequently hard coded for you to replicate. Denote the mixed data matrix as X, as we did previously. Note that with column-wise data, the mixing matrix equations become:

$$X = SA \Rightarrow X = SU\Sigma V^T$$

where we try to determine U, Σ, V.

Here is the script file to produce the signal separation:

```
%Script file to produce Example 1, ICA
numpts=400;
t=linspace(0,3*pi,numpts);
S=[cos(t'), sin(t'), randn(numpts,1)]; %Separated Signals
A =[  -0.0964   -0.1680    1.6777
    0.4458    0.1795    1.9969
   -0.2958    0.4211    0.6970]; %Mixing Matrix
```

```
X=S*A;

dX=diff(X);
[U,V,B,C,S] = gsvd(X,dX,0); %Clean Signal in U

%The double SVD code, equivalent to the GSVD:
[Ux,Sx,Vx]=svd(X,0);
Y=X*Vx*diag(1./sqrt(diag(Sx)));  %Stopping here is basic PCA
dY=diff(Y);
[Uy,Sy,Vy]=svd(dY,0);
S2=Y*Vy;  %S2 is also the clean signal


%Plotting routines below:
figure(1)
for j=1:3
   subplot(3,1,j)
   plot(U(:,j));  %Clean Signals from GSVD
end
figure(2)
for j=1:3
   subplot(3,1,j);
   plot(S2(:,j));  %Clean Signals from double SVD
end
figure(3)
for j=1:3
   subplot(3,1,j)
   plot(Y(:,j));  %Results of PCA (or KL)
end
```

6.2 Signal Separation of Voice Data

In this example, we will linearly mix two voice signals, then separate them using the techniques we previously described. This example is best done using a computer with a good sound card, but can be done without it.

Matlab comes with several sound files. For this example, we will use `handel` (a sample of the chorus to Handel's "Messiah"), and `laughter` (a sample of people laughing). The two files have different lengths, so we'll have to cut the longer file off so that they match.

Here is the Matlab code:

```
%Script for sound files
load handel
y1=y;
load laughter
S=[y y1(1:52634)];  %Clean samples in the columns of S
A =[ -0.5883   -0.1364; 2.1832    0.1139]; %Mixing Matrix
X=S*A;

mX=mean(X);
X=X-repmat(mX,52634,1);

figure(1)
plot(X(:,1),X(:,2),'.');
```

```
title('Mixed Signals');

%For comparison purposes, here's the SVD
[Ux,Sx,Vx]=svd(X,0);
Y=X*Vx*(1./sqrt(diag(Sx)));

figure(2)
plot(Y(:,1),Y(:,2),'.');
title('KL Results')
%Listen to the results:  You'll still hear mixtures
%soundsc(Y(:,1));
%soundsc(Y(:,2));

dX=diff(X);
[Y2,V,B,C,S3]=gsvd(X,dX,0);
figure(3)
plot(Y2(:,1),Y2(:,2),'.');
title('ICA Results');
%Listen to the results:  They will be clean!
%soundsc(Y2(:,1));
%soundsc(Y2(:,2));
```

More to Think About: There are a lot of experiments you can try with this data. Here are some things you might try:

- Plot the signals as time series if you've never looked at voice data before. Also plot the differenced signal. You might also look at the histograms of the two voice signals using `hist`. Do the signals look like they are normally distributed or do they follow a Laplace distribution?
- Change the mixing matrix to a random matrix. Will you always get good results?
- Listen to the difference matrix dX. Does it still sound like the original? Listen to the second, third, fourth difference. Why does the "derivative" of the signal sound just like the original (perhaps with a different timbre quality, but recognizable just the same)?
- Check the assumptions on the clean signal S and the differenced signal dS- Are the assumptions met?

6.3 A Closer Look at the GSVD

Suppose that we have two matrices $X \in \mathbb{R}^{m\times n}$ and $Z \in \mathbb{R}^{p\times n}$. The GSVD of matrices X, Z is a decomposition where we determine $\hat{U}, \hat{V}, W, C, S$ so that:

$$X = \hat{U}CW^T \qquad Z = \hat{V}SW^T$$

where $\hat{U}, \hat{V}$ have orthonormal columns, C, S are diagonal matrices such that $C^TC + S^TS = I$, and W is an invertible matrix. In Matlab, the command is:

```
[Ux,Vz,W,C,S]=gsvd(X,Z)
```

The values of C, S and W satisfy the following generalized eigenvalue problem:

$$s_i^2 A^T A w_i = c_i^2 B^T B w_i$$

The solution we use for the signal separation is now either:

$$X = \hat{U}CW^T = \hat{U}(CW^T) = SA \text{ or } X = \hat{U}CW^T = (\hat{U}C)W^T = SA$$

In the special case that $s_i \neq 0$, we'll show that we can find x_i using two regular SVD's as we did in the signal separation.

In this case, the eigenvector problem can be written as:

$$X^T X w_i = \lambda Z^T Z w_i \Rightarrow \left(X^T X - \lambda Z^T Z\right) w_i = 0$$

If we let $X = U \Sigma_x V_x^T$ be the SVD of X, then we can rewrite this as:

$$\left(V_x \Sigma_x^2 V_x^T - \lambda Z^T Z\right) x_i = 0$$

This can be factored as:

$$\left[V_x \Sigma_x \left(I - \lambda (\Sigma_x^{-1} V_x^T) Z^T Z (V_x \Sigma_x^{-1})\right) \Sigma_x V_x^T\right] w_i = 0$$

or equivalently, if V_x is square and Σ_x is invertible:

$$\left(I - \lambda (\Sigma_x^{-1} V_x^T) Z^T Z (V_x \Sigma_x^{-1})\right) \Sigma_x V_x^T w_i = 0$$

If we let $Y = Z V_x \Sigma_x^{-1}$, and $q_i = \Sigma_x V_x^T w_i$, the previous equation can be written as:

$$(I - \lambda Y^T Y) q_i = 0$$

Therefore, q_i is an eigenvector of $Y^T Y$, or a right singular vector of Y. We can compute $w_i = V_x \Sigma_x^{-1} q_i$, or

$$W = V_x \Sigma_x^{-1} Q = V_x \Sigma_x^{-1} V_y$$

To summarize, the GSVD is equivalent to two SVDs as follows:

- Let $X = U \Sigma_x V_x^T$ be the SVD of A
- Let $Y = Z V_x \Sigma_x^{-1}$ be a "whitening" transformation.
- Let $Y = U_y \Sigma_y V_y^T$ be the second SVD.
- Final answer: $W = V_x \Sigma_x^{-1} V_y$

To connect this process to our previous signal separation solution, let's recall what we did there: Let X be the mixed signal, dX be the differenced signal (we are now thinking of $Z = dX$):

- Let $X = U_x \Sigma_x V_x^T$ be the SVD of X.
- Let $dY = dX V_x \Sigma_x^{-1/2}$ be the "whitened" signal.
- Let $dY = U_y \Sigma_y V_y^T$ be the second SVD.
- Then $S = Y V_y = X V_x \Sigma_x^{-1/2} V_y = XW$ is the clean signal.

Chapter 7

Local Basis and Dimension

Chapter 8

Data Clustering

8.1 Introduction

The purpose of clustering data is so that we can represent groups of data by an exemplar, or cluster center. This is one way to deal with data sets that would otherwise be too large to handle.

Clustering can be intuitively clear, but algorithmically difficult. For example, consider the data in the first graph in Figure 8.1. I think you would probably agree that there appears to be three natural clusterings.

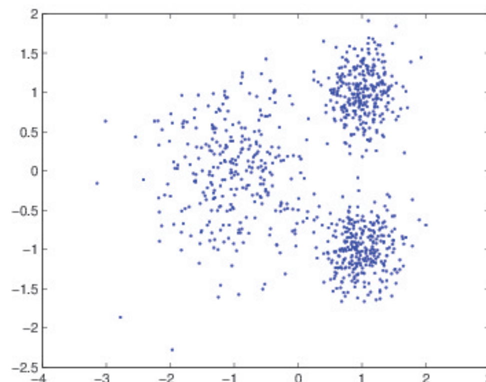


Figure 8.1: We see data that have no class labels, but there seems to be a natural separation of the data into three clumps.

A clustering then is primarily building a function that can take a data point as its input, and then output which cluster (or class) the data point belongs to.

Data may already have labels- We would call a clustering of that type to be *supervised learning*, since there are examples of "correct" labels. In our example, and in general, we'll be working with the unsupervised learning task, where the data points are given to us with no labels.

Definition: Unsupervised Clustering: Given a data set, X, whose elements are vectors $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$, we want to construct a "membership function" which has its domain in $\mathbb{R}^n$ and will output the cluster index (or label). Defining this function as m, we have, for the ith data point, its associated class label g_i:

$$m(\boldsymbol{x}^{(i)}) = g_i$$

where g_i is an integer from 1 to k (k being the number of clusters desired). Later we'll discuss how the algorithm might determine the value of k on its own, so for now, we'll consider k to be given.

The biggest issue with unsupervised clustering is that the problem specification is very ill-posed, meaning that there are many, many ways that one might build such a function m. For two extreme examples, consider these two membership functions:

$$m_1(\boldsymbol{x}^{(i)}) = 1$$

where i is the index for the data points. Another function that is about as useful as m_1 is the following:

$$m_2(\boldsymbol{x}^{(i)}) = i$$

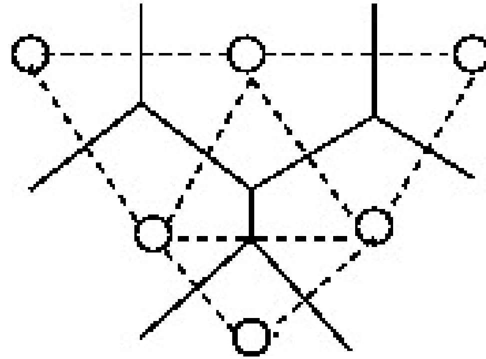
In this case, every data point is its own cluster. Neither situation is ideal, and we'll try to be somewhere in between these extremes.

In order to do determine some "best" clustering, that usually means that we'll be trying to minimize some error function, and the easiest way to do this is through *Voronoi Cells*:

Definition: Let $\left\{\boldsymbol{c}^{(i)}\right\}_{i=1}^{k}$ be points in $\mathbb{R}^n$. These points form k Voronoi Cells, where the jth cell is defined as the set of points that are closer to cell j than any other cluster:

$$V_j = \left\{\boldsymbol{x} \in \mathbb{R}^n \mid \|\boldsymbol{x} - \boldsymbol{c}^{(j)}\| \leq \|\boldsymbol{x} - \boldsymbol{c}^{(i)}\|,\ i = 1, 2, \ldots, k\right\}$$

The points $\left\{\boldsymbol{c}^{(i)}\right\}_{i=1}^{k}$ are called *cluster centers*. In the uncommon occurrence that a point $\boldsymbol{x}$ lies on the border between cells, it is customary to include it in the cell whose index is smaller (although one would fashion the decision on the problem at hand). The reader might note that a Voronoi cell has a piecewise linear border as shown.



Examples in "Nature"

1. In [6], Voronoi cells are used to define the "area potentially available around a tree". That is, each tree in a stand represents the center of the cell.
2. Using a map of the campus with the emergency telephone boxes marked out and used as the centers, we could always tell where the closest phone is located.

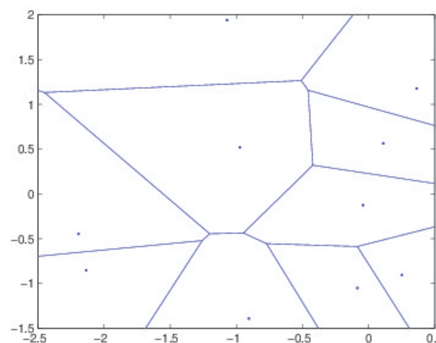
We can draw a Voronoi diagram by hand: Between neighboring cells, draw a line (that will be erased at the end), then draw the perpendicular bisectors for each line drawn. This can get complicated fairly quickly, so we will be using Matlab or Python to produce the plots. The algorithms that do these plots are very interesting (see any text in Computation Geometry) but are beyond the scope of our text.

Matlab Example:

```
X=randn(10,2);  %X is a tall matrix.
voronoi(X(:,1),X(:,2));
```

We can also have Matlab return the vertices of the Voronoi cells to plot them manually:

```
[vx,vy]=voronoi(X(:,1),X(:,2));
plot(vx,vy,'k-',X(:,1),X(:,2),'r*');
```



Python Example

In Python, we'll also build a Voronoi diagram using 10 random points in the plane. For this, well need `Voronoi` from `scipy`, and some other things for displaying the plot.

```
import matplotlib.pyplot as plt
from scipy.spatial import Voronoi, voronoi_plot_2d

A=np.random.rand(10,2)  # Note that A must be a tall matrix.
vor=Voronoi(A)

# Plot the results
fig = voronoi_plot_2d(vor)
plt.show()
```

In the algorithms that we work with, it will be convenient to have a function that will identify points within a given cluster- That is, will have the value 0 if the point is not in a given cluster, or 1 if it is. The "characteristic function" is typical for that purpose, and we'll define it as the following:

$$\chi_i(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \; m(\boldsymbol{x}) = i \\ 0 & \quad \text{otherwise} \end{cases}$$

One general way to measure the goodness of a clustering algorithm is to use a measure called the **distortion error** for a given cluster i, and the total distortion error[1]. Given a clustering of N points, let N_i denote the number of points in cluster i. We first define the error for cluster i, then sum them all for the overall error. Here is the definition for the distortion error for the ith cluster, where we notice that if the point $\boldsymbol{x}^{(k)}$ is not in cluster i, then we simply add 0:

$$E_i = \frac{1}{N_i} \sum_{k=1}^{N} \|\boldsymbol{x}^{(k)} - \boldsymbol{c}^{(i)}\|^2 \, \chi_i(\boldsymbol{x}^{(k)})$$

so that the overall distortion error is defined to be:

$$E_{\text{total}} = \sum_{k=1}^{p} E_k$$

Now that we have a way of measuring the goodness of a clustering, we want to have an algorithm that will minimize the function.

8.2 K-means clustering

A relatively fast method to perform the clustering is the $k-$means clustering. In some circles, this is called Lloyd's algorithm, and there was an update done by Linde, Buzo and Gray to call that the Linde-Buzo-Gray (LBG) algorithm [26].

The membership function for both $k-$means and LBG is defined so that we are in cluster i if we are in the Voronoi cell defined by the ith cluster center. That is,

$$m(\boldsymbol{x}) = i \;\text{ iff }\; \|\boldsymbol{x} - \boldsymbol{c}^{(i)}\| < \|\boldsymbol{x} - \boldsymbol{c}^{(j)}\| \;\; i \neq j, \;\; j = 1 : p \tag{8.1}$$

In the rare circumstance that we have a tie, we'll choose to put the point with the center whose index is smallest (but this is ad-hoc).

[1]Some authors do not square the norms, but it is more convenient for the theory, and does not change the end results.

The Euclidean Distance Matrix

You've seen that we'll need a function that will compute all of the inter-point distances for us. To be more precise, we'll need a function that will input two sets of points in $\mathbb{R}^n$, say there are M points in one set (organized in an $M \times n$ matrix), and N points in another (organized in an $N \times n$ matrix). The output will then be an $M \times N$ matrix where the (i, j) position is the distance between the point i in the first set and point j in the second set.

For Matlab/Octave, we'll write a short function to do this for us called `edm.m`. which will be available on the class website. For Python, you can compute the pairwise distances using the example below:

```
from sklearn.metrics.pairwise import euclidean_distances

X = [[0, 1], [1, 1]]
D=euclidean_distances(X, X) #D is 2 x 2 with distances.
```

The Algorithm

Keeping in mind that we'll be using Matlab for its implementation, we'll write the algorithm in matrix form.

Let X be a matrix of p data points in $\mathbb{R}^n$ (so each data point is a row, and the matrix is $p \times n$), and let C denote a matrix of k centers in $\mathbb{R}^n$ (each "center" is a row, and the matrix is $k \times n$).

At each pass of the algorithm, the membership function requires us to take $p \times k$ distance calculations, followed by p sorts (each point has to find its own cluster index), and this is where the algorithm takes most of the computational time.

K-Means Clustering

Given: p data points in $\mathbb{R}^n$ and k cluster centers (as vectors in $\mathbb{R}^n$). Then:

- Sort the data into k sets by using the membership function in Equation 8.1.
- Re-set center $\boldsymbol{c}_i$ as the centroid (mean) of the data in the ith set.
- Compute the distortion error.
- Repeat until the distortion error no longer decreases (either slows to nothing or starts increasing).

A Toy Example:

Here we use the $k-$means algorithm three times on a small data set, so you can have a test set for the code that you write.

Let the data set $X^{5\times 2}$ be the matrix whose rows are given by the data points in the plane:

$$\left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\}$$

You might try drawing these on a plane. You may see a clustering of three points to the right, and a clustering of two points to the left.

We will use two cluster centers, (typically initialized randomly from the data) in this case:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The EDM of distances will be $D^{5\times 2}$ and is the distance between our 5 data points and 2 cluster centers.

The membership function then looks for the minimum in each row, shown as the vector M below:

$$D = \begin{bmatrix} \sqrt{2} & 1 \\ 0 & 1 \\ \sqrt{2} & \sqrt{5} \\ 1 & 0 \\ \sqrt{5} & \sqrt{8} \end{bmatrix}, \qquad M = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$

Resorting into two clusters,

$$\left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\} \qquad \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

The cluster centers are re-set as the centroids:

$$\begin{bmatrix} -2/3 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 3/2 \end{bmatrix}$$

After one more sort and centroid calculation, we get:

$$\begin{bmatrix} -1 \\ -1/2 \end{bmatrix}, \begin{bmatrix} 2/3 \\ 4/3 \end{bmatrix}$$

which does not change afterward (so the clusters also remain the same).

The basic algorithm may have a few shortcomings. For example,

- The centers may be (or become) empty- That is, there are no data points closest to a given center. This is a good chance of this happening if you simply choose random numerical values as the initial centers, that's why we recommend choosing random *data points* as initial centers- You'll have at least one point in each cluster that way.
- We had to pre-define the number of clusters. Is it possible to construct an algorithm that will grow or shrink the number of clusters based on some performance measure? Yes, and that change is the LBG algorithm.

Matlab Note

If the "Statistics Toolbox" is available for your copy of Matlab, then `kmeans` is a built-in command.

Matlab's clustering routine expects the data to be given row-wise, so if we have p points in $\mathbb{R}^n$, then Matlab expects the data matrix to be $p \times n$.

Here's the previous example using the stats toolbox:

```
X=[1,0,-1,1,-1;2,1,0,1,-1]';
C=[0,1;1,1]';
[M,C1]=kmeans(X,2,'start',C);
```

If we didn't have an initial clustering C, the command is a lot shorter. For example, with just the data and number of clusters, we would have:

```
X=[1,0,-1,1,-1;2,1,0,1,-1]';
[M,C]=kmeans(X,2);
```

Matlab gives us several ways of initializing the clusters- See the documentation.

Exercises

1. Fill in the missing details from the example in the text: Fill in the EDM and the vector M after the second sort, and continue the example one more iteration to show that the clusters do not change. Hint: To compute the distance matrix, use the points as they were originally ordered.

2. Given p scalars, $x_1, x_2, \ldots, x_p$, show (using calculus) that the number μ that minimizes the function:

$$E(\mu) = \sum_{k=1}^{p} (x_k - \mu)^2$$

   is the mean, $\mu = \bar{x}$. Hint: We think of $x_1, \ldots, x_p$ as being fixed numerical values, so E is a function of only one variable μ.

3. Generalize the last exercise so that the scalars (and μ) are now vectors in $\mathbb{R}^n$. That is, given a fixed set of p vectors in $\mathbb{R}^n$, show that the vector μ that minimizes:

$$E(\mu) = \sum_{k=1}^{p} \|\mathbf{x}^{(i)} - \mu\|^2$$

   is the mean (in $\mathbb{R}^n$).

4. Write the following as a Matlab function. The abbreviation EDM is for Euclidean Distance Matrix. Good programming style: Include comments!

```
function z=edm(w,p)
%  A=edm(w,p)
%  Input:  w, number of points by dimension
%  Input:  p is number of points by dimension
%  Ouput:  Matrix z, number points in w by number pts in p
%          which is the distance from one point to another

[S,R] = size(w);
[Q,R2] = size(p);
p=p';
if (R ~= R2), error('Inner matrix dimensions do not match.'),end

z = zeros(S,Q);
if (Q<S)
  p = p';
  copies = zeros(1,S);
  for q=1:Q
    z(:,q) = sum((w-p(q+copies,:)).^2,2);
  end
else
  w = w';
  copies = zeros(1,Q);
  for i=1:S
    z(i,:) = sum((w(:,i+copies)-p).^2,1);
  end
end
z = z.^0.5;
```

5. If you don't have the `kmeans` function built-in, try writing one using the EDM function. It is fairly straightforward using the EDM function and the min function.

6. Will the cluster placement at the end of the algorithm be independent of where the clusters start? Answer this question using all the different possible pairs of initial points from our toy data set, and report your findings- In particular, did some give a better distortion error than others?

LBG modifications

We can think about ways to prune and grow the number of clusters, rather than making it a predefined quantity. Here are some suggestions for modifications you can try. This actually changes the $k-$means algorithm into the LBG algorithm.

- Split that cluster with the highest distortion measure, and continue to split clusters until the overall distortion measure is below some preset value. The two new cluster centers can be initialized a number of ways- Here is one option:
$$\boldsymbol{c}^{(i_1,i_2)} = \boldsymbol{c}^{(i)} \pm \boldsymbol{\epsilon}$$
However, this may again lead to empty clusters.
- We can prune away clusters that either have a small number of points, or whose distortion measure is smaller than some pre-set quantity. It is fairly easy to do this- Just delete the corresponding cluster center from the array.

8.3 Neural Gas

So far, the clustering algorithms we have considered use arithmetic means to place cluster centers. This works well when the data consists of "clouds" (coming from a random process, for example). This process does not work well when the data lies on some curved manifold[2]

If we imagine all of the data to be on the boundary of a circle, for example, taking an average will place the cluster centers off the circle.

Figure 8.2: If the data lies on a curved surface, then using the mean may pull the centers (stars in the graph) out of the data.

But there is something that may be exploited about the cluster centers to help us in visualizing how the data lies in its space.

It would be helpful if we knew what kinds of *structure* the data may have. For example, in Figure 8.2, it would be helpful to know that the data is actually (locally) one dimensional.

The method we will use to determine the structure of the data will be to define connections (or edges) between clusters. Consider the data in Figure 8.3. It is this kind of structure that the clustering should reflect- Which cluster centers are "neighboring" points in the data? Just as importantly, which are *not* neighbors in the data?

[2]A manifold is generally defined to be an object that looks locally like $\mathbb{R}^k$ for some k. For example, a path in three dimensions is locally 1 dimensional.

Figure 8.3: The star shaped points are the cluster centers. It would be nice to know which cluster centers belong together in a single "blob" of data, versus which cluster centers are in the other set of data.

In mathematical terminology, we want to find what is called a *topology preserving* clustering. For us, the *topology* of the clusters is defined by how the cluster centers are connected to each other (that is, so we know which cluster centers are neighbors and which are not).

More generally, we want the clustering to be what is called "topology preserving" in the sense that clusters defined to be neighbors in the cluster center topology are actually neighbors in the data, and vice versa. Before continuing, some graphical examples should help us with this concept.

In Figure 8.4, we see three topologies mapped to the data, which is a uniform distribution of points in $\mathbb{R}^2$. In the first picture, the topology of the cells is a one dimensional set. In this situation, the cluster mapping is not topology preserving, because neighboring cells in the topology are not adjacent in the plane. In the second situation, we have a three-dimensional topology mapping to the plane. In this case, neighboring data points in $\mathbb{R}^2$ are mapped to non-neighboring cells in the topology. Only in the third picture do we see that both parts of the topology preserving mapping are satisfied.

The Neural Gas Algorithm [27, 29, 28] is designed with a couple of big changes in mind (changes from k-means and LBG).

1. We don't want to use arithmetic means if possible. It is more desirable to have cluster centers that are actually embedded in with the data (like in Figure 8.3) rather than every cluster outside of the data (like in Figure 8.2).

2. We want the algorithm to build connections between the cluster centers. Hopefully this will tell us if we have separated regions of data (like in Figure 8.3).

Embedding the Clusters

To solve the first problem, we are going to endow the data with the power to attract the cluster centers. That is, when a data point $\mathbf{x}$ is chosen, we will determine which cluster center is closest (the "winner"), and we will move the cluster center towards that point. We don't want to go very far, just a slight move in that direction.

Mathematically, for a given data point $\mathbf{x}$, find the winning center, $\mathbf{c}$ and move in the direction $\mathbf{x} - \mathbf{c}$. The update rule would then look like:

$$\boldsymbol{c}_{\text{new}} = \boldsymbol{c}_{\text{old}} + h(t)(\mathbf{x} - \mathbf{c}_{\text{old}}) \tag{8.2}$$

where $h(t)$ will be a function determining how far to move. Normally, at the beginning of the clustering, h is relatively large. As the number of iterations grow larger, h gets very small (so that the movement at each iteration gets very slight).

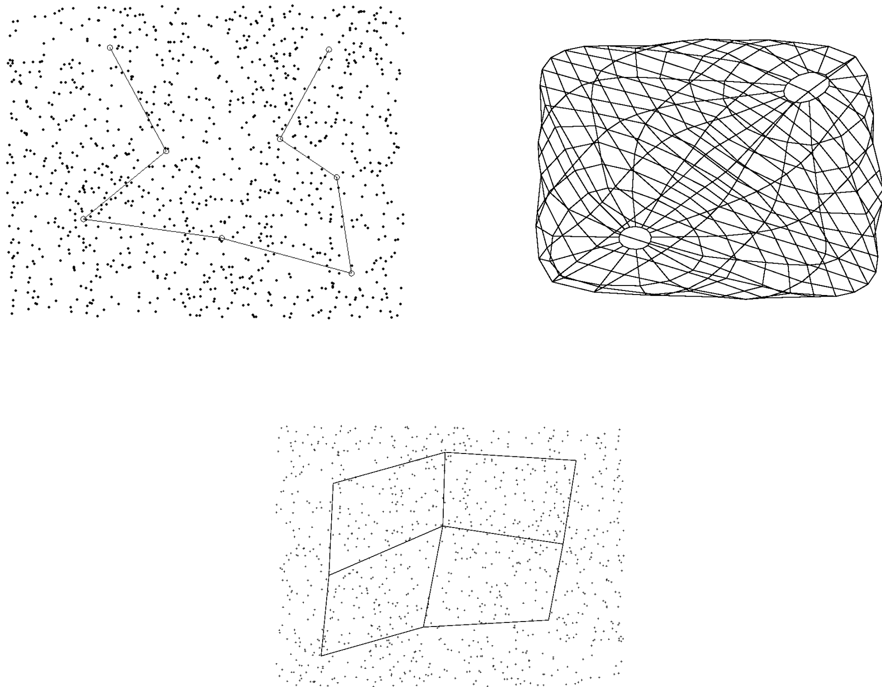


Figure 8.4: Which mapping is topology preserving? Figure 1 shows a mapping that is not topology preserving, since neighboring cells are mapped to non-neighboring points in $\mathbb{R}^2$. On the other hand, the middle clustering is not topology preserving, because neighboring points in $\mathbb{R}^2$ are mapped to non-neighboring cells in the topology. Only the third picture shows a topology preserving clustering.

This method would work very well, but it might result in a lot of empty cluster centers (centers that are never winners). This is where the concept of the "gas" comes into play- We think of the data as being in a viscous fluid, like jello.

How does this work in practice? As before, we first select a data point at random, and find the winning cluster center. We move the cluster center towards the data point. In a viscous fluid, that means that the cluster centers close to the winner will ALSO move toward the data point (just not as much). Therefore, instead of updating the single winning center using Equation 8.2, we update all centers.

The key point is that the winning center is attracted to $\mathbf{x}$ with the strongest force, then the next closest center is attracted with the next strongest force, and so on. To do this, we'll need to sort the centers by their distance to our data point $\mathbf{x}$.

You might have a negative reaction at this point- Given a lot of cluster centers, this is a lot of sorting. What is typical is that we will define a value k, and we will only update the k closest centers.

We define a way of setting this measure: We define s_i to be the number of centers closer to $\mathbf{x}$ than $\boldsymbol{c}^{(i)}$. The easiest way to compute this is to sort the centers by their distance to $\boldsymbol{x}$, then put the first k indices in a vector $\boldsymbol{v}$. So, for example, the vector $\boldsymbol{v}$ can be defined as:

$$\boldsymbol{v} = \{i_1, i_2, \ldots, i_k\}$$

so the "winning" center has index i_i. Now we define the values s:

$$s_{i_1} = 0, \quad s_{i_2} = 1, \quad s_{i_3} = 2, \quad \ldots, \quad s_{i_k} = k - 1$$

Example: Let $C = 0.1, 0.2, 0.4, 0.5$. If $x = 0.25$, then $i_1 = 2$, and $\boldsymbol{v} = \{2, 1, 3, 4\}$. The values of s: $s_2 = 0$, $s_1 = 1$, $s_3 = 2$, $s_4 = 3$.

Building Connections

Now that we have set up an algorithm that moves the centers into the data, we look at the problem of defining connections. The underlying idea is that we'll connect a winning center to its neighbor, indexed by $\boldsymbol{v}(2)$. If that connection has not been selected for a long time, that probably means that over time, the centers have drifted apart, and we should remove the connection.

Therefore, to build connections, we will use two arrays (if the number of centers is k, they are both $k \times k$), M and T.

Definition: A Connection Matrix, M, is a matrix of 1's and 0's where

$$M_{ij} = \begin{cases} 1 & \text{If cell i connected to j} \\ 0 & \text{Otherwise} \end{cases}$$

To build the connection matrix, once $\boldsymbol{v}$ is determined, then M_{i_1,i_2} is set to 1, and the age of the connection T_{i_1,i_2} is set to 0. All the other ages have 1 added to them.

Lastly, we should check to see how many iterations have passed since each edge was constructed. If too much time has passed (pre-set by some parameter), then remove those connections.

Parameter List

The number of parameters we'll use is a little large, but coding them is no problem. Here is a list of them, together with some standard values, where N is the number of data points.

- Number of data points: N
- Relative maximum distance to move, if we update the winning center: ϵ

$$\epsilon_{initial} = 0.3 \qquad \epsilon_{final} = 0.05$$

- Parameter λ also gives some control over how much to move the winner and neighbors.

$$\lambda_{initial} = 0.2N \qquad \lambda_{final} = 0.01$$

- The maximum amount of time before a connection is removed is:

$$T^m_{initial} = 0.1N \qquad T^m_{final} = 2N$$

- Maximum number of iterations: `tmax`$= 200N$.

The parameters that have an "initial" and "final" time are updated using the power rule. If the initial value is $\alpha_{initial}$ and the final value is α_{final}, then on iteration i,

$$\alpha_i = \alpha_{initial} \left(\frac{\alpha_{final}}{\alpha_{initial}} \right)^{i/\text{tmax}} \tag{8.3}$$

The Neural Gas Algorithm:

1. Select a data point $\boldsymbol{x}$ at random, and find the winner, $c^{(i_1)}$.
2. Compute $\boldsymbol{v}$ by finding the next $k-1$ closest centers.
3. Update the k centers:
$$c^{(i_k)} = c^{(i_k)} + \epsilon \exp\left(\frac{-s_{i_k}}{\lambda} \right) (\boldsymbol{x} - c^{(i_k)})$$
4. Update the Connection and Time Matrices: Set $M_{i_1,i_2} = 1$, and $T_{i_1,i_2} = 0$.
   Age all connections by 1, $T_{j,k} = T_{j,k} + 1$ for all j, k.
5. Remove all old connections. Set $M_{j,k} = 0$ if $T_{j,k} \geq T^m$
6. Repeat.

2021 Note: I'm pulling all of the implementation details out of the main text and will provide it separately- These details can change a lot over the years.

8.3.1 Project: Neural Gas

This project explores one application of triangulating a data set: Obtaining a path preserving representation of a data set.

For example, suppose we have a set of points in $\mathbb{R}^2$ that represents a room. We have a robot that can traverse the room - the problem is, there are obstacles in its path. We would like a discretization of the free space so that the robot can plan the best way to move around the obstacles.

The data set is given in `obstacle1.mat`, and is plotted below. The areas where there are no points represent obstacles. **Assignment:** Use the Neural Gas clustering using 1,000 data points and 300 cluster centers to try to represent the obstacle-free area.

8.4 DBSCAN

The method known as "Density-based spatial clustering of applications with noise", or more simply as DBSCAN[3] is called a *density-based* clustering algorithm. What makes it so distinctive from $k-$means is that it can create complex shapes for each "cluster", and secondly, it can identify "outliers", or points that shouldn't be classified as members of any cluster.

[3] Ester, Kriegel, Sander, and Xu, "A Density Based Algorithm for Discovering Clusters", KDD-96 Proceedings, 1996.

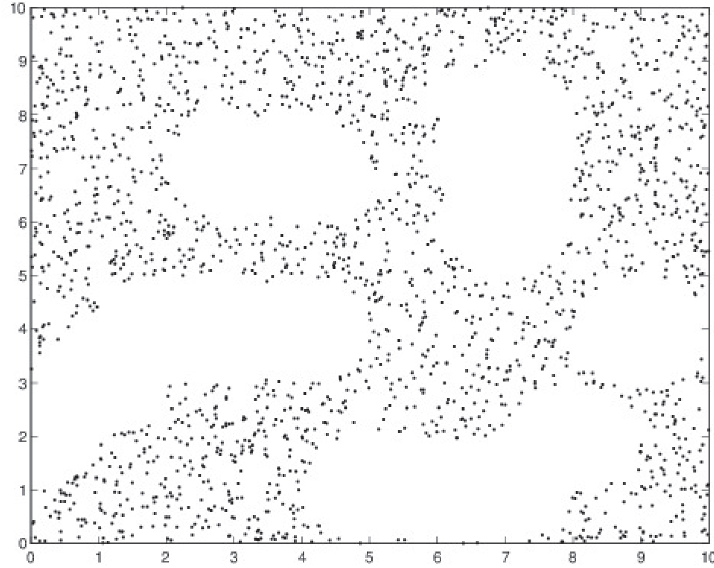


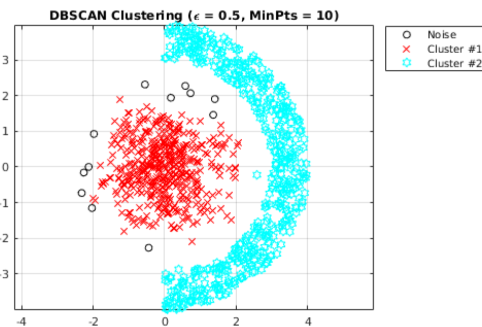
Figure 8.5: The data set that represents the ”free” area. White regions indicate obstacles.

In the method, a “cluster” is defined by growing a set of points using a density criterion. That is, to be considered neighbors, a point will need to be within a given ϵ, of the center, and further, there will need to be a certain number of points also within ϵ before the set is considered a cluster. In this way, we might think of a cluster as, what some people would call, a “maximal set of density-connected points”.

So given ϵ and a value for the smallest number of points allowed to make a cluster, `MinPts`, the algorithm separates data into three categories: A **core** point, a **border** point, or **noise**.

- A point is a **core** point if it has more than `MinPts` within ϵ.
- A point is a **border** point if if has fewer than `MinPts` within ϵ, but is still in the neighborhood of a core point.
- A **noise** point is a point that is not a core point, nor a boundary point.

What results from the DBSCAN method is not a set of cluster centers- Rather, each “core point” is provided with an index of which cluster it belongs to. So the plot of the output is straightforward, and one such output is given in the figure to the right.



The algorithm for DBSCAN proceeds with two helper functions: One function will will take in the core point and epsilon, and will indicate which points are within ϵ of the core point. In our Matlab code, this is `RegionQuery`.

The second helper function is designed start with some set of neighboring points from a core point. It then expands the given cluster by seeing if the neighboring points also contain the minimum number of neighbors within ϵ, and continues until it runs out of points. We would say in this case that the helper function is collecting all points that are “density-reachable” from the core. Let’s define that term- it is actually a bit nuanced, so we’ll take a little side trip to discuss it.

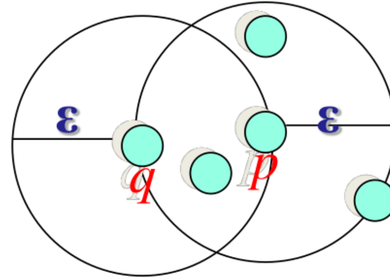
8.4.1 Density Reachable Points

Definition: A point q is **directly density-reachable** from a point p if point p is a core point, and q is in the $\epsilon-$neighborhood of p.

Below is an example of two points, p and q with ϵ shown, and `MinPts=4`. We see that point p has 4 points within its neighborhood, so point p is a core point. Point q has only three points in its neighborhood, so it is NOT a core point.

In the figure[a] to the right, point q is directly density-reachable from p, but p is not directly density-reachable from q. This is an important example, in that it shows that the property of density-reachability is asymmetric.

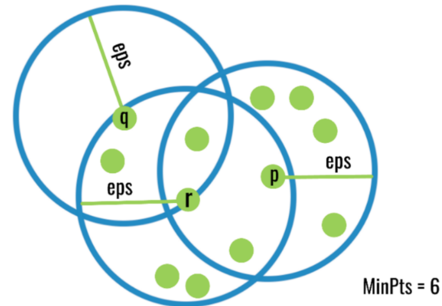
[a] Adapted from a talk by Prof Jing Gao of SUNY Buffalo.



Next we have simply **density-reachable**:

Definition: A point p is density-reachable from a point q with respect to ϵ and `MinPts` if there is a chain of points $p_1, p_2, \ldots, p_n$, where $p_1 = q$ and $p_n = p$, such that p_{i+1} is directly density-reachable from p_i.
In the figure[a] to the right, the point q is density reachable from p through r. But again, as last time, p is not density reachable from q since q is not a core point (`MinPts=6`).

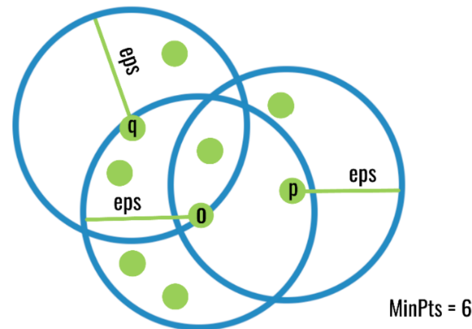
[a] From "ML—DBSCAN reachability and connectivity", from *GeeksforGeeks.com*



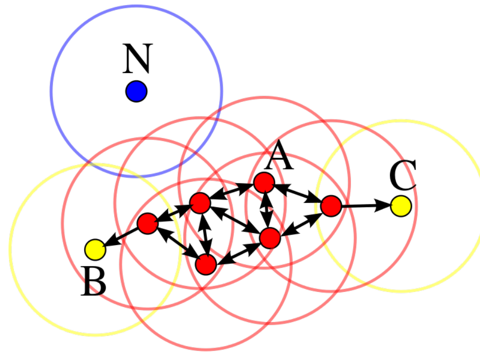
In our last definition, consider two border points of the same cluster C. They are possibly not density reachable from each other because of the core point condition. However, there must be a core point in C from which both border points of C **are** density reachable. This this is the motivation behind the next definition.

Definition: A point p is **density-connected** to a point q with respect to ϵ and `MinPts` if there is a point o such that both p and q are density-reachable from o with respect to ϵ and `MinPts`.
We might note that if p is density connected to q, then q is density connected to p. In the figure[a] to the right, `MinPts=6`, so points p, q are not core points. However, they are both reachable from core point o, so they are density connected.

[a] From "ML—DBSCAN reachability and connectivity", from *GeeksforGeeks.com*



Before we go to the pseudocode for the algorithm, let's look at one more example, where we can consider the three types of points and what they might contribute to the clustering. Consider the figure below.



In this diagram[4], `minPts`= 4. Point A and the other red points are core points, because the area surrounding these points in an ϵ radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and thus belong to the cluster as well. Point N is a noise point that is neither a core point nor directly-reachable.

From the pseudocode for DBSCAN below, you'll see that the concept of density-reachable points is critical to the algorithm.

8.4.2 Pseudocode for DBSCAN

Here is the pseudo-code for DBSCAN. It's actually quite simple:

```
for each point x in X do:
   if x is not yet classified then:
       if x is a core point then:
            collect all points density-reachable to x,
            and assign them to a new cluster.
       else
            assign x to NOISE
```

We will see this structure in the Matlab/Octave files[5] that we'll be using for the homework. Python has a built-in command, and you may use that as a substitute, although you should look at the Matlab code.

Theoretical Results

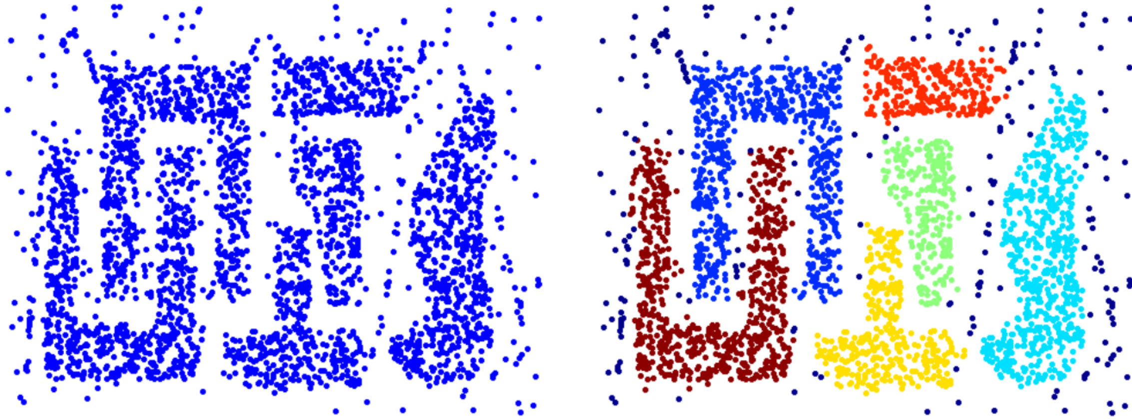
There are a couple of rather nice results given in the original paper characterizing the clusters that DBSCAN produces.

- For all points p, q in a cluster C, p and q are density-connected.
- Given a cluster C with an arbitrary core point p, the cluster C is equal to the set of all points that are density-reachable from p.
- Points that are not directly density-reachable from at least one core point are classified as *noise*.

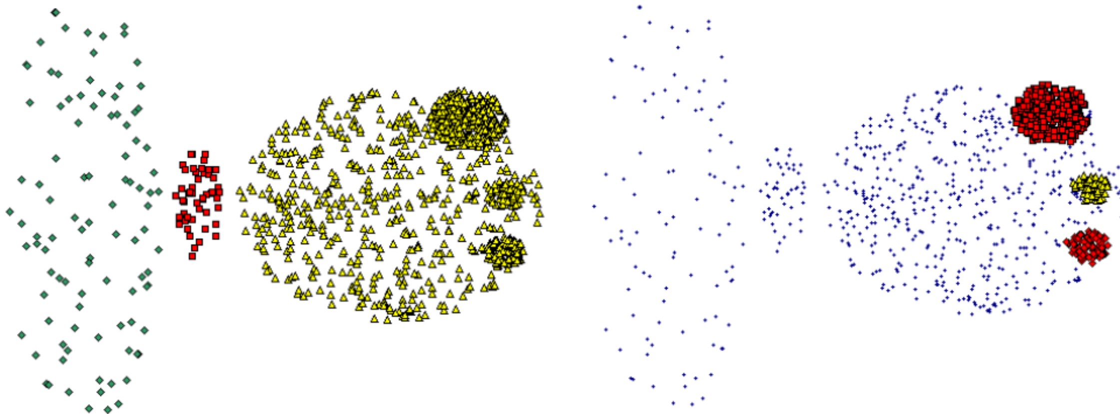
Examples and Rules of Thumb

Finally, we need to consider when DBSCAN works well, and when it doesn't (or might not) work well. DBSCAN, as we introduced it, works very well to construct odd-shaped clusters and to identify what might be noise. Here is a template example of that below[6], with the original data given to the left, and a result of DBSCAN output to the right. It seems like an intuitively good clustering.

[4] Figure and caption from https://en.wikipedia.org/wiki/DBSCAN
[5] The files we'll use were authored by S. Mostapha Kalami Heris of *Yarpiz.com*, 2015.
[6] Adapted from a talk by Prof Jing Gao of SUNY Buffalo.



So when might DBSCAN have trouble? The biggest problem with DBSCAN is that the clustering it provides can be extremely sensitive to the two parameters that the user has to provide (ϵ and `MinPts`). We'll see examples of that in the homework, but there's a nice example below[7] where the original data consists of several regions of differing size and differing density- with some very dense clusters buried inside the less dense cluster. We can see that changing the ϵ by a very small amount completely changes the nature of the clusters found. For the results on the left, the parameters were $\epsilon = 9.92$ and `MinPts=4`. On the right, same `MinPts`, but ϵ was changed to 9.75.



There are some rules of thumb for setting the parameters[8]. For example, `MinPts` should be at least twice the dimension of the data. If our data is in the plane, that would make the default value 4. It is recommended that for datasets that have a lot of noise, that are very large, that are high- dimensional, or that have many duplicates it may improve results to increase `MinPts`.

The value of ϵ is harder to estimate. Here are some considerations:

- ϵ should be as small as possible.
- Some researches say you should measure the distance to the fourth nearest neighbor for 2-dimensional data, or the distance to the kth neighbor, where k is twice the dimension.
- The value might be highly problem dependent. For example, in a clustering problem using GPS, it might make physical sense to set ϵ to 1 km.

[7] Adapted from a talk by Prof Jing Gao of SUNY Buffalo.

[8] From Schubert, Sander et al in "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN" ACM Transactions on Database Systems. 2017

And of course, there is the question of how much work should we put into the estimate, when varying these parameters may give us insight into what the point density is like in our example.

8.5 Comparisons between the algorithms

As we look back at the algorithms, each was designed to extract different information from data.

1. $k-$means:

   - The $k-$means algorithm forms Voronoi cells in the plane.

     If believe we have clusters that can be separated by the polygonal regions the the Voronoi cells produce, then $k-$means may work well.

     In our examples, $k-$means would not have worked well for something like interwoven spirals.
   - The $k-$means algorithm is fast and the output is easy to interpret (giving you templates for the data).

     We found that $k-$means did a fantastic job separating colors in the image segmentation problem, and in clustering the handwritten digits.
   - How are new points handled?

     As another point of comparison, if you're given new data points, it is easy to find which cluster the point belongs to: Take the distance to each cluster, and find the center closest to it.
   - Probably the biggest issue with using $k-$means is that the clustering you get depends on the initialization.

2. Neural Gas:

   The Neural Gas algorithm is an algorithm designed to uncover the topological structure of the data. The cluster centers here are not centers in the $k-$means sense; they do not represent a small number of template data points (like in the handwritten digits example). However, if we do want to use the cluster points as centers, we can do that.

   - The cluster points, used as cluster centers, form Voronoi cells for membership- like $k-$means.

     We have had some success in treating these as cluster centers when the data actually represents an object in lower dimensional space (something called *neural charts*[9]). In this case, we wanted to construct local coordinate systems (using a local version of the PCA), and by using Neural Gas, we were able to make sure that the origin to each coordinate system was actually embedded in the data.
   - How are new points handled? As with $k-$means, the cluster with the closest distance would give the cluster membership.
   - As with $k-$means, probably the biggest issue with Neural Gas is that the cluster points and connections change depending on the initialization (and of course with how you set the parameters).

3. DBSCAN:

   DBSCAN is an interesting algorithm in that it allows you to pick out "noisy" points. The values of the two inputs, ϵ and `MinPts` can be adjusted from anywhere between two extremes- On one extreme, every point is "noise" (ϵ too small or `MinPts` too large), or on the other extreme, the entire data set is one big cluster (ϵ too large and/or `MinPts` too small).

   - We have no cluster points per se, every point is either in a cluster (with an index provided), or is classified as noise.

[9] "Empirical Dynamical Systems Reduction II: Neural Charts", D. Hundley and M. Kirby. In *Semi-analytic methods for the Navier-Stokes Equations*, Edited by K. Coughlin, p. 65-83, 1999.

- With new data, we would either have to re-run the algorithm, or come up with some algorithm to decide which cluster it should belong to:
  - With only the original data available, we could find k closest original data points from the new point, and either use the closest point or some weighted average of the k original point classifications to determine which cluster the new point belongs to.

Part II

Functional Representations

Chapter 9

Optimization

9.1 Introduction: Going from Data to Functions

In this chapter, we take a turn into the heart of function building (which is, in turn, the primary topic of machine learning and model building as well).

In many cases, we'll have some kind of proposed form for a function. For example, in the line of best fit problem, the proposed form for the line was: $y = mx + b$, or $F(x) = mx + b$, where we needed to determine the model parameters m and b. Given data points (x_i, t_i) where t is for targets, this translated to our goal of finding m, b that satisfied some equation or system of equations:

$$\begin{array}{rl} F(x_1) & = t_1 \\ F(x_2) & = t_2 \\ \vdots & \\ F(x_p) & = t_p \end{array}$$

There was no exact solution to this system, which gives us error, which is typically a sum-of-squares:

$$E(m, b) = (t_1 - y_1)^2 + (t_2 - y_2)^2 + \cdots + (t_p - y_p)^2 = \sum_{k=1}^{p} (t_k - y_k)^2$$

where y_i is our model output, $F(x_i)$. And here we go- We need to minimize the error. In fact, most problems in machine learning (and model building) are cast as optimization problems.

How do we optimize a function? We learned the steps back in Calculus, and the algorithm depended on whether or not we had a bounded domain. We'll assume a domain that is not bounded for the remainder of this chapter, unless specified otherwise.

In the unbounded case, we know that the candidates for the maximum and minimum are found among the critical points of f (where $f'(x) = 0$ or where the derivative does not exist). Therefore, we need to have an algorithm that can seek out the roots (or the zeros) of a function.

9.2 Optimization and Calculus

As discussed, there is a close relationship between finding the roots to a function and optimizing a function. For the roots, we solve for where $g(x) = 0$, and in the optimization problem, we solve for where $f'(x) = 0$.

Therefore, discussions about optimization often turn out to be discussions about finding roots. In that spirit, we look at a very basic algorithm for finding roots- An algorithm called "The Bisection Method".

9.2.1 The Bisection Method

The motivation for this method comes from the Intermediate Value Theorem from Calculus- In this case, suppose that our function g is continuous on $[a, b]$, and further $g(a)$ and $g(b)$ have different signs (more compactly $g(a)g(b) < 0$), then there is a c in the interval such that $g(c) = 0$.

Now, given such an interval $[a, b]$, we can get a better approximation by breaking the interval in half. Here is the algorithm:

The Bisection Algorithm to Solve $f(x) = 0$

- Initialize with function $f(x)$, values of a, b and tol, so that the root of f is inside the interval $[a, b]$ (we should have $f(a)f(b) < 0$).
- Set $f_a = f(a), f_b = f(b)$.
- Main loop:
  - Set $c = (a + b)/2$, and compute $f_c = f(c)$.
  - If $f_a f_c < 0$, then the new interval is $[a, c]$. Set $b = c$, $f_b = f_c$.
    Otherwise, the new interval is $[c, b]$. Set $a = c$, $f_a = f_c$
  - Stopping criteria (one or more):
    - $(b - a)/2 < tol$, or
    - $|f_c| < tol$ (or $f(c)$ is actually 0)
  - At the end, output the midpoint of the current interval as the solution: $(a + b)/2$.

The nice thing about the bisection method is that it is easy to implement. It takes a lot of iterations to converge, however.

There are examples of this algorithm available to you in Matlab and Python that we wrote ourselves directly from our pseudo-code. At the end of this section, we'll discuss the built-in code.

When using the Bisection Method in either Matlab or Python, we need to be able to pass in the name of the function to our algorithm. Here is an example, where we solve $x^3 + x - 1 = 0$ with a starting interval of $[0, 1]$ and a tolerance of "5e-5" (which is short for scientific notation, 5×10^{-5}).

In Matlab, we'll assume that `bisect.m` is in the folder or search path, and in Python, we'll assume that the **def**inition of the function **bisection** has been run (again, as a reminder, these are functions that we defined and are available on our class website).

Below we show how to create what is referred to as an **anonymous function**, and we'll use that to pass in the function $g(x)$ into our algorithm. The nice thing about these kinds of functions is that they do not need a separate file to define them (like Matlab), nor do the need a whole **def** section in Python. Of course, if your function is more complicated, you may need to go that route, and in the next section we'll have an example of creating that kind of function.

Matlab Code Example	Python Code Example
`>>f= @(x) x.^3+x-1;`	`>>> f = lambda x: x**3 + x - 1`
`>>x=bisect(f,0,1,5e-5);`	`>>> x=bisection(f,0,1,5e-5)`

9.2.2 Newton's Method

Another method we have from Calculus is Newton's Method. Newton's Method is a lot faster, but it also requires the computation of the derivative. The formula is given below. If you're not sure where the formula comes from, we'll review it in class (you can also look it up online or in a calculus text).

Newton's Method

Newton's method is used to locate a root to a function g. It uses an initial guess and produces a sequence of values that (hopefully) converge to a root.

Given $g(x)$, and an initial guess of the root, x_0, we iterate the following:

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

Coding this is straightforward; the biggest consideration is how we'll pass in the function. It might be easiest to have the function compute both the value of the function *and* the value of the derivative at a given point. That is currently the way the code is written.

We might mention that the code might be more streamlined if you have one (anonymous) function to handle the computation of g, and a second to compute the computation of the derivative, g'. Feel free to make those changes.

To use either the bisection or Newton's method Matlab functions, we'll need to define the function (that we're finding the roots to). You can either use an anonymous function handle (defined in-line) or use an $m-$file- It is useful to go ahead and program in both f and the derivative so you can use either bisection or Newton's method. Here's an example finding a root to $\mathrm{e}^x = 3x$.

First, we'll write a function file. To the left is Matlab, to the right is Python:

```
function [y,dy]=testfunc01(x)
% Test function e^x-3x
y=exp(x)-3*x;  % Output y
dy=exp(x)-3;   % and dy
```

```
import math

def testfunc01(x):
y=math.exp(x)-3*x
dy=math.exp(x)-3
return y,dy
```

We'll put together examples in Matlab and Python. To keep things tidy, we'll have the function definitions in the same file as the script. (This is available as `ExScript1.m` or `ExScript1.py` on our class website).

Built-in Methods

I don't believe that Matlab has the Bisection Method or Newton's Method specifically built-in on their own. Rather, it uses `fzero` as a general solver for the roots of a function, but it uses methods that don't rely on the derivative (bisection is one option), so it is slower. It also provides `fminbnd, fminunc, fmincon` and others in the Optimization toolbox. For now, we'll just stick to our two algorithms.

In contract, in Python we have `scipy.optimize.newton()`, where the inputs vary depending on the information available.

9.3 Homework

1. It's always good to be able to do a simple example of these algorithms "by hand". Try these out (you may use a hand calculator):
   (a) Estimate the first root of $e^x - 5x$ by computing three iterations of the bisection method on the interval $[0, 1]$.
   (b) Estimate the second root of $e^x - 5x$ by computing Newton's Method twice, with a starting guess of $x_0 = 2$.
2. Use the example script file in Matlab or Python to solve the next two problems. You can put them both on the same script, both functions can be anonymous.
   (a) Use the bisection method to find the root correct to 6 decimal places: $3x^3 + x^2 = x + 5$.

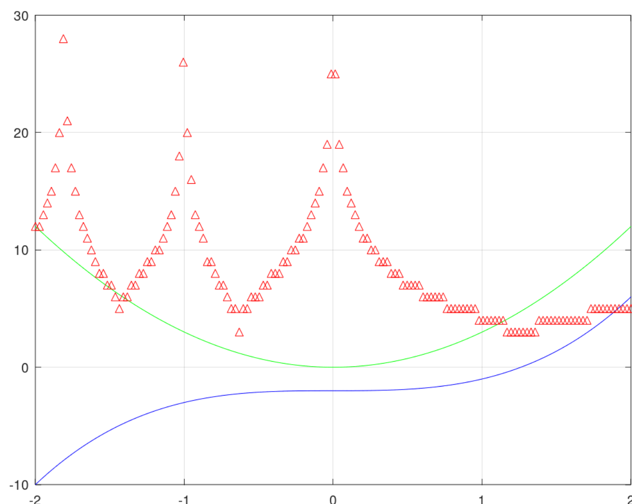
(b) Use Newton's Method to find the same root.

3. In this exercise, we want to look at how fast Newton's Method will converge as a function of the initial guess. We'll use $f(x) = x^3 - 2$ (so we'll be estimating $\sqrt[3]{2}$). We will choose the initial guess x_0 from the interval $[-2, 2]$, and run Newton's Method- Keeping track of how many iterations it takes to converge.

   We then repeat this for another x_0, and another x_0- In fact, we take 150 values between $[-2, 2]$ for the initial condition, and plot the number of iterations for convergence (in red triangles below).

   We'll then plot the curve $f(x) = x^3 - 2$, and the derivative $y = 3x^2$ in the same graph.

   See if you can examine the curve and explain the shape you see.



   (Extra for students with coding experience: Reproduce the graph for yourself!)

9.4 Linearization

Before continuing with these algorithms, it will be helpful to review the Taylor series for a function of one variable, and see how it extends to functions of more than one variable.

Recall that the Taylor series for a function $f(x)$, based at a point $x = a$, is given by the following, where we assume that f is analytic:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Therefore, we can *approximate* f using a constant:

$$f(x) \approx f(a)$$

or using a linear approximation (which is the tangent line to f at a):

$$f(x) \approx f(a) + f'(a)(x-a)$$

or using a quadratic approximation:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2$$

We can do something similar if f depends on more than one variable. For example, in Calculus III we look at functions like

$$z = f(x, y)$$

In this case, the linearization of f at $x = a$, $y = b$ is given by the tangent plane:

$$z = f(a, b) + f_x(a, b)(x - a) + f_y(a, b)(y - b)$$

Similarly, if $y = f(x_1, x_2, \ldots, x_n)$, then the tangent plane at $x_1 = a_1, \ldots, x_n = a_n$ is given by:

$$\begin{aligned} z =& f(a_1, a_2, \ldots, a_n) + f_{x_1}(a_1, \ldots, a_n)(x_1 - a_1) + f_{x_2}(a_1, \ldots, a_n)(x_2 - a_2) + \ldots \\ & f_{x_n}(a_1, \ldots, a_n)(x_n - a_n) \end{aligned}$$

If we want to go further with a second order (quadratic) approximation, it looks very similar. First, if $z = f(x, y)$ at (a, b), the quadratic approximation looks like this:

$$\begin{aligned} z = f(a, b) + f_x(a, b)(x - a) + & f_y(a, b)(y - b) + \frac{1}{2}[f_{xx}(a, b)(x - a)^2 + \\ & 2f_{xy}(a, b)(x - a)(y - b) + f_{yy}(a, b)(y - b)^2] \end{aligned}$$

where we assume that $f_{xy}(a, b) = f_{yx}(a, b)$. For functions of n variables, the notation gets a little awkward. We'll define a new structure that will make the notation work a little better.

Gradient and Hessian

Let $y = f(x_1, \ldots, x_n)$ be a real valued function of n variables. To make the notation a bit easier to read, we'll denote the partial derivatives using subscript notation, where

$$\frac{\partial f}{\partial x_i} \doteq f_i \qquad \frac{\partial^2 f}{\partial x_i \partial x_j} \doteq f_{ji}$$

(Does the order of the differentiation matter? In the last equation, could I have written f_{ij}?)

We'll recall from Calculus III that the gradient of f is usually defined as a row vector of first partial derivatives:

$$\nabla f = [f_1, f_2, \ldots, f_n]$$

The $n \times n$ matrix of second partial derivatives is called the **Hessian matrix**, where the (i, j) element is f_{ij}, or

$$Hf = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & & & \vdots \\ f_{n1} & f_{n2} & \cdots & f_{nn} \end{bmatrix}$$

Using this notation, the **linear approximation** to $f(x_1, \ldots, x_n) = f(\boldsymbol{x})$ at the point $\boldsymbol{x} = \boldsymbol{a}$ is:

$$f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

(That last term is a dot product, or a row vector times a column vector) The **quadratic approximation** to f is:

$$f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{a})^T Hf(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

As another example, suppose we want the quadratic approximation to the function

$$w = x \sin(y) + y^2 z + xyz + z$$

Find the quadratic approximation to w at the point $(1, 0, 2)$.

SOLUTION: We need to evaluate f, all the first partials, and all the second partials at the given point:

$$f(1,0,2) = 1 \cdot 0 + 0 \cdot 2 + 0 + 2 = 2$$

Now, using our new notation for partial derivatives:

$$f_1 = \sin(y) + yz \quad \Rightarrow \quad f_1(1,0,2) = 0$$

$$f_2 = x\cos(y) + 2yz + xz \quad \Rightarrow \quad f_2(1,0,2) = 3$$

$$f_3 = y^2 + xy + 1 \quad \Rightarrow \quad f_3(1,0,2) = 1$$

Therefore, $\nabla f(1,0,2) = [0, 3, 1]$. Now computing the Hessian, we get:

$$\left[\begin{array}{ccc} 0 & \cos(y)+z & y \\ \cos(y)+z & -x\sin(y)+2z & 2y+x \\ y & 2y+x & 0 \end{array}\right]\Bigg|_{x=1,y=0,z=2} = \left[\begin{array}{ccc} 0 & 3 & 0 \\ 3 & 4 & 1 \\ 0 & 1 & 0 \end{array}\right]$$

Now we can put these into the formula for the quadratic approximation:

$$2 + [0,3,1]\left[\begin{array}{c} x-1 \\ y \\ z-2 \end{array}\right] + \frac{1}{2}[x-1, y, z-2]\left[\begin{array}{ccc} 0 & 3 & 0 \\ 3 & 4 & 1 \\ 0 & 1 & 0 \end{array}\right]\left[\begin{array}{c} x-1 \\ y \\ z-2 \end{array}\right]$$

And expanded, we could write this as:

$$2 + 3y + (z-2) + \frac{1}{2}\left[6(x-1)y + 4y^2 + 2y(z-2)\right] = \\ -2y + z + 3xy + 2y^2 + yz$$

This is the quadratic approximation to $x\sin(y) + y^2z + xyz + z$ at the point $(1, 0, 2)$.

Linearization, Continued

Suppose we have a general function $\boldsymbol{G}$ that inputs n variables and outputs m variables. In that case, we might write $\boldsymbol{G}$ as:

$$\boldsymbol{G}(\boldsymbol{x}) = \left[\begin{array}{c} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_m(x_1, x_2, \ldots, x_n) \end{array}\right]$$

For example, here's a function that inputs two variables and outputs 3 nonlinear functions:

$$\boldsymbol{G}(x,y) = \left[\begin{array}{c} x^2 + xy + y^2 - 3x + 5 \\ \sin(x) + \cos(y) \\ 1 + 3x - 5y + 2xy \end{array}\right] \tag{9.1}$$

We need a new way of getting the derivative- In this case, it is called **the Jacobian matrix**. If $\boldsymbol{G}$ maps $\mathbb{R}^n$ to $\mathbb{R}^m$, then the Jacobian matrix is $m \times n$, where each row is the gradient of the corresponding function:

$$\boldsymbol{G}(\boldsymbol{x}) = \left[\begin{array}{c} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_m(x_1, x_2, \ldots, x_n) \end{array}\right] \quad \Rightarrow \quad JG = \left[\begin{array}{c} \nabla g_1(x_1, x_2, \ldots, x_n) \\ \nabla g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ \nabla g_m(x_1, x_2, \ldots, x_n) \end{array}\right]$$

Or, written out, the (i, j) element of the Jacobian matrix for $\boldsymbol{G}$ is:

$$(JG)_{ij} = \frac{\partial g_i}{\partial x_j}$$

Continuing with our previous example (Equation 9.1), we'll construct the Jacobian matrix:

$$\boldsymbol{G}(x, y) = \begin{bmatrix} x^2 + xy + y^2 - 3x + 5 \\ \sin(x) + \cos(y) \\ 1 + 3x - 5y + 2xy \end{bmatrix} \quad \Rightarrow \quad JG = \begin{bmatrix} 2x + y - 3 & x + 2y \\ \cos(x) & -\sin(y) \\ 3 + 2y & -5 + 2x \end{bmatrix}$$

As a second example, suppose that we have a function f that maps $\mathbb{R}^n$ to $\mathbb{R}$. Then we could let

$$\boldsymbol{G}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$$

Note that we have to think of the gradient as a column vector instead of a row, and the ith row is:

$$f_{x_i}(x_1, x_2, \ldots, x_n)$$

so that $\boldsymbol{G}$ maps $\mathbb{R}^n$ to $\mathbb{R}^n$. Furthermore, in that case, the **Jacobian of $\boldsymbol{G}$** is the **Hessian of f**:

$$(JG)_{ij} = \frac{\partial f_{x_i}}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}\right) = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

(We're using Clairaut's Theorem to simplify our expression).

Now we get to the **linearization** of the function $\boldsymbol{G}$ at a point $\boldsymbol{x} = \boldsymbol{a}$:

$$\boldsymbol{G}(\boldsymbol{a}) + JG(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

To illustrate this notation, let's linearize Equation 9.1 at the origin $(0, 0)$.

$$\boldsymbol{G}(0, 0) = \begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix} \qquad JG(0, 0) = \begin{bmatrix} -3 & 0 \\ 1 & 0 \\ 3 & -5 \end{bmatrix}$$

Therefore, the linearization at the origin is:

$$\begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -3 & 0 \\ 1 & 0 \\ 3 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 - 3x \\ 1 + x \\ 1 + 3x - 5y \end{bmatrix}$$

You might notice that each row is the linearization of the corresponding function from $\boldsymbol{G}$- For example, $1 + x$ is the linearization of $\sin(x) + \cos(y)$ at the point $(0, 0)$.

Now we can get back to Newton's Method.

Multivariate Newton's Method

In the general case, we are solving for the critical points of some function f,

$$\nabla f(\boldsymbol{x}) = \boldsymbol{0}$$

That is, if f depends on n variables, then this is a system of n equations (in n unknowns).

As we did in the previous section, we can think of the gradient itself as a function:

$$\boldsymbol{G}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$$

and think about how Newton's Method will apply in this case.

Recall that in Newton's Method in one dimension, we begin with a guess x_0. We then solve for where the *linearization* of f crosses the $x-$axis:

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

From which we get the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Now we do the same thing with the vector-valued function $\boldsymbol{G}$, where $\boldsymbol{G} : \mathbb{R}^n \to \mathbb{R}^n$. Start with an initial guess, $\boldsymbol{x}_0$, then we linearize $\boldsymbol{G}$ and solve the following for $\boldsymbol{x}$:

$$\boldsymbol{G}(\boldsymbol{x}_0) + JG(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0) = 0$$

Remember that JG, the Jacobian of $\boldsymbol{G}$, is now an $n \times n$ matrix, so this is a system of n equations in n unknowns. Using the recursive notation, and solving for $\boldsymbol{x}$, we now get the formula for **multivariate Newton's Method**:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - JG^{-1}(\boldsymbol{x}_i)G(\boldsymbol{x}_i)$$

We should notice the beautiful way that method generalizes to multiple dimensions- The reciprocal $1/f'(x_i)$ becomes the matrix inverse: $JG^{-1}(\boldsymbol{x}_0)$.

9.5 Nonlinear Optimization with Newton

Now we go back to our original question: We want to optimize a function whose domain is multidimensional:

$$\min_{\mathbf{x}} \ f(\mathbf{x}) \qquad \text{or} \qquad \max_{\mathbf{x}} \ f(\mathbf{x})$$

where it is assumed that $f : \mathbb{R}^n \to \mathbb{R}$. To use Newton's Method, we let $\boldsymbol{G} = \nabla f$ for the formulas in the previous section. Here's a summary:

Multivariate Newton's Method to Solve $\nabla f(\mathbf{x}) = 0$

Given an initial $\mathbf{x}_0$, compute a sequence of better approximations to the solution:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - Hf^{-1}(\boldsymbol{x}_i)\nabla f(\boldsymbol{x}_i)$$

where Hf is the Hessian matrix ($n \times n$ matrix of the second partial derivatives of f).

Example: Minimize $f(x, y) = \frac{1}{4}x^4 - \frac{1}{2}x^2 + \frac{1}{2}y^2$.

SOLUTION: First, we can use Calculus to check the computer output. The gradient is:

$$\nabla f = [x^3 - x, \quad y]$$

so that the critical points are $(-1, 0), (1, 0)$ and $(0, 0)$.

9.5.1 Issue: Local Extrema

As in calculus, once we find the critical points, we need to determine if they represent a local max, a local min, or something else (like a saddle).

We may recall the "second derivatives test" from Calculus III- It actually uses the Hessian:

The Second Derivatives Test for $z = f(x,y)$ at critical point (a,b)

Let $D = f_{xx}(a,b)f_{yy}(a,b) - f_{xy}^2(a,b)$. If

- If $D > 0$ and $f_{xx}(a,b) > 0$, then $f(a,b)$ is a local min.
- If $D > 0$ and $f_{xx}(a,b) < 0$, then $f(a,b)$ is a local max.
- If $D < 0$, then $f(a,b)$ is a saddle point.

Returning to our example, the Hessian of f:

$$Hf = \begin{bmatrix} f_{xx}(x,y) & f_{xy}(x,y) \\ f_{yx}(x,y) & f_{yy}(x,y) \end{bmatrix} = \begin{bmatrix} 3x^2 - 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Therefore, the second derivatives test is a test using the determinant of the Hessian at the critical point. In this example, at $(\pm 1, 0)$, the determinant is positive and $f_{xx}(\pm 1, 0)$ is positive. Therefore, $f(\pm 1, 0) = 1/4$ is the local minimum. We have a saddle point at the origin.

9.5.2 Issue: Matrix Inversion

Since the multivariate Newton's method requires us to invert an $n \times n$ matrix, we need to be careful. In fact, many algorithms will watch the eigenvalues of the Hessian to make sure that the matrix doesn't get too close to being non-invertible. One way to do this is to track what is called **the condition number** of the matrix. The condition number of the matrix is defined as:

$$k = \frac{|\lambda_{\max}|}{|\lambda_{\min}|}.$$

If the matrix is not invertible, or very close to being not invertible, then $\lambda_{\min} \approx 0$, which makes the condition number blow up to a very large number.

One way we may avoid this problem is through the use of the pseudo-inverse, but one need to be careful about when to employ this technique.

Multivariate Newton's Method, in Matlab and Python

These will be provided in a separate handout.

9.6 Gradient Descent

The method of gradient descent is an algorithm that will search for local extrema by taking advantage of the fact about gradients:

The gradient of a function at a given point will point in the direction of fastest increase (of the function).

Therefore, if we started at a point (or vector) $\boldsymbol{a}$, if we want to find a local maximum, we should move in the direction of the gradient. To locate a local minimum, we should move in the direction negative to the gradient.

Therefore, the method of gradient descent proceeds as follows: Given a starting point $\boldsymbol{a}$ and a scalar α (which is referred to as the step size), we iterate the following:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \alpha \nabla f(\boldsymbol{x}_i)$$

The method is fairly quick, but can have bad convergence properties- Mainly because we have that step size. There is a way to get "best" step size α. In the following, we'll continue with finding the local **minimum**.

Once we choose a direction of travel, the idea will be to follow that line until we find the minimum of the function (along the line). That is, we have a small optimization problem. Find the value of t that minimizes the expression:

$$f(\boldsymbol{a} - t\nabla f(\boldsymbol{a}))$$

This is called a *line search*, and can be time consuming. Before continuing, let's look at this more closely. Define a new function $\phi(t)$ as the output along the line:

$$\phi(t) = f(\boldsymbol{a} - t\boldsymbol{u}) = f\begin{pmatrix} a_1 - tu_1 \\ a_2 - tu_2 \\ \vdots \\ a_n - tu_n \end{pmatrix}$$

where $\boldsymbol{u}$ is a vector and f depends on $x_1, \ldots, x_n$.

Then optimizing this means to find where the derivative of ϕ is zero. Let's compute the derivative using the chain rule:

$$\phi'(t) = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \cdots + \frac{\partial f}{\partial x_n}\frac{\partial x_n}{\partial t} = \frac{\partial f}{\partial x_1}u_1 + \cdots + \frac{\partial f}{\partial x_n}u_n = -\nabla f(\boldsymbol{a} - t\boldsymbol{u}) \cdot \boldsymbol{u}$$

(Note the minus sign from differentiating $-t\boldsymbol{u}$). Thus, setting this to zero and solving should give us an optimal step size.

Example: By hand, compute one step of gradient descent to the following (with optimal step size).

$$f(x, y) = 4x^2 - 4xy + 2y^2 \qquad (x_0, y_0) = (2, 3).$$

SOLUTION:

Step 1: Compute the gradient and evaluate it at $(2, 3)$:

$$\nabla f = [8x - 4y, -4x + 4y] \quad \Rightarrow \quad \nabla f(2, 3) = [4, 4]$$

Step 2A: Find the parametric equations of the line along which we will travel (h is the stepsize to be found later):

$$\boldsymbol{x}_0 - h\nabla f(\boldsymbol{x}_0) = \begin{bmatrix} 2 \\ 3 \end{bmatrix} - h\begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 - 4h \\ 3 - 4h \end{bmatrix}$$

Step 2B: Apply f to the path found in part $2A$. This is the expression we want to minimize.

$$\phi(h) = f(\boldsymbol{x}_0 - h\nabla f(\boldsymbol{x}_0) = f(2 - 4h, 3 - 4h)$$

We could write out this expression, but it is not needed at the moment.

Step 2C: To find the optimal path, we define $\phi(h)$ to be the function restricted to the line:

$$\phi(h) = f(2 - 4h, 3 - 4h)$$

We'll need to evalate the gradient along the line, so let's go ahead and compute that:

$$f_x(2 - 4h, 3 - 4h) = 8(2 - 4h) - 4(3 - 4h) = 16 - 32h + 12 + 16h = 4 - 16h$$

$$f_y(2 - 4h, 3 - 4h) = -4(2 - 4h) + 4(3 - 4h) = -8 + 16h + 12 - 16h = 4$$

Step 3: Take the derivative of ϕ, set it equal to zero, and solve for h.

$$\phi'(h) = -\nabla f(2-4h, 3-4h) \cdot \nabla f(2,3) = -[4-16h,\ 4]\begin{bmatrix} 4 \\ 4 \end{bmatrix} = -32 + 64h$$

Therefore, the derivative is zero for $h = 1/2$, and the second derivative is positive ($\phi''(h) = 64$), so this is indeed where the minimum occurs.

Step 4: Now, using that step size to advance to the next point in the domain.

$$\boldsymbol{x}_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

And now we would repeat our steps until we find the minimum, or some other stopping criteria is met.

9.6.1 Backtracking line search

Rather than a regular line search for the minimum, we might estimate the step size at each iteration using what's called a backtracking line search. We'll start with a larger step-size, and at each iteration, we'll check the inequality:

$$f(\mathbf{x}_t - \alpha_t \nabla f(\mathbf{x}_t) \leq f(\mathbf{x}_t - \frac{\alpha_t}{2}\|\nabla f(\mathbf{x}_t)\|^2$$

If the inequality is verified, the current step size is accepted. If not, we divide it by 2 and check again.

9.6.2 Gradient Descent with Data

Going back to our introduction in Section 9.1, we said that typically, we'll have some data and a model function with some parameters, $F(x; \mathbf{w})$. For example, in the line of best fit problem, the proposed form for the line was: $y = mx + b$, or $F(x; m, b) = mx + b$, where we needed to determine the model parameters m and b. Given data points (x_i, t_i) where t is for targets, we want to minimize the error between the desired targets t_k and the model output $y_k = F(x_k)$:

$$E(m,b) = (t_1 - y_1)^2 + (t_2 - y_2)^2 + \cdots + (t_p - y_p)^2 = \sum_{k=1}^{p}(t_k - y_k)^2$$

If we compute the gradient, we'll need $\partial E/\partial m$ and $\partial E/\partial b$:

$$\frac{\partial E}{\partial m} = 2\sum_{k=1}^{p}(t_k - y_k)\left(-\frac{\partial y_k}{\partial m}\right) = 2\sum_{k=1}^{p}(t_k - y_k)\,(-x_k) \qquad \frac{\partial E}{\partial b} = 2\sum_{k=1}^{p}(t_k - y_k)\left(-\frac{\partial y_k}{\partial b}\right) = 2\sum_{k=1}^{p}(t_k - y_k)(-1)$$

Finally, our goal: Update parameters m and b to make the error decrease (be sure you're moving in the direction of the *negative* gradient!).

$$\begin{bmatrix} m_{\text{new}} \\ b_{\text{new}} \end{bmatrix} = \begin{bmatrix} m_{\text{old}} \\ b_{\text{old}} \end{bmatrix} - \alpha \nabla E(m,b)$$

For $y = mx + b$, we could actually solve the problem exactly by setting the partial derivatives to zero, however, in the more general setting, we see that, if w_i is a parameter for the model, then

$$\frac{\partial E}{\partial w_i} = 2\sum_{k=1}^{p}(t_k - y_k)\left(-\frac{\partial y_k}{\partial w_i}\right)$$

and the update rule is:

$$w_i^{\text{new}} = w_i^{\text{old}} - \alpha\frac{\partial E}{\partial w_i}$$

To do the full gradient descent, we see that we are required to take a sum over all the data, and that's for each training parameter (and for each step we take). To understand the implications of this, consider a small training problem (one that we'll actually work with later). Suppose our data input consists of 60,000 images (to be specific, let's say 28×28 grayscale), so that the domain is in $\mathbb{R}^{784}$, with a scalar output (for example, classifying small images would be such a case). The model we will build later will have $101,770$ parameters. Practically speaking, it would take a very very long time to find an approximation to a local minimum for the error. We need a faster approach (that will be the topic of the next section).

9.7 Exercises

1. Suppose we have a function f so that $f(0,1) = 3$ and $\nabla f((0,1)) = [1, -2]$. By hand, use the linearization of f to estimate $f(1/2, 3/2)$.

2. Let $f(x, y) = x^2 y + 3y$. By hand, compute the linearization of f at $x = 1, y = 1$.

3. Let $f(x, y) = xy + y^2$. At the point $(2, 1)$, in which direction is f increasing the fastest? How fast is it changing? (These are quickly done without a computer).

4. By hand, use the second derivatives test to classify the critical points of the following functions:

   (a) $f(x, y) = x^2 + xy + y^2 + y$

   (b) $f(x, y) = xy(1 - x - y)$

5. Use Newton's Method (on the computer) to find two critical points of f correct to three decimal places.

   $$f(x, y) = x^4 + y^4 - 4x^2 y + 2y$$

   Hint: There are three of them in $-1 \le x \le 1, -1 \le y \le 1$, so try several starting points.

6. By hand, perform the computations involved in the second step of the gradient descent example (with $f(x, y) = 4x^2 - 4xy + 2y^2$).

7. We said that, in order to optimize the step size, we set the derivative of ϕ to zero:

   $$\phi'(t) = \nabla f(\boldsymbol{a} - t\boldsymbol{u}) \cdot \boldsymbol{u} = 0$$

   Show that this implies the following theorem about the method of gradient descent:

   $$\nabla f(\boldsymbol{x}_{i+1}) \cdot \nabla f(\boldsymbol{x}_i) = 0$$

   or, the gradient vectors from one step to the next are orthogonal to each other.

9.8 Stochastic Gradient Descent (SGD)

At the end of the section on Gradient Descent, we discussed its primary drawback. If you have a large-scale optimization problem, it is practically impossible to use regular gradient descent. However, in recent years researchers have had great success in using an *approximation* to the gradient- That method is Stochastic Gradient Descent (SGD).

Interestingly, the method first appeared by Herbert Robbins and Sutton Monroe, back in 1951. What has made it popular recently is the appearance of very large scale problems, starting approximately 2005 with the invention of the convolution neural network, and developing into what we now call "deep learning".

The idea is actually very easy; the fact that the algorithm still converges is more complicated. The idea is that, instead of using all of the data to estimate the gradient, we use only one point. In terms of our previous example before the exercises, recall that

$$\frac{\partial E}{\partial m} = 2\sum_{k=1}^{p}(t_k - y_k)\left(-\frac{\partial y_k}{\partial m}\right) = 2\sum_{k=1}^{p}(t_k - y_k)\,(-x_k) \qquad \frac{\partial E}{\partial b} = 2\sum_{k=1}^{p}(t_k - y_k)\left(-\frac{\partial y_k}{\partial b}\right) = 2\sum_{k=1}^{p}(t_k - y_k)(-1)$$

In SGD, we estimate the partial derivatives. Below, r is an index chosen at random (uniformly):

$$\frac{\partial E}{\partial m} \approx 2(t_r - y_r)\left(-\frac{\partial y_r}{\partial m}\right) = 2(t_r - y_r)\,(-x_r) \qquad \frac{\partial E}{\partial b} \approx 2(t_r - y_r)\left(-\frac{\partial y_r}{\partial b}\right) = 2(t_r - y_r)(-1)$$

Interestingly, with this change, we cannot expect strict convergence of the error to its minimum. Here are some samples from the following website (go there and try these out!):

`fa.bianp.net/teaching/2018/eecs227at/stochastic_gradient.html`

Just to clarify the web page's notation- Suppose that we have 100 data points in $\mathbb{R}^2$, (x_i, t_i) and we want to find a line of best fit, $y = mx + b$. Then the usual error function is given by

$$E(m,b) = (t_1 - y_1)^2 + (t_2 - y_2)^2 + (t_3 - y_3)^2 + \cdots + (t_{100} - y_{100})^2 = \sum_{i=1}^{100}(t_i - y_i)^2$$

or, in terms of m, b:

$$E(m,b) = (t_1 - (mx_1 + b))^2 + \cdots + (t_{100} - (mx_{100} + b))^2 = \sum_{i=1}^{100}(t_i - (mx_i + b))^2$$

On the website, they say: We want to minimize $F(\mathbf{x})$, where the error function is given by

$$\sum_{i=1}^{n} f_i(\mathbf{x})$$

To make this clear, the $\mathbf{x}$ being referred to there is our vector (m, b), and

$$f_i(\mathbf{x}) \Rightarrow (t_i - y_i)^2 \text{ or } (t_i - (mx_i + b))^2$$

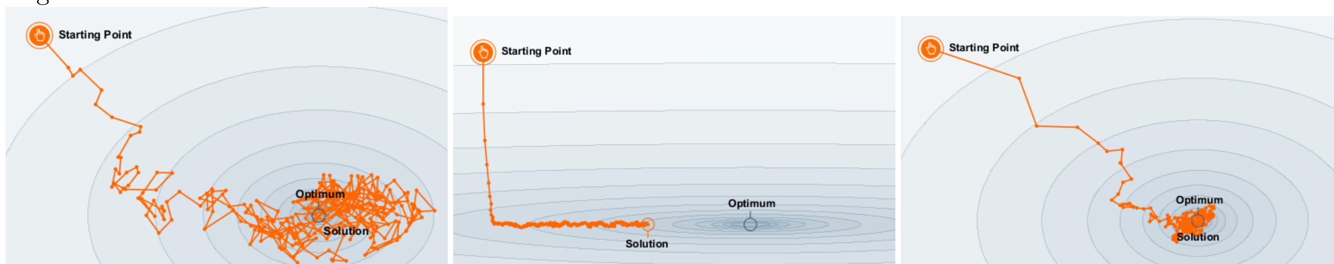
Now, in place of the full gradient, we're **estimating** the gradient using a single point. Suppose we're on some iteration, and we choose data point 10. Then we estimate the error function using the 10th data point (on this iteration):

$$E(m,b) \approx (t_{10} - y_{10})^2 \text{ or } (t_{10} - (mx_{10} + b))^2$$

so that the partial derivatives are evaluated as:

$$\frac{\partial E}{\partial m} = 2(t_{10} - (mx_{10} + b))(-x_{10}) \qquad \frac{\partial E}{\partial b} = 2(t_{10} - (mx_{10} + b))(-1)$$

There are three images below- One with the learning rate too high, one with it too low, and the other just right.



Rather than a strict convergence to the optimum, we see what is called **convergence to a noise ball**, meaning that with a constant step size, we'll keep bouncing around the optimum value.

We'll experiment a little with the parameters to get a better understanding of what's happening.

9.8.1 Computer Examples

Suppose we have some data representing the square footage of certain houses, together with their prices. Suppose we want to construct a function that will predict the house price based on the square footage.

Such a situation might call for a "line of best fit", but in general, we're given a model function, $y = mx + b$, with two training parameters. We define our error in the usual way as the sum of squares error.

In the first script, `HousingPrices.m`, we load the data in, we scale the footage values and the price values so that the have zero mean and unit standard deviation, and then we find the best m, b using gradient descent with a fixed step size.

In the second script, `HousingPricesAdaptedStep.m`, we adjust the step size using the backtracking line search, and we find extremely fast convergence.

In the third script, `HousingPricesSGD.m`, we adjust our training parameters using the stochastic gradient descent. What's very interesting here is that the error does approach some local minimum value (jumps around a bit), but the gradient does NOT converge to zero- it actually bounces around quite a bit.

Why would we use Stochastic Gradient Descent? When we do larger problems with many, many data points, it would take a very long time to train if we had to sum through every single data point.

An adjustable parameter

There is another alternative to the Stochastic Gradient Descent algorithm itself that you may have thought of. In the original gradient descent, we used all the data. In the SGD, we use only one data point. How about if we use a small sample of data points? That is in fact the approach called **Batch SGD**. Of course, then there is a question of how many points should be in the batch- The answer is most likely problem dependent. How much time are you willing to put into estimating the gradient? The choices we can make run from computing the gradient exactly but very slowly with all the data, to going extremely quickly but with lots of error in the gradient using a single data point.

Once again we're reminded that running these algorithms is more of an art than a science. With practice, you get a sense for what values work and which don't.

9.9 Homework

1. Suppse we have four points below, and we're building a line of best fit: $y = mx + b$.

x	-1	1	2	3
t	0	1	3	2

   (a) Write down the sum of squares error function, and the partial derivatives of the error with respect to m and b.

   (b) Use one step of gradient descent if we begin at the point $m = 0, b = -1$ with step size 0.1. Use a calculator, but not a computer- The point of this exercise is for you to step through all the calculations.

   (c) Use one step of stochastic gradient descent if we begin at $m = 0, b = -1$ and use data point 3 (same step size, 0.1) for the gradient estimate.

2. There is actual housing data from Walla Walla in the file HousingWW.m. This is a script file, so to load the data, you would just type the name of the file (in your script or on the command line): HousingWW. Re-run the scripts with this new data.

   (a) How many steps does it take each algorithm to converge or stop?

   (b) What is the final error value (using any of the scripts)

   (c) Given your m, b, predict the price for a house with 2,000 square feet.

Chapter 10

k-Nearest Neighbors

We might observe that we use experience in order to predict outcomes. It is the accumulation of experiences that give us the ability to become more nuanced in those predictions, and if a given situation is completely outisde of that set of accumulated experience, then we will probably not be able to anticipate what will happen.

It is this basic observation that lies at the heart of the **nearest neighbor classifier**. Let's set this up properly: Suppose we have p points in $\mathbb{R}^n$ (stored in a matrix X) that have class labels stored in a target vector $\boldsymbol{t}$ (so $\boldsymbol{t}$ might be $p \times 1$). The problem is this: Given a new point $\hat{\boldsymbol{x}}$, I want to determine the most appropriate class label for this point. The nearest neighbor rule is the following:

Nearest Neighbor Rule

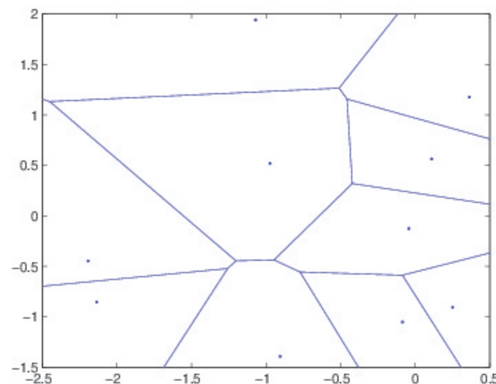
Given data in matrix X and class labels in $\boldsymbol{t}$, the class label for a new point $\hat{\boldsymbol{x}}$ is given by the class label of the point in X to which it is closest. That is, if i^* is the index of the point in X closest to $\hat{\boldsymbol{x}}$:

$$i^* = \min_i \|X(:, i) - \hat{\boldsymbol{x}}\|$$

then the target label for the new point is $\boldsymbol{t}_i$.

As a side note, this is a **supervised learning** task, as opposed to data clustering that we looked at earlier, which was **unsupervised** because we did not have the class label.

The nearest neighbor rule is about as straightforward a rule as they come. Think about what the set of boundaries between classes would look like, and you might visualize something we've seen before:



Yes- the decision boundaries for the nearest neighbor rule forms Voronoi cells, with the data points from X at each center.

10.0.1 Implementing the Nearest Neighbor Rule

The main computation will be the set of distances from the new point $\hat{\boldsymbol{x}}$ to every column of the matrix X. In fact, we really don't need the actual distances, rather the distances squared would work just as well and would save us from having to do the extra square root.

Rather than creating a loop through the data, it is usually easier and faster to "vectorize" this computation. In Matlab, this would look like:

```
temp=X-hat_x;                   % Subtract hat_x from each column of X.
distances=sum( temp.*temp );    % Sum (down) the square of temp to get a row
                                %   of squared distances
[vals,idx]=sort(distances);     % Sort the distances in ascending order
ClassLabel=t(idx(1));           % The desired label is the label whose index
                                %   is first in idx, found in target vector t.
```

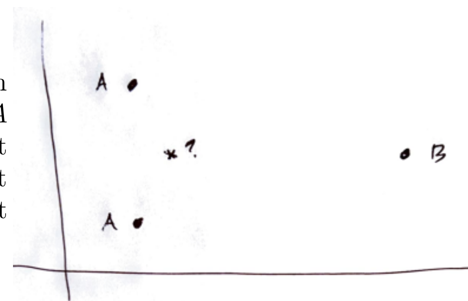
In Python, we'll have something similar.

```
import numpy as np

temp=X-hat_x
distances=np.sum(np.square(temp),axis=0)
idx=np.argsort(distances)
ClassLabel=t[idx[0]]
```

Example

To the right is an example. We have three points in the plane for the matrix x. Two have class label A and one has class label B. There is an unknown point also marked. Which class label should that have? It would seem reasonable to assign that to class A. But wait!



Here is my actual data:

$$X = \begin{bmatrix} 0.1 & 0.1 & 0.2 \\ 1000 & 2000 & 1500 \end{bmatrix} \qquad \hat{\boldsymbol{x}} = \begin{bmatrix} 0.11 \\ 1500 \end{bmatrix} \qquad \texttt{distances} = [250,000 \quad 250,000 \quad 0.008]$$

$$\mathbf{t} = [\ A \quad A \quad B\ ]$$

We see that the image is very misleading- the scales on the x, y axes are completely different, so that the distance on the x axis is almost completely swamped out by the scale on the y.

This can happen frequently with "mis-scaled" data, especially when we're relying on the Euclidean distance metric. We have two ways to fix this, if it is an issue:

- Rescale the data: The typical fix is to mean subtract and divide by the standard deviation in the x and y coordinates independently, so that both have approximately the same scale.

- Change the metric: For example, we can weight the distances:

$$\|\alpha_1(\Delta x)^2 + \alpha_2(\Delta y)^2\|$$

where α_1, α_2 are the weights- we could make α_1 much larger than α_2 in order to correct the imbalanced scale.

The main point here: Pay attention to your data. Be sure you know what kinds of scales you're working with, and re-scale the data if necessary.

10.0.2 K-nearest Neighbor Classifiers

In the $k-$nearest neighbors algorithm, we look at the $k-$nearest points in X to the new point $\hat{\boldsymbol{x}}$. To decide on a class label, it is common to use "majority rule" voting, so for example, if we had 5 nearest neighbors, and 3 of them were class A and 2 were class B, we would choose class B. The only change to our previous code is to count the class labels and choose the one with the max.

You might also want to allow the closest points to have more weight in the calculation than points farther away. In that case, instead of adding a count of 1 for each neighbor in a certain class, you would add $1/d$. Then the class with the higher score wins.

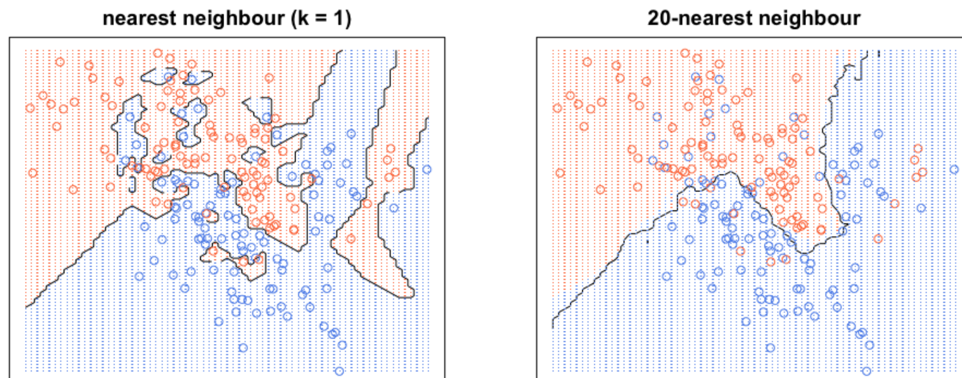
Extreme Cases

If $k = 1$, the function will simply find the data point it is closest to, and give that output. In particular, if we classify data used in the model, then we would have 100% classification, as the model will have memorized the data. If k is large enough to always capture all the data, then the classification will be constant (whichever class is most prominent in the training data).

Presumably, there is an ideal number of nearest neighbors that we should use- but how should that number be selected? One method- *Cross Validation* will be described shortly. First, given a classifier, how accurate is it?

Is a smaller k better than a large k?

For small values of k, we run the risk of modeling "noise" in the data. The boundaries between classes can be unstable and complicated (but "accurate" because we are overfitting). Large values of k will smooth out the borders between classes making it easier for us to generalize. See the image below, where the image to the left is with $k = 1$ (very accurate, but does not generalize well) and the image to the right is $k = 20$ (less accurate, but good generalization).



I've seen some rules of thumb that say we should use about $\sqrt{N}$ as the value of k, where N is the number of sample points, but depending on the problem, this could be unreasonably large or small, so this tends to be problem dependent. Later in this section, we'll see how to use a technique (cross validation) to help us determine a good value of k.

10.1 Accuracy: Testing, Training and Validation Sets

As we mentioned in the previous section, it is possible to get 100% accuracy if we choose $k = 1$ and only look at data that was used to build the model. So if we're always 100% accurate, then there's really no point in analyzing why- the data has been memorized. In this case, we would say that the model has been **overfit-** That is, we're modeling everything about the data and not trying to generalize.

On the other extreme where every data point is included in the neighbors, then the prediction is constant- this is an extreme case of *underfitting*, where we only have the general trend modeled.

We try to fix the overfitting, underfitting issue by considering the following: The "error" needs to be measured using data that was NOT used for training. Before training, we need to reserve some data for that purpose.

Some definitions before continuing:

- The **training set** is the set of data we've reserved for building the model.
- The **validation set** is the set of data we've reserved for checking model parameters (if needed).
- The **testing set** is reserved to measure the error at the very end of training. It must be data that the model has not used for training or testing. The golden rule of machine learning is to **never** use training data for measuring accuracy!

Now we use our training set to build the model parameters. Then you use the validation set to measure the error- We hope that the validation set will tell us how well our model is generalizing to "new" data. As our parameters are being tuned (for example, the number of neighbors), we will expect that the error using validation data will decrease until we've reached some optimal value, at which point the error will begin to increase again. It only now that the third set, the testing set, can be used to measure the "true" error.

Now that we've described how to use the three data sets, let's talk about some practicalities. We will probably not have enough data to keep producing new training sets as we tune the model parameters. If you're a car manufacturer, for example, those data points may be costing you tens of thousands of dollars each. We'll need a different idea for model tuning- That is *Cross Validation*.

10.1.1 k-fold Cross Validation for Tuning Your Model

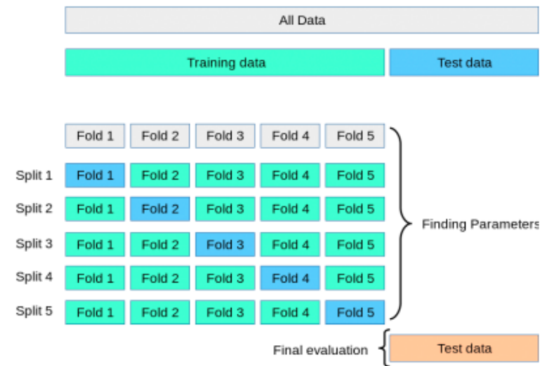
To begin with, suppose we have split our data into a **training set** and a **test set**. A common percentage might be 70-30, although this is problem dependent.

Is there anything wrong with doing the following?

- Use the training set with some set of model parameters (like the number of neighbors),
- Determine the error using the test set.
- Repeat this process while changing the model parameters over some finite set of values (but not changing the training or test set).

The biggest problem with this method is that we used our test set to determine the model parameters, so now we have no reserved data that we can use to measure the "true" error. Secondly, since we're choosing the data split randomly, it might happen that we have chosen a bad (nonrepresentative) sample- In that case, we have no remedy while we're determining the parameters.

What can be done instead? In $k-$fold cross validation, we'll split the training set into k distinct sets (called folds), and we'll be training k times. Each time we train, we will designate one of the k sets to be the validation set, and the other $k-1$ sets will be used to train the model. Afterwards, we determine the error as the average over the k training sessions. As an example of how to split the data, here is a "cartoon" showing $5-$fold cross validation (as a general rule of thumb, researchers tend to use $k = 5$ and $k = 10$ if they have enough data).



As an example for using CV for determining the best number of nearest neighbors, we first split the data into training and testing sets, and reserve the test set. We now split the training data into k folds, and with some fixed value of k (nearest neighbors), we run the k training sessions and get the error estimate. Now we increase k by one and repeat. And repeat. And repeat- At the end, we'll choose the number of nearest neighbors that gave us the best average error. Once that is done, we will re-train the algorithm with that number of neighbors, and use the reserved test set for the final error computation.

We will mention here as well- this kind of process can also tell you what kind of algorithm to run on your data. You may be selecting from several kinds of classifiers, for example. Running $k-$fold cross validation is a good way of measuring the error from each classifier, then you would select the algorithm with the best average error. Finally, you would train the classifier on the training set and use your reserved test set for the error computation.

We won't go too much further into cross validation right now, but Python does have versions of cross validation available in `GridSearchCV` and `RandomizedSearchCV` to determine the best values of the designated parameters. For more information, see the `scikit-learn` documentation.

10.2 The Confusion Matrix

In the previous section, we learned how to get a good measure of our error. In classification problems, we typically want to see more than the overall average error. Rather, it can be very useful to see what kinds of errors are being made. For example, if we are misclassifying label A, is it because we're labeling it only as B, or sometimes C as well?

To answer these questions, we set up an array called a **confusion matrix**. Here's a small example, where we have some algorithm classifying on object as a *dog, cat, or rabbit.*

	Actual Dog	Actual Cat	Actual Rabbit
Classified Dog	23	12	7
Classified Cat	11	29	13
Classified Rabbit	4	10	24

Predicted classes are along the vertical axis of the matrix, actual classes are along the horizontal, although this "rule" tends to not be followed very strictly, so do pay attention to your axis labels.

Along the diagonal entries are where the predicted result is equal to the actual result (which are the correct classifications), and the off-diagonal elements represent classification error. For example, our set had 11 objects that were dogs but classified as cats.

Although missing from our example, some confusion matrices will also provide the row and column sums, which can be helpful. To find the total number of objects being classified, we sum all the cells together. In this example, we have 133 objects being classified, and 74 of those were classified correctly, giving us an overall accuracy of 57%. We can also look at some subscores, for example, looking only at dogs, we had 38 total (sum down the column), and we correctly classified 23, giving an accuracy of about 61%. For cats, we had 51 objects total and classified 29 correctly, giving 57% accuracy (and so on).

Now that we have some ideas about how to visualize our error, we have one more topic to cover.

10.2.1 K-nearest Neighbor Regression

So far we have discussed only the clustering (or classification) problem. As it turns out, the algorithm can, with very few changes, be made to model a function as well.

In this case, our data points are still in X, but the function output is in the target vector $\boldsymbol{t}$. To find the function output for a new point $\hat{\boldsymbol{x}}$, we would still find the $k-$nearest neighbors in X, determine their output values from $\boldsymbol{t}$, and combine these values in some way to produce our estimate for $\hat{\boldsymbol{x}}$. There are multiple ways of doing this- for example, the new output is the average of the k given outputs.

The problem with this is it gives equal weight to points very close to $\hat{\boldsymbol{x}}$ and points that may be far away (even though it is within the 5 closest points). Therefore, we might provide a weighted average. Since the distances are all non-negative, we can form weights α_i for the k points as the following, where d_i is the distance to the ith point:

$$\alpha_i = \frac{1/d_i}{\sum_{j=1}^{k}(1/d_j)}$$

You'll notice that $0 \leq \alpha_i \leq 1$ for each i, and $\sum_{i=1}^{k}\alpha_i = 1$, like a set of probabilities. Then the output of function at $\hat{\boldsymbol{x}}$ is given by:

$$f(\hat{\boldsymbol{x}}) = \sum_{i=1}^{k}\alpha_i t_i$$

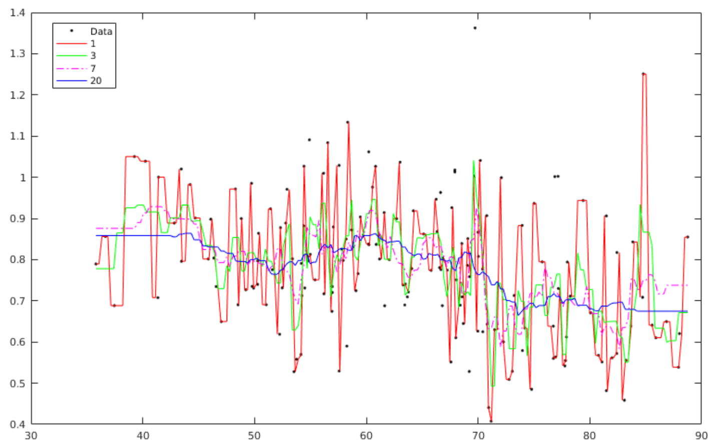
This technique could be extended to functions of more than one variable. For example, if the known target values are vectors $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p$, then we can compute the output just as before:

$$f(\hat{\boldsymbol{x}}) = \sum_{i=1}^{k}\alpha_i \boldsymbol{t}_i$$

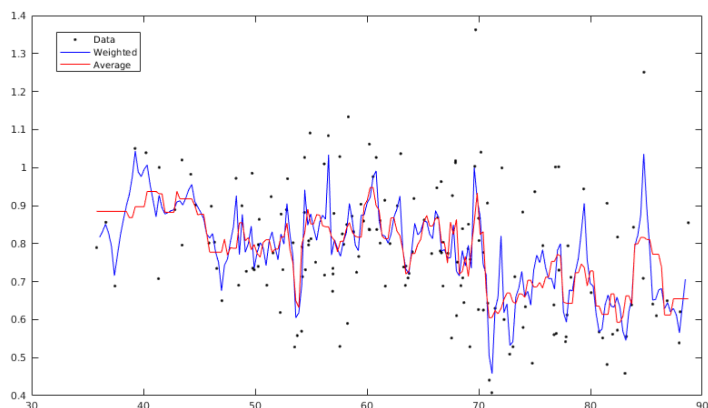
This kind of a linear combination (with non-negative weights that sum to 1) has a special name- This is called a **convex combination** of the vectors $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p$ (where the weights for the targets outside of k points are set to zero).

10.2.2 Example

Here is an example of the output of the one-dimensional regression. We would like to predict Bone Mineral Density (BMD) based on the age of a patient. The dots in the image below represent the raw data. In the first graph below, we show the result of using an increasing number of nearest neighbors (with a straight average output). It is clear that the larger the number of neighbors, the smoother the function becomes. With $k = 1$ (the red curve) we are jumping all over the graph, but the blue curve (20 nearest neighbors), we have smoothed the function by quite a bit.



Further, below we show the difference in using the standard average as the output versus the weighted average (weighted as in the text). The weighted average in this example actually shows a bit more variation.



We'll take a look at this example more closely in the homework exercises.

10.3 Conclusions

In this section, we discussed the $k-$nearest neighbor classifier and regression algorithm. But just as importantly, we're starting to talk about data. In particular:

1. Given a novel data set, be sure and examine it. You're looking for scaling issues, but there could be missing data or duplicate data as well. We'll be talking about **preprocessing** more later- In today's work, we saw the importance of scaling the variables.

2. When we engage in model building, it is critically important that we reserve some data for estimation of the true model error, so we initially divide the data (both input and target) into a training set and a test set. A rule of thumb is approximately 70-30.

3. The training set can then be further subdivided, into training and validation sets. This can be done using a single partition, or we can prep $k-$fold cross validation and separate into k folds.

4. The output of the classifier can be visualized using a **confusion matrix**.

10.4 Homework

1. It has been said that $k-$nearest neighbors should not be used for high dimensional input- Let's see why. Suppose our data is in the unit hypercube,

$$0 \leq x_i \leq 1, \qquad i = 1, 2, 3, \ldots, n$$

So in dimension 1, we're on the interval $[0, 1]$. In dimension two, we have a square $[0, 1] \times [0, 1]$ or $[0.1]^2$. In dimension 3, the domain is $[0, 1]^3$ and so on. Further, let's set the number of data points in each problem to 1000.

- In dimension 1, find the number of points so that $0 \leq x \leq 1/2$.
- In dimension 2, find the number of $\mathbf{x}$ such that $0 \leq x_i \leq 1/2$ for $i = 1, 2$.
- In dimension 3, find the number of $\mathbf{x}$ such that $0 \leq x_i \leq 1/2$ for $i = 1, 2, 3$.
- In dimension 4, find the number of $\mathbf{x}$ such that $0 \leq x_i \leq 1/2$ for $i = 1, 2, 3$ and 4.

As an example in Matlab for two dimensions, we would have:

```
N=1000;
dim=2;

X=rand(1000,dim); %Data
count=0;          %keep track of the data inside nghbrhd
for j=1:N
    if X(j,1)<0.5 && X(j,2)<0.5
       count=count+1;
    end
end
fprintf('Count is %d\n',count);
```

What should we find? As the dimension increases, the data becomes sparse. Notice that initially our neighborhood took half of the data, but as we increase the input dimensions to just 4, that number drops signicantly. Can you see why geometrically?

This phenomenon is known as the **curse of dimensionality**: As we increase the number of input dimensions, the problem becomes exponentially more difficult.

Chapter 11

Linear Neural Networks

In this chapter, we introduce the concept of the linear neural network. As we will see, a neural network is a biologically inspired algorithm that allows us to build a functional representation from data. In statistical terms, a neural network generally represents nonlinear regression. In this chapter, however, we start with a linear network and examine its properties and shortcomings.

11.1 A Model of Learning

D.O. Hebb (1904-1985) was a physiological psychologist at McGill University. In Hebb's view, learning could be described physiologically: There is some physical change in the nervous system to accommodate learning, and that change is summarized by what we now call Hebb's postulate (from his 1949 book):

> When an axon of cell A is near enough to excite a cell B and repeatedly takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

As with many named theorems and postulates, this was not an idea that was completely new, but he does give the postulate in a form that can be used as a basis for machine learning. Here are some questions you might think about:

1. Hebb's postulate describes a strengthing or weakening of connections. How is this biologically or chemically done? What exactly is that "physical change"?

2. There is also near instantaneous learning- Circumstances that are so emotionally intense, that we do not require repeated exposure. But perhaps internally repeating the exerience is enough.

3. The postulate does not give any consideration to *feedback*. If the action causes pain, will the neuron connections still be strengthened? How does emotion in general mitigate or assist in these constructions? Again, we can't give authoritative answers to these questions.

We'll leave these questions for you to think about, and move into something we can answer. How do you mathematically model Hebb's postulate?

11.2 Linear Neural Nets

We will go into the formal details later for defining *neural nets*, but this is a good place to get a feel for what they're all about.

Let us first build a simple model for a neuron. A neuron has three body parts- The dendrites, which carry information to the cell body, the cell body, and the axon, which carries information away from the cell body.

Multiple signals come in to the cell body from the dendrites. Mathematically, we will assume they all arrive at the same time, and the action of the dendrites (or the arrival site of the cell body) is that each signal is changed by the physiology of the cell. That is, if x_i is information along dendrite i, arrival at the cell body changes it to $w_i x_i$, where w_i is some real value. Next, the cell body collates this information by summing these signals together. So far, then, this action is simply a dot product of the vector $\boldsymbol{w}$ (the *weights*) to the signal $\boldsymbol{x}$. An additional value is added to the result, which we can think of as the resting state of the cell (or in statistical terms, the bias). In Figure 11.1, we graphically depict the flow of information from the input layer to the output layer.

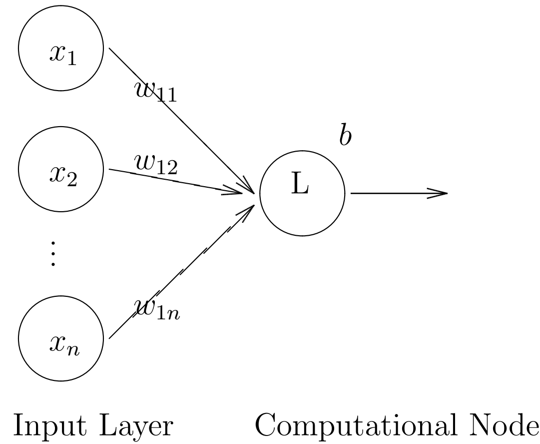


Figure 11.1: The Linear Node. Information travels from left to right along the edges.

Some vocabulary that we'll use:

- $x_1, \ldots, x_n$ are presented to the "input layer". Some researchers call this an actual layer, some do not (which makes some counts of the network different).
- The w's are called the "weights", and we will also denote the edge by the same notation. When a signal travels along an edge, the result is that the signal is multiplied by the weight.
- At node L, the sum of the incoming signals is taken, and added to a value, b. We think of b as the "resting state" of the cell, which is also called the bias term.

We see that mathematically, this single node of a linear network is an affine function from $\mathbb{R}^n$ to $\mathbb{R}$:

$$\boldsymbol{x} \mapsto (w_{11}, w_{12}, \ldots, w_{1n}) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b = w_{11}x_1 + w_{12}x_2 + \ldots + w_{1n}x_n + b = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

If we have m neurons and $\mathbf{x}$ is a vector in $\mathbb{R}^n$, then W is a $m \times n$ matrix, and each row corresponds to a signal neuron's weights (that is, W_{ij} refers to the weight taking x_j to neuron i). Graphically, we see this in Figure 11.2. The full mapping is now formally an affine map from $\mathbb{R}^n$ to $\mathbb{R}^m$ (affine because we're adding the bias terms in $\boldsymbol{b}$).

$$\boldsymbol{x} \to W\boldsymbol{x} + \boldsymbol{b}$$

As we know, problems that are *linear* are usually easier to work with, so we can use a "trick" from computer graphics to convert our affine map to a linear map by going up one dimension. First an example of how this will be done:

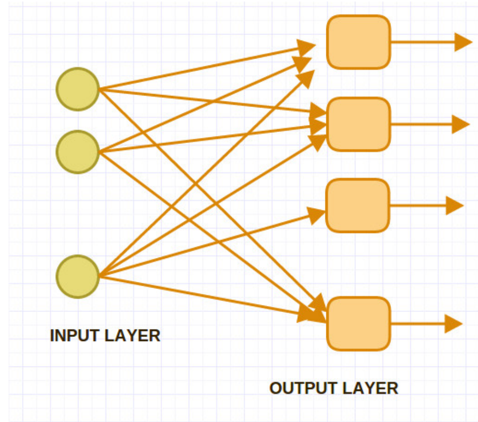


Figure 11.2: The Linear Neural Network is an affine mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$

Example (Convert Affine to Linear)

Suppose that we have the 2×2 affine problem:

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

If we put the vector $\boldsymbol{b}$ as the last column of the matrix, we just need to add a dimension to the vector $\boldsymbol{x}$ by putting a 1 in that position. That is, you should verify that our affine map is equivalent to the following linear map:

$$\begin{pmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

As a summary, we have the following conversion: Given the affine problem

$$A\boldsymbol{x} + \boldsymbol{b} = \boldsymbol{y}$$

define $\hat{A} = [A \quad \boldsymbol{b}]$ and $\hat{\boldsymbol{x}} = [\boldsymbol{x},\ 1]^T$. Then the affine map is equivalent to the linear map

$$\hat{A}\hat{\boldsymbol{x}} = \boldsymbol{y}.$$

11.3 Training a Network

So far, we've seen that a linear neural network is modeled by $W\boldsymbol{x}+\boldsymbol{b}$. In this case, we would call the weights and biases the **model parameters**. We now need to determine the model parameters given some data, and that is what is called **training**.

To set up the problem, we are given p data pairs (where t is for known "targets").

$$(\boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)})$$

where our goal is to determine the weights W and biases b so that

$$W\mathbf{x} + \mathbf{b} = \boldsymbol{y} \approx \boldsymbol{t}$$

There are many ways one may perform the training, but there is one way of classification that is helpful: On-line or Batch.

- On-line training (or on-line learning) is an adaptive approach where the weights and biases are modified after looking at a single data pair $(\boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)})$. A network with this kind of training is called an **Adaline** net (for "Adaptive Linear").

- Batch training is a one-step training method where the weights and biases are determined after the entire data set has been analyzed.

As it turns out, both training processes are designed to minimize an error function, and the most common way to construct an error function is to take the sum of squares error (SSE). If we assume that we have p data pairs, then the sum of squares error is defined as the following.

$$E(W, \boldsymbol{b}) = \sum_{j=1}^{p} \|\boldsymbol{t}^{(j)} - \boldsymbol{y}^{(j)}\|^2 = \sum_{j=1}^{p} \|\boldsymbol{t}^{(j)} - (W\boldsymbol{x}^{(j)} - \boldsymbol{b})\|^2$$

Some formulations of E will multiply it by $1/2$ so the derivative doesn't have a "2" in it, or will take the average error (so multiply by $1/p$). If we multiply E by a constant, the values of W and $\boldsymbol{b}$ that give us the optimal value will be the same, so we may do so if it makes our computations easier.

In the next section, we'll talk about online training, and later, we'll discuss batch training.

11.4 Hebbian Learning (On-line training)

We'll recall that the linear network inputs a pattern, $\boldsymbol{x} \in \mathbb{R}^n$, and it outputs a pattern, $\boldsymbol{y} \in \mathbb{R}^m$ (remember that in our notation, $\boldsymbol{y}$ represents the output of the network, and $\boldsymbol{t}$ is our known target data). In terms of the weights, matrix W is $m \times n$, and so individually,

W_{ij} connects the jth value of the input to the ith value of the output.

Thus we might take the following as Hebb's Rule: The change in the weight connecting the jth input to the ith cell is given by:

$$\Delta W_{ij} = \alpha y_i x_j$$

where α is called **the learning rate**.

If both x_j and y_i match in sign, then W_{ij} becomes larger, and if there is a mismatch in sign, W_{ij} gets smaller. This is the **unsupervised Hebbian rule**. Let's take a closer look at this using some linear algebra. With $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$, then the outer produce $\mathbf{y}\mathbf{x}^T$ produces an $m \times n$ matrix (same dimensions as W):

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} [x_1, x_2, \ldots, x_n] = \begin{bmatrix} y_1x_1 & y_1x_2 & \cdots & y_1x_n \\ y_2x_1 & y_2x_2 & \cdots & y_2x_n \\ \vdots & & & \vdots \\ y_mx_1 & y_mx_2 & \cdots & y_mx_n \end{bmatrix}$$

You should verify that in this case, we can compactly write Hebb's rule as:

$$W_{\text{new}} = W_{\text{old}} + \alpha \mathbf{y}\mathbf{x}^T$$

and that this change is valid for a single $\mathbf{x}, \mathbf{y}$ stimulus-response pair.

There are some difficulties with unsupervised Hebbian learning- in particular, if α stays fixed, then the update rule will "blow up" on us- We need to incorporate the targets.

11.4.1 Widrow-Hoff Learning

We want to define the update rule in such a way as to go to zero as the output $\boldsymbol{y}$ gets closer to the target $\boldsymbol{t}$. Here is one way to accomplish this, and it is called the Widrow-Hoff learning rule[1]:

$$\Delta W_{ij} = \alpha(t_j - y_j)x_i$$

If we put this in matrix form, the learning rule becomes:

$$W_{\text{new}} = W_{\text{old}} + \alpha\,(\mathbf{t} - \mathbf{y})\,\mathbf{x}^T \tag{11.1}$$

where $(\mathbf{x}, \mathbf{t})$ is a desired input-output relation, and $\mathbf{y} = W\mathbf{x} + \boldsymbol{b}$, and the update rule for the bias vector is similar:

$$\mathbf{b}_{\text{new}} = \mathbf{b}_{\text{old}} + \alpha\,(\mathbf{t} - \mathbf{y}) \tag{11.2}$$

11.4.2 Derivation of Widrow-Hoff (Exercises)

In this series of exercises, you'll see that the Widrow-Hoff update is really an approximation to gradient descent on our error function - in fact, we've seen it in stochastic gradient descent. First, we'll look at one-dimensional output, and then extend that to multidimensional output.

For notation, let $k = 1, 2, \ldots, p$ index the data. Let $(\boldsymbol{x}^{(k)}, t^{(k)})$ denote the input, target pairs for the linear network, and the output of the network is $y^{(k)} = \boldsymbol{w}^T\boldsymbol{x}^{(k)} + b$, and E is the SSE as defined earlier.

1. Find an expression for $\partial E/\partial w_j$ (be sure to substitute the function in for y), where E is our sum of squares error.
2. Find an expression for $\partial E/\partial b$.
3. Using the previous two answers, what would our update rule look like if we performed gradient descent on the error function?
4. Instead of using the full error function, we will estimate the full error by using only one data point. That is, for the kth data point,

   $$E(\boldsymbol{w}, b) \approx (t^{(k)} - y^{(k)})^2$$

   Show that using the approximation gives us the Widrow-Hoff rule, where α is a constant.

   $$\boldsymbol{w}_{\text{new}} = \boldsymbol{w}_{\text{old}} + \alpha(t_k - y_k)\boldsymbol{x}^{(k)} \qquad \text{and} \qquad b_{\text{new}} = b_{\text{old}} + \alpha(t_k - y_k).$$

5. Show that the multidimensional extension leads us to Equation 11.1 and 11.2.

11.4.3 Looking at the Learning Rate

In order to optimize the learning rate, it is possible to show that, if matrix H is the Hessian of our error function, then as long as α is bounded as shown:

$$0 < \alpha < \frac{1}{\lambda_{\max}}$$

then the Widrow-Hoff algorithm will converge. For us, we can show that the Hessian on the full error will be XX^T, if X is $n \times p$ (so a scaled covariance matrix). Since we're approximating the error with the single vector $\boldsymbol{x}$, then:

1. As an exercise, show that $\|\boldsymbol{x}\|^2$ is the non-zero eigenvalue for the rank 1 matrix $\boldsymbol{x}\boldsymbol{x}^T$.
2. Matlab uses the following estimate for the maximum learning rate:

   $$\alpha = \frac{0.9999}{\texttt{max(eig(xx}^T\texttt{))}} = \frac{0.9999}{\|\boldsymbol{x}\|^2}$$

[1] Also goes by the names Least Mean Squares rule, and the delta rule.

11.4.4 Implementation Details

We'll provide Matlab/Python implementations separately, but let's consider the details using pseudo-code.

In our code, we'll assume that the data is in matrix X, the desired targets are in matrix T, and the training parameters are weight matrix W and vector $\mathbf{b}$. We'll assume that X is $n \times p$, and that sets up the dimensions for the other matrices and vectors:

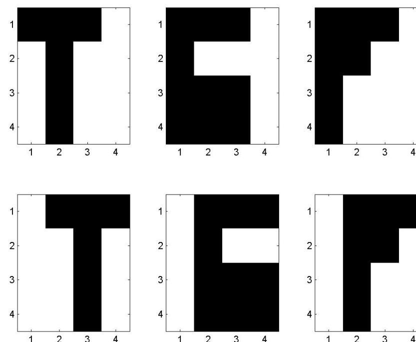
$$X \text{ is } n \times p \quad T \text{ is } m \times p \quad W \text{ is } m \times n \quad \mathbf{b} \text{ is } m \times 1$$

- Randomly initialize `W` and `b` (`randn` is better than `rand` for this).
- Loop for i from 1 to the number of epochs (an epoch is one pass through the data)
  - Randomly permute the training data.
  - For j from 1 to the number of points p:
    * Compute the output y using the jth input.
    * Compute the error: $t - y$ for the jth input.
    * Update the weights: $W_{\text{new}} = W_{\text{old}} + \alpha(t-y)x^T$
    * Update the b: $b_{\text{new}} = b_{\text{old}} + \alpha(t-y)$.
    * End of j loop.
  - Compute the SSE for this epoch:
    * `temp`$= T - (WX + \mathbf{b})$.
    * SSE$=$ sum(temp. $*$ temp).

11.4.5 Example: Associative Memory

Here we will reproduce an experiment by Widrow and Hoff[2] who built an actual machine to do this (we'll do a computer simulation).

We'll have three letters as input, T, G and F. We'll associate these letters to the numbers $-60, 0, 60$ respectively. We want our network to perform the association using the Widrow-Hoff learning rule.
The letters will be defined by 4×4 arrays of numbers, where 1 corresponds to the color black, and -1 corresponds to the color white. In this example, we'll have two samples of each letter, as shown in the figure to the right.



Implementation and problem specification:

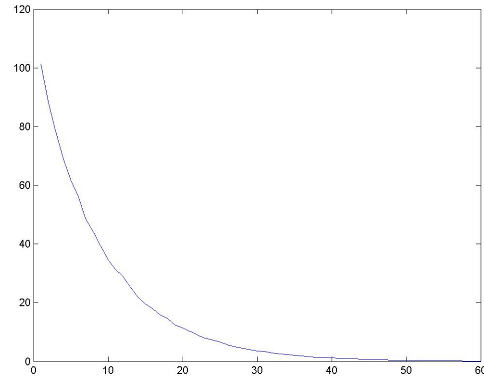
- Dimensions:

$$X \text{ is } 16 \times 6, \qquad T \text{ is } 1 \times 6, \qquad W \text{ is } 1 \times 16, \qquad b \text{ is a scalar}$$

- We'll use an $\alpha = 0.03$.
- We'll take 60 passes through the data (60 training epochs).

[2]See "Adaptive Switching Circuits" by B. Widrow and M.E. Hoff, in 1960 IRE WESCON Convention Record, New York: IRE, Part 4, p. 96-104. You'll find reprints on the internet.

The plot of the error is shown in the figure to the right. The horizontal axis counts the number of passes through the data, and the vertical axis gives the sum of the squared errors. Since we only had 6 points, the error graph (taken on the full set) is just to show that the training is succeeding, it does not represent the error of the classifier.



11.5 One step training

In this section, we'll assume that all of the data is available to us at the beginning of training. Further, rather than affine form ($Wx + b$), we'll convert the output to the linear form by creating $\hat{W}$ and $\hat{X}$. If you'll recall, we start with:

$$W\left[\boldsymbol{x}^{(1)},\ \boldsymbol{x}^{(2)},\ \ldots,\ \boldsymbol{x}^{(p)}\right] + [\boldsymbol{b},\ \boldsymbol{b},\ \ldots,\ \boldsymbol{b}] = \left[\boldsymbol{y}^{(1)},\ \boldsymbol{y}^{(2)},\ \ldots,\ \boldsymbol{y}^{(p)}\right],$$

and then we transform this into a linear problem by appending $\boldsymbol{b}$ to the last column of W and we put a row of 1's as the bottom row in the data:

$$\hat{W} = [W \quad \boldsymbol{b}], \qquad \hat{X} = \begin{bmatrix} \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \ldots & \boldsymbol{x}^{(p)} \\ 1 & 1 & & 1 \end{bmatrix}$$

Now the output of the linear network is simply $Y = \hat{W}\hat{X}$, and we can solve the system of equations for $\hat{W}$ directly

$$\hat{W}\hat{X} = T$$

In particular, we can use the pseudo-inverse from the **reduced** SVD of $\hat{X}$:

$$\hat{X} = U\Sigma V^T \quad \Rightarrow \quad \hat{W} = TV\Sigma^{-1}U^T$$

If $\hat{X}$ has rank k, be sure you can identify the dimensions of all the matrices above.

11.6 Small Batch Training

Between training using a single point at a time versus using all the data, we might update the parameters after gathering some small number of data points together. One advantage to this is that the direction of gradient descent will be better established than using a single point.

Furthermore, rather than completing training as with one-step learning, regular online learning keeps the network flexible and adaptable to new changes in the data.

In small batch training, the algorithm proceeds as with the single point training, except t, y, and x are replaced by their averages taken over the small group. Further, the value of α can be approximated by

$$\alpha \approx \frac{0.99}{\max(\mathrm{eig}(XX^T))}$$

where X in this context is the small group of points $\boldsymbol{x}$ collected in a temporary matrix X.

11.7 Time Series and Linear Networks

In this section, we'll see that linear neural networks can play a role in signal processing. In that context, we're thinking of an incoming signal parameterized by t and sampled digitally to create a sequence of points. We'll start there.

Definitions

A **time series** is a sequence of real numbers. We denote a time series in the usual way:

$$S = \{x(1), x(2), x(3), \ldots, x(t), \ldots\}$$

A **tapped delay line with** k **taps** is constructed from a time series:

$$\hat{x}_1 = \begin{pmatrix} x(1) \\ x(2) \\ \vdots \\ x(k) \end{pmatrix}, \quad \hat{x}_2 = \begin{pmatrix} x(2) \\ x(3) \\ \vdots \\ x(k+1) \end{pmatrix}, \quad \hat{x}_3 = \begin{pmatrix} x(3) \\ x(4) \\ \vdots \\ x(k+2) \end{pmatrix}, \ldots$$

This is also called a **lagged vector (lag k)**. We would say that the time series has been **embedded in** $\mathbb{R}^k$.

Consider the Matlab code that would embed the data into $\mathbb{R}^5$.

```
p=length(S);
X=zeros(k,p-k);
for j=1:k
    X(j,:)=S(j:p-(k+1-j));
end
T=S(k+1:p);
```

This code is the basis for the function `lagX.m` on our class website. The function creates our domain vectors as the columns of matrix X, and the set of targets (in this case, 1 dimensional) in vector T. From here, we can apply the neural network and train it. That is, our goal is to build a time series predictor. We're going to predict the value of x_6 based on x_1, x_2, x_3, x_4, x_5. Then we predict the value of x_7 based on x_2 through x_6, and so on. Actually, this kind of a function is called a filter.

Definition: A *filter* with $k-$taps is a function on the time series so that:

$$x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-k})$$

In particular, a *linear filter* will have the form:

$$\boldsymbol{w}^T \boldsymbol{x} + b = x_i$$

where $\boldsymbol{w}$ are the weights, b is the bias, and $\boldsymbol{x} = (x_{i-1}, x_{i-2}, \ldots, x_{i-k})^T$.

Side Remark: In signals processing dialect, the linear filter above is called a Finite Impulse Response (FIR) filter

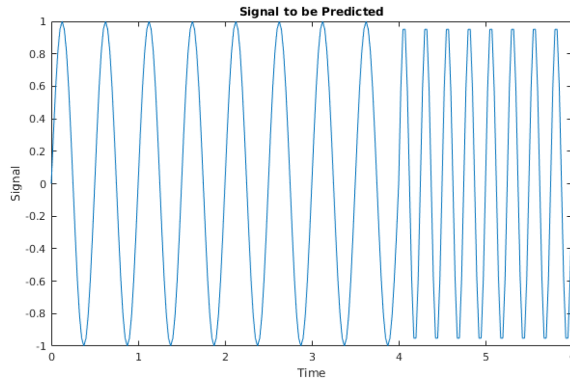
11.7.1 Example: Adapting to a Changing Signal

In this example, we build a time series from two time series- the signal is:

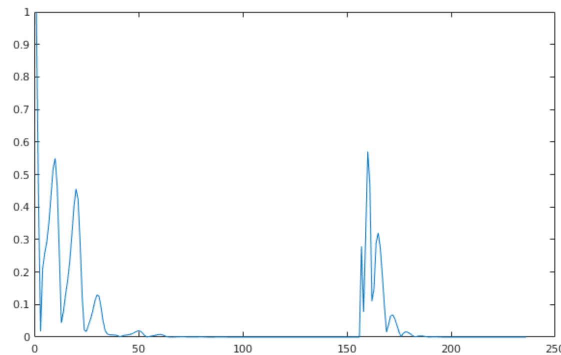
$$S(t) = \begin{cases} \sin(4\pi t) & \text{if } 0 \le t < 4 \\ \sin(8\pi t) & \text{if } 4 \le t \le 6 \end{cases}$$

The idea is that the online training will find the weights and biases for the first part of the signal quite rapidly. The hope is that, when the signal changes, the Widrow-Hoff algorithm will allow the function to rapidly adapt to the new signal (and it does!).

Below is the original signal.



The value of α in this example was fixed. We're doing that so we can see the result of changing the learning rate. For this example, we found that the optimal fixed value was approximately 0.153, but you might change that in the script for yourself. Below is shown the output error as a function of the time index. Indeed, we see that the error rapidly goes to zero, but then pops up and adapts as the signal changes.



Application: Noise removal

This idea is presented by Matlab. I haven't tried it myself yet, but it would be interesting to see what kinds of results you would get.

- Background: There is a signal that we would like to have as pure as possible, but there is some noise contaminating it. For example, a pilot's voice may be contaminated by engine noise. We would like to remove the noise using a linear neural network. We assume that the noise *source* is available for sampling, but the noise contamination is an unknown function of the noise source.
- GOAL: Filter out the noise, given only access to the noise source.
- Idea for the solution:

  Suppose that the noise source is input (using time delays) to a linear filter. What can the linear network do? It can only form linear combinations of its past values, and therefore can only estimate signals that are (at least) correlated to the noise source. The pilot's voice (or signal of interest) should NOT be correlated to the noise.

  If we ask a linear network to model the noise PLUS the pilot's voice, the linear network will only be capable of modeling the noise.

This gives us an easily implemented algorithm:

Let v_k be the main signal (or voice) sampled at time k. Let n_k be the sample of the noise source at time k. The contaminated signal is then: $v_k + f(n_k)$, where f is a (unknown) model of how the noise is transformed.

We will design the linear network so that:

- INPUT: $n_k, n_{k-1}, \ldots, n_{k-(m-1)}$ (m lags)
- DESIRED OUTPUT: $v_k + f(n_k) = c_k$
- ACTUAL NETWORK OUTPUT: a_k

Algorithm: At time k, input the lagged vector, and compute a_k. The error is $a_k - c_k$. Use the Widrow-Hoff learning rule to update the weights and bias.

Homework

The homework will consist of a computer lab where you'll practice training the linear neural network. Since the specific implementation will depend on the coding language used, we'll keep that as a separate document.

Chapter 12

Radial Basis Functions

12.1 Introduction

The neural network has been so popular because of it is actually a **universal function approximator**. That is, for any given function (expressed partially as data), there is a neural network that will approximate it. This extraordinary property means that the applications of neural networks are only bounded by our ability to compute and train them.

We're going to start our discussion of the more general neural nets by looking at one class in particular- The Radial Basis Functions (or RBFs). While they had been around for a long time, it was an important work by Dave Broomhead and David Lowe, "Multivariable Functional Interpolation and Adaptive Networks" published in 1988 that connected the RBF to the neural net.

Further, RBFs were initially used (Powell, late 1970s) to perform *interpolation* rather than regression- That means that they were used to find an *exact fit* to a high dimensional function rather than looking for a generalization (in the regression problem). We know from that, that it is possible to drive the training error to zero (which would typically be overfitting), and so we need to keep that in mind when we're training.

We will see that the RBF network has some very attractive features: Training can be split up and computed in layers, perhaps making it more tractable, and they are fast and intuitive as well.

12.2 Radial Basis Functions

The architecture of a radial basis function consists of the following pieces, which we'll go through more carefully in a moment:

- A set of **centers**, $\{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_k\}$. The centers are in the same space as the input data, $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$, and can in fact be selected from the input data.
- One or more **transfer functions**, ϕ (a map from $\mathbb{R}$ to $\mathbb{R}$). Common choices will be given shortly.

The action of the RBF network is then defined as follows (given in layers): The first mapping is from the input space, $\mathbb{R}^n$ to $\mathbb{R}^k$ (where k is the number of centers). At this point, the transfer function ϕ is applied to each element, and that is followed by an affine mapping to the output layer, $\mathbb{R}^m$.

$$\mathbf{x} \rightarrow \begin{bmatrix} \|\mathbf{x}-\mathbf{c}_1\| \\ \|\mathbf{x}-\mathbf{c}_2\| \\ \vdots \\ \|\mathbf{x}-\mathbf{c}_k\| \end{bmatrix} \rightarrow \phi\left(\begin{bmatrix} \|\mathbf{x}-\mathbf{c}_1\| \\ \|\mathbf{x}-\mathbf{c}_2\| \\ \vdots \\ \|\mathbf{x}-\mathbf{c}_k\| \end{bmatrix}\right) \rightarrow W \begin{bmatrix} \phi(\|\mathbf{x}-\mathbf{c}_1\|) \\ \phi(\|\mathbf{x}-\mathbf{c}_2\|) \\ \vdots \\ \phi(\|\mathbf{x}-\mathbf{c}_k\|) \end{bmatrix} + \boldsymbol{b} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{12.1}$$

Before going further, we might just make a note of where the name "radial basis function" comes from.

Definition: A radial function is any function of the form $\phi(\mathbf{x}) = \phi(\|\mathbf{x}\|)$, so that ϕ acts on a vector in $\mathbb{R}^n$, but only through the norm so that $\phi:\ [0,\infty) \to \mathbb{R}$.

It is possible to then take some set of radial functions and then have a basis for some function space. We're going to use the radial basis functions for function approximation however.

Definition: The Transfer Function and Matrix Let $\phi : \mathbb{R}^+ \to \mathbb{R}$ be chosen from the list below.

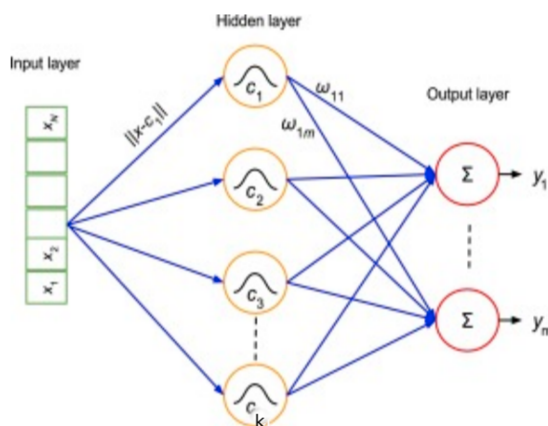
$$
\begin{aligned}
\phi(r,\sigma) &= \exp\left(\frac{-r^2}{\sigma^2}\right) && \text{Gaussian} \\
\phi(r) &= r^3 && \text{Cubic} \\
\phi(r) &= r^2 \log(r) && \text{Thin Plate Spline} \\
\phi(r) &= \frac{1}{r+1} && \text{Cauchy} \\
\phi(r,\beta) &= \sqrt{r^2+\beta} && \text{Multiquadric} \\
\phi(r,\beta) &= \frac{1}{\sqrt{r^2+\beta}} && \text{Inverse Multiquadric} \\
\phi(r) &= r && \text{Identity}
\end{aligned}
$$

The reader may note that some transfer functions have extra parameters- Like the Gaussian, and Multiquadrics. Also, we will assume that ϕ, when applied to a vector or a matrix, will be defined element-wise.

There are other transfer functions one can choose. For a broader definition of transfer functions, see Micchelli [30]. We will examine the effects of the transfer function on the radial approximation shortly, but we focus on a few of them. Matlab, for example, uses only a Gaussian transfer function, which may have some undesired consequences (we'll see in the exercises).

12.2.1 The RBF as a Neural Network

The RBF as a neural network is shown to the right. The input layer has as many nodes as input dimensions. The middle layer has k nodes, one for each center $\mathbf{c}_k$. The processing at the middle layer is to first compute the distance from the input vector to the corresponding center, then apply ϕ. The resulting scalar value is passed along to the output layer, $\mathbf{y}$. The last layer is linear in that we will be taking linear combinations of the values of the middle layer.



Training the RBF can be split into two steps. The first step is to locate the centers and to determine the transfer function (and its parameters). The second step is to build the design matrix Φ and solve the linear system of equations for the weights and biases, W and $\boldsymbol{b}$.

It is possible for us to include center locations and the transfer function parameters in the error function. There are lots of journal articles about how this might be done, but none have turned out to be that popular. This is probably due to the fact that separating centers and transfer function parameters out makes the problem fast and linear- this is the primary reason one would use RBFs instead of other methods.

12.2.2 Determining the Centers and transfer function

Center Selection

We will primarily use one of the following techniques to select the centers "offline" (meaning this can be and is usually done independently from finding the weights and biases).

- $k-$means clustering (or any clustering that returns cluster centers).

  Data clustering is popular, but what we're really doing is spreading the clusters around the input data. The desired targets are typically not taken into account.

- Random center selection: This seems like it would be a terrible thing to do since it guarantees a suboptimal solution, but for smaller data sets, this does actually work pretty well.

- Orthogonal Least Squares: This is Matlab's default algorithm, and we'll discuss this later in this section.

- Center placement as part of the optimization problem. This is possible, but as we noted before, for larger problems this might become an unreasonable thing to do (in practice). We won't do this, although it is interesting to see where the centers end up.

Notes on the transfer function

Some transfer functions do not require us to provide values for extra parameters. The cubic function can be an attractive option that way. However, the Gaussian is probably the most popular choice for transfer function, and so it is good to have some way to set the spread. It is possible to always set the spread to 1 by re-scaling the data appropriately, but there may be reasons not to do that.

When looking at $k-$means, some recommend using the distance between centers as a guide for setting the spread. If we define the spread as `spr`, where

$$\phi(r) = \mathrm{e}^{-\mathtt{spr}\, r^2}$$

then, if M is the maximum distance between our k centers,

$$\mathtt{spr} = \frac{M}{\sqrt{2k}}.$$

Matlab's formula for calculating `spr` is, given the desired spread s (default is $s = 1$) then

$$\mathtt{spr} = \frac{\sqrt{-\ln(1/2)}}{s} = \frac{\sqrt{\ln(2)}}{s}$$

At this point, we'll assume that the **location**, **number** of centers, and the **transfer function** with its parameters has been chosen. We continue on to train the net.

12.2.3 Training the RBF

As usual, we should start by examining the data we're given, scale it if necessary, and split it into training and testing sets. We don't necessarily require k-fold cross validation for the remainder, although we could use it to try to get a better estimate of the error (of course, that requires extra data that we may not have). Given all that, let's assume that we have p training points, each point is in $\mathbb{R}^n$. Further, we have p targets, each in $\mathbb{R}^m$. We're using k centers and the transfer function ϕ has been defined and its parameters set.

Training proceeds by setting up the linear algebra problem for the weights and biases- We use the diagram in Equation 12.1 for each input $\boldsymbol{x}$ output $\boldsymbol{t}$ pairing to build the system of equations which we will solve using the least squares error.

1. Build the design matrix Φ, where
$$\Phi_{i,j} = \phi(\|\boldsymbol{x}_j - \boldsymbol{c}_i\|)$$
As defined here, Φ will be $k \times p$ (number of centers by number of points).

2. Let T be the $m \times p$ matrix representing p target vectors in $\mathbb{R}^m$.

3. Now the weight matrix takes us from the middle layer (in $\mathbb{R}^k$) to the outer layer (in $\mathbb{R}^m$), so it should be $m \times k$. Further, the bias vector $\mathbf{b}$ should be $m \times 1$. The equation we need to solve for $W, \boldsymbol{b}$ is given by:
$$W\Phi + \boldsymbol{b} = T \tag{12.2}$$

4. It's best to convert the equation in the previous step to a linear equation (rather than affine). We modify W and Φ to take care of that:
$$\hat{W} = [W \quad \boldsymbol{b}], \qquad \hat{\Phi} = \begin{bmatrix} \Phi \\ 1, 1, 1, \cdots 1 \end{bmatrix}$$
Now, $\hat{W}$ is $m \times k+1$ and $\hat{\Phi}$ is $k+1 \times p$, and our previous equation becomes:
$$\hat{W}\hat{\Phi} = T$$

5. Finally, take the pseudoinverse of Φ (manually using the SVD or by using a built-in command) and solve, giving
$$\hat{W} = T\hat{\Phi}^{\dagger}.$$
We also recall that $\hat{W} = [W\, \boldsymbol{b}]$, so if we want these separated, we can do this now.

12.2.4 Predicting data using the RBF

Once we have constructed the RBF, we can predict how new data will behave. To do this, remember that we need the following:

- The set of centers, usually as a matrix.
- Which transfer function ϕ is being used, and its parameters, if necessary.
- The weights and bias values, W and $\boldsymbol{b}$.

Once we have these, given N new input vectors, we would perform the following steps to output the result:

1. Compute the distance matrix between the N data points and k centers. Let's assume this is a $k \times N$ matrix.

2. Apply the transfer function ϕ, and this results in the $k \times N$ matrix Φ.

3. Apply our affine map, where W is $m \times k$:
$$W\Phi + \boldsymbol{b} = Y$$
(This is not good linear algebra notation, but I hope you understand that this notation means to add the column vector $\boldsymbol{b}$ to each column of the matrix $W\Phi$).

4. Y is $m \times N$ and the model output (or prediction).

Example

As an example of the interplay between the number of centers and the error, let's try an experiment.

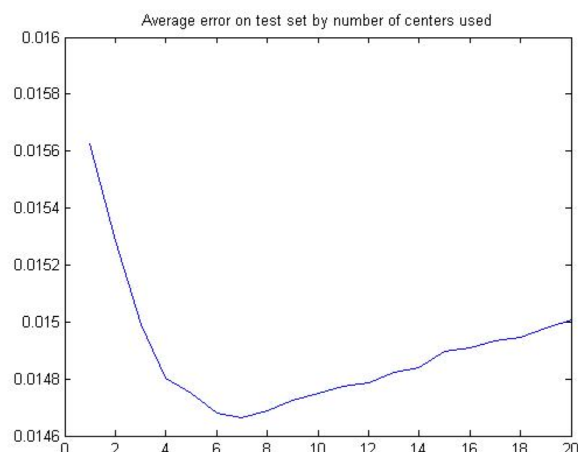
The data will be two dimensional, and the "true" underlying function will be

$$z = \exp\left(-\frac{x^2+y^2}{4}\right)$$

with some noise added (with standard deviation of about 0.5).

We'll have 1500 points in the domain, and we'll randomly select the centers (for each training session) from the data, and then train the network on 300 points. This could give us a lot of variation in the error, so we'll train 30 times (each with a different selection of centers), and take the average for the given number of cluster centers.

We will then consider the error on the **test** set of 1200 points, and plot the error versus the number of centers.



As we expect, the error decreases initially as we increase the number of centers. However, once we go beyond a certain point, the error starts to **increase** again. This is the point at which we should stop adding centers, because it means we're starting to **overfit**.

Exercises

1. The following exercises will consider how we might set the width of the Gaussian transfer function.

   (a) We will approximate:

   $$\left(\int_{-b}^{b} \mathrm{e}^{-x^2}\,dx\right)^2 = \int_{-b}^{b}\int_{-b}^{b} \mathrm{e}^{-(x^2+y^2)}\,dx\,dy \approx \int\int_B \mathrm{e}^{-(x^2+y^2)}\,dB$$

   where B is the disk of radius b. Show that this last integral is:

   $$\pi\left(1-\mathrm{e}^{-b^2}\right)$$

   (b) Using the previous exercise, conclude that:

   $$\int_{-\infty}^{\infty} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\,dx = \sigma\sqrt{\pi}$$

   (c) We'll make a working definition of the *width* of the Gaussian: It is the value a so that k percentage of the area is between $-a$ and a (so k is between 0 and 1). The actual value of k will be problem-dependent.

   Use the previous two exercises to show that our working definition of the "width" a, means that, given a we would like to find σ so that:

   $$\int_{-a}^{a} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\,dx \approx k\int_{-\infty}^{\infty} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\,dx$$

(d) Show that the last exercise implies that, if we are given k and a, then we should take σ to be:

$$\sigma = \frac{a}{\sqrt{-\ln(1-k^2)}} \tag{12.3}$$

The previous exercises give some justification for Matlab's choice for approximating σ:

$$\sigma = \frac{a}{\sqrt{-\ln(0.5)}}$$

(I'm not sure why they stick with $-\ln(1/2)$ rather than just $\ln(2)$!)

12.3 Orthogonal Least Squares

The following is a summary of the work in the reference [7]. We present the multidimensional extension that is the basis of the method used in Matlab, but does not seem to be widely available in Python yet (Apr 2021).

First, we'll get the big picture, and then we'll look at some of the linear algebra that makes it work. It's most convenient to start with the linear version of the RBF equation:

$$\hat{W}\hat{\Phi} = T$$

Dimensions are important here, so let's define them again: We'll assume that we have p data points in $\mathbb{R}^n$ as input, and the output dimension is $\mathbb{R}^m$.

Initially, we'll assume that every data point is a center, so we have p centers. That means that W is $m \times (p+1)$, Φ is $(p+1) \times p$, and T is $m \times p$.

The first thing to notice is that j^{th} **row** of Φ (except for the last one) corresponds to the jth center $\boldsymbol{c}_j$ (that is, the row of Φ is constructed by taking distances from $\boldsymbol{c}_j$ and the p points in X).

The second thing to notice is that on the left side of the equation, we're working with the **row space** of $\hat{\Phi}$ and on the left side, we have the **rows** of T. Orthogonal Least Squares (or OLS) assumes that we'll get a "good solution" to the RBF equation if it is possible to select a small number of rows of Φ (corresponding to a small number of centers) from which we can construct the rows of T.

Another way to think of OLS is that we want to try to use a small number of rows of $\hat{\Phi}$ (not really counting the last row) in order to form a basis for the row space of T.

We cannot use the SVD in this case, because we want to use **actual** rows of $\hat{\Phi}$ and not combinations of rows- that's an important restriction to keep in mind.

The OLS Algorithm

Begin with all data in $\hat{\Phi}$, so this is now $(p+1) \times p$.

1. Look for the row of Φ (in $\mathbb{R}^p$) that most closely points in the same direction as a row of T. Call the winning row index w.
2. Put point $\mathbf{x}_w$ as a center.
3. Remove row w from $\hat{\Phi}$
4. Remove the component of the wth row from the other rows (this makes all the remaining rows orthogonal to row w).
5. Compute the error on the data using the current set of centers as the model.
6. Repeat until the error is small enough, or we have used the maximum number of centers allowed.

This algorithm always stops, either because some maximum number of centers has been reached, or because we have used all the data (and the RBF equation will be interpolating the test data).

Exercises for the OLS

There is some lovely linear algebra here. Let's review it!

1. Show that, if we have a set of vectors $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k]$ and a vector $\boldsymbol{y}$, then the vector in X that most closely points in the direction of $\boldsymbol{y}$ (or $-\boldsymbol{y}$) can be found by computing the maximum of:

$$\left(\frac{\boldsymbol{x}_i^T \boldsymbol{y}}{\|\boldsymbol{x}_i\|\|\boldsymbol{y}\|}\right)^2 \tag{12.4}$$

   (Hint: What does this expression represent? Have we seen it before?)

2. To "remove the component of $\mathbf{y}$ in the direction of $\mathbf{x}$" means to do what, exactly? It means that we want to subtract the projection of $\mathbf{y}$ onto $\mathbf{x}$ from $\mathbf{y}$- which means that the "new version" of $\mathbf{y}$ will be orthogonal to $\mathbf{x}$. Show that this formula will do the trick (that is, show $\mathbf{y}_{\text{new}} \perp \mathbf{x}$).

$$\mathbf{y}_{\text{new}} = \mathbf{y} - \text{Proj}_x(\boldsymbol{y}) = \mathbf{y} - \frac{\mathbf{y}^T \mathbf{x}}{\mathbf{x}^T \mathbf{x}} \mathbf{x}$$

   Side Remark: We did this back in linear algebra when we looked at Gram-Schmidt orthogonalization.

3. Why might OLS not be used much now? (Hint: What if you have a very large number of data points?)

Homework:

Homework will be assigned separately, since it will involve an application of the RBF to several different data sets, and you may choose to use Python or Matlab/Octave.

12.4 Summary of the Radial Basis Functions

1. What is a Radial Basis Function?

   A radial basis function (RBF) is a general model to build a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$.

2. Choices that need to be made when using an RBF:

   (a) The transfer function ϕ that will be used.
   Some choices: $\phi(r) = r$, $\phi(r) = \mathrm{e}^{-r^2/\sigma^2}$, $\phi(r) = r^3$, etc.

   (b) The model parameters for an RBF are given by:
   - The number of centers, k.
   - The center locations, $\mathbf{c}_i$.
     Common choices:
     - The centers are randomly chosen from the data.
     - The centers are chosen as the centroids of data clusters.
     - The centers are chosen automatically using Orthogonal Least Squares (this is Matlab's default).
     - The centers are chosen through optimization.
     - In all cases, we could also define individual values of each parameter (like the widths of the Gaussians).

3. Once ϕ and the centers have been chosen, we solve for weights and biases using a least squares solution. That is:

- Compute the distance matrix between the k centers and p data points.
- Form the transfer matrix Φ by applying ϕ to the distance matrix, so Φ is $k \times p$.
- Add a row of ones to the bottom of matrix Φ, so that $\hat{\Phi}$ is $(k+1) \times p$.
- Solve $\hat{W}\hat{\Phi} = T$ by using the pseudoinverse, so $\hat{W} = T\hat{\Phi}^{\dagger}$.

4. To test the function on new domain points (we have now fixed the centers and the weight matrix W):
   - Form the distance matrix between the k centers and the $\hat{p}$ new data points, this will be $k \times \hat{p}$.
   - Apply ϕ to the distance matrix to get Φ (dimenesions $k \times \hat{p}$).
   - Add a row to the bottom of matrix Φ to get $\hat{\Phi}$.
   - The new output is $\hat{W}\hat{\Phi}$.
5. Why use an RBF?
   (a) Once the centers have been chosen, this is a linear modeling problem. That is, to find the weights, we are solving a linear equation. Therefore, an RBF is much faster than methods that involve nonlinear optimization.
   (b) There are nice connections to statistics if we use the Gaussian transfer function, although we have not discussed them.
6. Software and the RBF.

   There are two ways of producing an RBF model in Matlab- one is to do it explicitly yourself, the other is to use Matlab's built-in routines using the "Neural Network Toolbox". If you're using the toolbox, then the OLS is built into the `newrb` command.

   Unfortunately, Octave doesn't maintain a version of the neural network toolbox anymore, so the subroutines needed need to be all provided (I have written those separately for you to use).

   Alternatively, there isn't much pre-written for Python for some reason (April 2021), so I will provide you with the functionality needed, along with some examples.

Chapter 13

Neural Networks

To give you an idea of how new this material is, let's do a little history lesson. The origins of neural nets are typically dated back to the early 1940's and work by two physiologists, McCulloch and Pitts. Hebb's work came later, when he formulated his rule for learning. In the 1950's came the *perceptron. The perceptron* is what we called a linear neural network- it became clear that the perceptron could only classify certain types of data (what we now call linearly separable data). This shortcoming led to a stop in neural net research for many years- It was not until the 1980's that neural net research really took off from a coming together of many disparate strands of research- There was a group in Finland led by T. Kohonen that was working with self-organizing maps (SOM); there was the work from the 70s with Stephen Grossberg, and finally several researchers were discovering methods for training new networks that could be put in parallel (back-propagation).

It was in the 80's and 90's that researchers were able to prove that a neural network was a **universal function approximator**- That is, for any function with certain nice properties, we are able to find a neural network that will approximate that function with arbitrarily small error. It is interesting to note that the use of a neural network could be used to solve Hilbert's 13th Problem[1]:

"Every continuous function of two or more real variables can be written as the superposition of continuous functions of one real variable along with addition."

Let's look at some highlights of more recent developments in neural nets:

- 1989: George Cybenko proves that neural nets with certain types of activation functions are universal function approximators.
- 1989: Yann LeCun used convolutional neural networks to read handwritten digits. This led to many rapid developments, and Dr. LeCun shared the Turin prize in 2018 for this work.
- 1989: Q-Learning is something that takes machine learning to a different branching- This greatly improves the area of reinforcement learning.
- 1993: Support Vector Machines (SVM), around for a long time but brought into machine learning, designed by C. Cortes and V. Vapnik.
- 1998: Yann LeCun brings in Stochastic Gradient Descent as a practical solution to large problems.
- 2006 or so: Approximately the beginning of "deep learning", as coined by Geoffrey Hinton (a key player in machine learning for a very long time, also given the Turin award in 2018).
- 2009: The launch of ImageNet (Fei-Fei Li at Stanford). As of 2017, it contains 14 million **labeled** images that are available to researchers. The casual reader may not appreciate how key this is-

[1] In 1900, mathematician David Hilbert listed what he viewed as the 23 greatest unsolved problems. This list has been worked on ever since, and as of 2021, problems 8, 12, 13 and 16 remain unsolved.

"Deep learning" requires vast amounts of data, and this kind of data was not readily available to the community earlier.

- 2011: IBM's Watson computer wins *Jeopardy!*.
- 2011: The use of the Rectified Linear Unit (or ReLU) as activation function. This function is used in deep networks to counter what is called "the vanishing gradient problem".
- 2012: Creation of AlexNet - Its success kicked off a spike in researching convolutional neural nets (that continues to this day).
- 2014:

Generative Adversarial Networks (GAN). Basically, a second neural net is built as an "adversary", where it builds data to try to fool the neural net that is learning a particular task. The basic idea has expanded, and these networks can now generate realistic images of human beings. The person to the right does not exist.
(Photo from Wikipedia Commons)

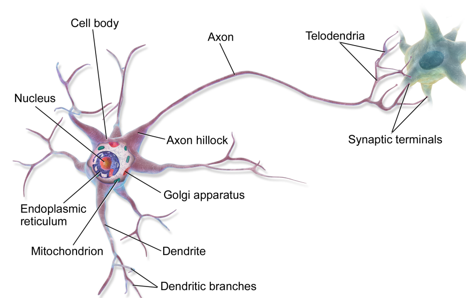
Coming to present day, research is aimed at something called **deep neural nets-** in summary, the difference now is that we have a LOT of training data, and new algorithms and architectures that make it feasible to train complex associations. A nice summary of "The Decade of Deep Learning" is given at the site below (accessed Apr 2021):
`https://bmk.sh/2019/12/31/The-Decade-of-Deep-Learning/`

Back to feed forward nets

The term "neural network" has come to be an umbrella term covering a broad range of different function approximation or pattern classification methods. The classic type of neural network can be defined simply as a connected graph with weighted edges and computational nodes- We've seen a linear neural network (using Widrow-Hoff training rule). We now turn to the workhorse of the neural network community: The feed forward neural network.

13.1 From Biology to Construction

We've seen the basic model before in the linear neural networks. We include some of it again for clarity. Information flows from the dendrites to the cell body through the axon to a synaptic junction connecting to the dendrite to the next neuron.
The synaptic junction is made up of the presynaptic node (the end of an axon), the postsynaptic node (the beginning of a dendrite), and the "empty space", which is the synapse.
A diagram of a neuron is shown to the left.



As information flows across a synapse, information can be amplified, inhibited, or re-polarized. We'll discuss our model of cell body processing and flow over the axon, but this is all the biology we use. It's best to think of neural nets as being inspired by biology, although there are researchers interested specifically in working with biological neural nets.

Before we look at the neural net on the level of a layer of neurons, let's look at how a single neuron (or "node") processes information:

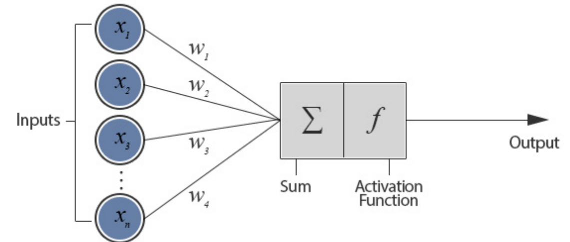
Let $x_1, \ldots, x_n$ denote the signals incoming along the dendrites. We model the processing done here by multiplying the signal by a real number which we call a **weight**.

At the cell body, these signals are first collected in what we'll call the **prestate** of the cell. They are added to a "bias term", b, which loosely represents the resting state of the cell.

$$p = \sum_{j=1}^{n} w_j x_j + b = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

An activation function is applied, we'll just call it f for now. Applying f, we get what is called the **state** of the cell, or the cell's **activation**:

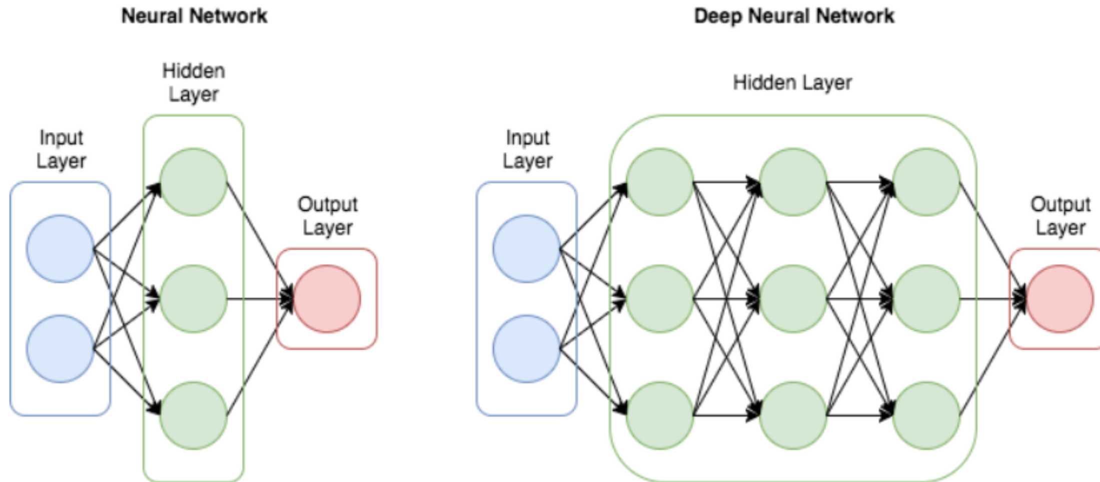
$$s = f(p)$$



And this value is then sent along the axon. Here is a typical way we might make a diagram of a single neuron, where the cell body has been divided into the sum stage, then the activation.

Putting lots of cells together is what a neural network is. Below on the left, we show a three layer network. Data is input at the first layer, the second layer is really the layer of cells, and the last layer is the output layer. In finding a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$ with k cells, we would construct a $n - k - m$ network.

Below and to the right, we see networks don't have to have a single layer; we can apply lots of layers to get a multilayer network. Training these can be difficult, and is the subject of our next chapter in deep learning. For the time being, we'll keep with a single hidden layer.



Next we'll look at how information flows through the network. It is easiest to describe using layers and linear algebra.

13.2 How Networks Compute

At the input layer, we're presented with a real value at each input node, which we'll denote as a vector $\mathbf{x} \in \mathbb{R}^n$. Next, we move from $\mathbb{R}^n$ to $\mathbb{R}^k$. The affine mapping that takes us from the input layer to the hidden layer is defined as

$$P^{(2)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}$$

where $P^{(2)}$ is the vector in $\mathbb{R}^k$ representing the prestate in each node (or cell). Since the weight matrix $W^{(1)}$ is $k \times n$, we make an important observation:

$$W_{jk}^{(i)} \text{ is the weight connecting cell } k \text{ from layer } i \text{ to cell } j \text{ on layer } i+1.$$

It is tempting to say this connection backwards ("j connected to k"), so please be careful with that- It is the opposite of what you might think it is.

Now continuing, we apply function f to the prestate vector P (element-wise) to get the state vector S.

$$S^{(2)} = f\left(P^{(2)}\right) = f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)$$

Now we apply an affine transformation from $\mathbb{R}^k$ to the m dimensional output layer. This means that the second weight matrix has dimensions $m \times k$:

$$P^{(3)} = W^{(2)}S^{(2)} + \mathbf{b}^{(2)}$$

Typically, we don't do any more processing, so we can think of the activation function f on the final layer as the identity function, $f(x) = x$.

$$S^{(3)} = P^{(3)} = W^{(2)}S^{(2)} + \mathbf{b}^{(2)} = W^{(2)}\left(f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)\right) + \mathbf{b}^{(2)}$$

To be consistent with all layers, we usually define the prestate at the input layer to be just $\boldsymbol{x}$, and the activation function as the identity, so the layer doesn't do anything but output the same as what is input. With that, the computations a neural network takes can be described by the chain:

$$\text{input} = (P^{(1)} \to S^{(1)}) \longrightarrow (P^{(2)} \to S^{(2)}) \longrightarrow (P^3 \to S^{(3)}) = \text{ output}$$

It is simple now to see how to add more layers. We can keep extending this chain almost indefinitely.

The Network Produces Composition(s)

There is one important way to look at this sequence of operations that has some implications when we compute derivatives: This sequence of operations is function composition, which is clear when we write the output layer in terms of the input layer:

$$S^{(3)} = W^{(2)}\left(f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)\right) + \mathbf{b}^{(2)}$$

If we add more layers, then the state at the third layer should be written again as $f(P^{(3)})$, because presumably, f would not now be the identity. In that case, the output at the added fourth layer would be:

$$S^{(4)} = W^{(3)}\left[f(W^{(2)}\left(f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)\right) + \mathbf{b}^{(2)})\right] + \boldsymbol{b}^{(3)}$$

and so on.

13.3 The Activation Function

The function f here is key. If f is linear, then the neural net is a linear network (we've already discussed those). Importantly, f on the hidden layer must be nonlinear. And importantly, in 1989, Cybenko proved the result about universal function approximation assuming that f is a **sigmoidal function**.

Definition: Suppose a function $\sigma(x)$ has the following properties:

- It is monotonically increasing.

- $\lim_{x \to -\infty} \sigma(x) = A$
- $\lim_{x \to \infty} \sigma(x) = B$

Then $\sigma(x)$ is called a **sigmoidal function**.

Common choices for the sigmoidal function include:

1. The log sigmoidal function (in Matlab, this is `logsig`)

$$\text{logsig}(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

2. The hyperbolic tangent (in Matlab, this is `tansig`)

$$\tanh(x) = \frac{\mathrm{e}^{2x} - 1}{\mathrm{e}^{2x} + 1}$$

Typically, we will use the first option, since its derivative is very easy to compute (see the exercises), and typically the so-called "linear activation function", $\sigma(x) = x$ is used at the input and output layer.

13.3.1 Recent Developments for the Activation Function

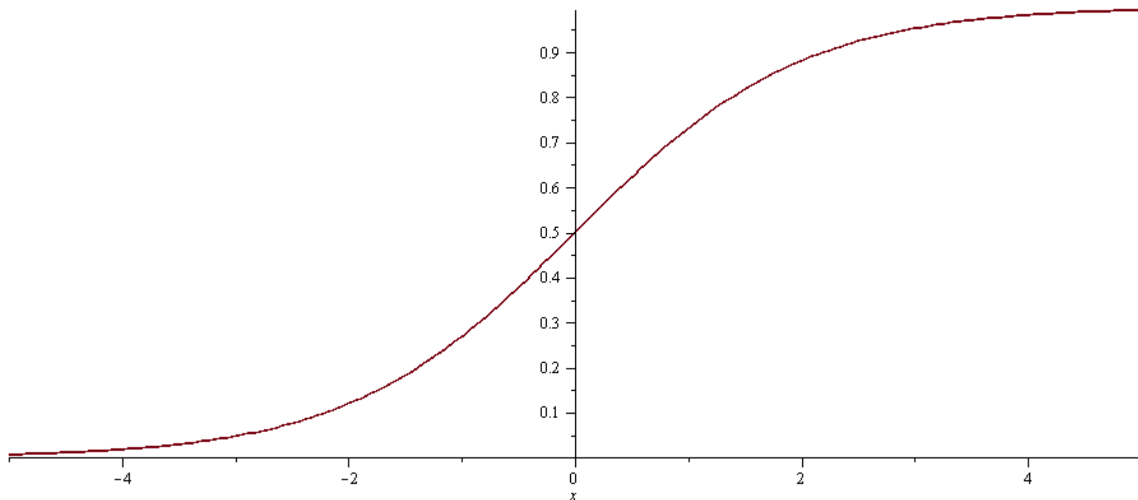
While the sigmoidal function was used by Cybenko in his proof, it has some shortcomings that didn't become clear until very large networks were being trained. The two main shortcomings:

- The "squashing" function becomes saturated.

  The sigmoidal only has a small interval on which it can keep input points apart (see the graph below). If $|x|$ is greater than about 2, then all positive points start to map to 1, and all negative points to 0 (this is explored more in the exercises).

- The vanishing gradient problem.

  From the graph, see if you can esimate the derivative. For most x, the derivative is less than 1. Recall that a neural net is function composition so that the derivative is multiplication. What happens when you multiply a bunch of numbers that are less than 1? The values go to zero- This is the vanishing gradient problem (we'll discuss this more in the section on training).



The Rise of ReLU

The activation function that most use now is the "rectified linear unit", or **ReLU**. Mathematically, this is defined as:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

The perhaps obvious issues with using this function are (1) nondifferentiable at 0, and (2) it becomes unbounded. Further, if a learning rate is set too high, it has been observed that the ReLU's can be "pushed" out so far that they are basically zero- this is the "Dying ReLU problem".

In spite of the difficulties, in 2011 it was found that the ReLU function is a viable alternative to the sigmoidal function, even though it does not have all the properties of a sigmoidal. A few short years later, in 2017 it was found to be the most popular activation function for deep networks.

Exercises

1. Sketch the ReLU transfer function and its derivative.

2. If $\boldsymbol{x} \in \mathbb{R}^n$ and our targets $\boldsymbol{t} \in \mathbb{R}^m$, and we use k nodes in the hidden layer, how many unknown parameters do we have to find?

   As you are constructing your network, keep this number in mind. In particular, you should have at least several data points for each unknown parameter that you are looking for.

3. We have to be somewhat careful when our data is badly scaled. For example, complete this table of values:

x	0	0.5	1	10	40	100
$\tanh(x)$						
$\mathtt{logsig}(x)$						

   You should see that, while the change in x between neighboring points is getting very large, that the corresponding changes in y are going to zero. This is problematic- If your data is large, or the weights are large, then the sigmoidal may simply start to output a constant.

   This phenomenon goes by the name of *saturation.*

4. Some people like to scale the sigmoidal function by an extra parameter, β, that is $\sigma(\beta x)$. Show by sketching what happens to the graph of the sigmoidal (either the `tansig` or `logsig`) as you change β.

   It is not necessary to scale the sigmoidal, as this is equivalent to scaling the data instead (via the weights).

5. Show that if $\sigma(x)$ is the `logsig` function, then

   $$\sigma'(\beta x) = \beta\sigma(\beta x)(1 - \sigma(\beta x))$$

   This is the reason this function is so popular- It is very easy to compute its derivative.

6. Show, using the definition of the hyperbolic sine and cosine, that the hyperbolic tangent can be written as:

   $$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

7. Show that the hyperbolic tangent can be computed as:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

(Matlab claims that this version is faster, but warns about possible numerical error)

8. **Other Extensions of the activation function**

Some other interesting activation functions can be used at the nodes. Here are a couple of unique ones- They are used to encode circular or spherical information:

(a) The Circular Node (two inputs, two outputs per node):

$$\sigma(x, y) = \left(\frac{x}{\sqrt{x^2 + y^2}}, \frac{y}{\sqrt{x^2 + y^2}} \right)$$

(b) The Spherical Node (three inputs, three outputs):

$$\sigma(x, y, z) = \left(\frac{x}{\sqrt{x^2 + y^2 + z^2}}, \frac{y}{\sqrt{x^2 + y^2 + z^2}}, \frac{z}{\sqrt{x^2 + y^2 + z^2}} \right)$$

See [19, 20] for examples of how to implement the last two transfer function types.

13.4 Training and Error

To define a three layer neural network in the form $n - k - m$, we should first define the activation function f. Although we could define a different function for every neuron, we typically will use the same transfer function for all the neurons in a single layer.

Once that is done, then we have to find matrices $W^{(1)}, W^{(2)}$ and the bias vectors $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}$. Altogether, this makes $(nk + k) + (mk + m)$ parameters. Ideally, we would have much more data than that in order to get good estimates. In any case, we want to minimize the usual sum of squared error:

$$E(W^{(1)}, W^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}) = \frac{1}{2} \sum_{i=1}^{p} \|\mathbf{t}^{(i)} - \mathbf{y}^{(i)}\|^2$$

where $\mathbf{y}^{(i)}$ is the output of the neural net using the ith input. In the case of a single hidden layer, we have:

$$\mathbf{y}^{(i)} = W^{(2)} \left(\sigma \left(W^{(1)} \mathbf{x}^{(i)} + \mathbf{b}^{(1)} \right) \right) + \mathbf{b}^{(2)}$$

As usual, "training" means finding the weights and biases that minimize the error function, and now we're back at function optimization. We've discussed these methods before, but here is a short list:

- Method of Steepest Descent (or Gradient Descent).
- Stochastic Gradient Descent is an excellent one to use when we're programming it ourselves, since we only have to deal with the derivative one data point at a time.
- Newton's Method (an indirect method, solving for where the derivative of the error is 0).
- Conjuage Gradient (Search along the eigenvectors of the Hessian of the error)
- Levenburg-Marquardt (A combination of the techniques above).

These techniques are typically implemented in any good system- one can find these solvers in Matlab and in Python, look at `scipy.optimize.least_squares`. Before we go too much farther into the general case, let's take a look at a concrete example.

Most optimization algorithms will require that we obtain expressions to compute the partial derivatives of E with respect to the weights and biases. It is fortunate that, for large networks, there is a recursive algorithm to accomplish this called the **backpropagation of error**. In the next section, we're going to look at the computational steps of this algorithm, then after that, we'll prove that we have actually computed the desired partial derivatives.

13.5 Backpropagation of Error

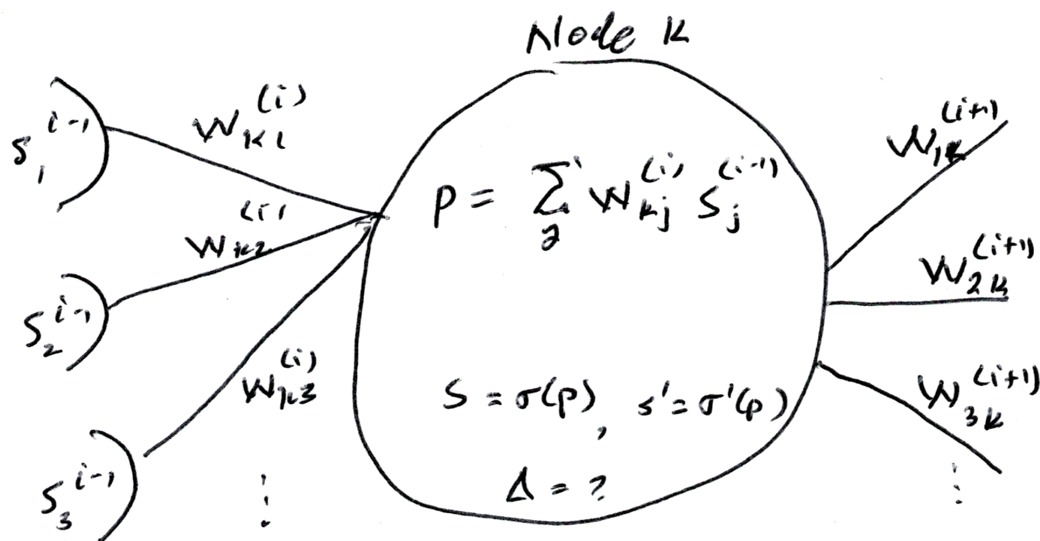
First, let's reconsider the single node of a neural network, and consider the computations it makes. Since we're looking to program this soon, think of each node as storing these values somewhere for future access.

When we go through what's called a **forward phase** for a network, we move from left to right computing the given values, where P is the prestate, S is the state (so S' is its derivative evaluated at P), and Δ is a placeholder for the **backwards phase**

In particular, looking at node k in layer i as in the diagram below, the computations are (layer-wise):

$$P^{(i)} = W^{(i)} S^{(i-1)} + \boldsymbol{b}^{(i)}, \qquad S^{(i)} = \sigma(P^{(i)}) = \sigma\left(W^{(i)} S^{(i-1)} + \boldsymbol{b}^{(i)}\right) \qquad {S^{(i)}}' = \sigma'(P^{(i)})$$

If N_{i-1}, N_i, N_{i+1} are the number of nodes in layers $i-1$, i, and $i+1$ respectively. The first operation through this layer represents an affine map from $\mathbb{R}^{N_{i-1}}$ to $\mathbb{R}^{N_i}$ that is the layer's prestate. So $W^{(i)}$ is an $N_i \times N_{i-1}$ matrix, and $P^{(i)}$ is a vector in $\mathbb{R}^{N_i}$, and so is $S^{(i)}$. The derivative is the same size, and the derivatives (evaluated at the current prestate) are stored in ${S^{(i)}}'$. In the picture below, $P, S,$ and S' are all scalar values being computed specifically at node k.



There are two special cases:

- The input layer:
$$P^{(1)} = \boldsymbol{x}, \qquad S^{(1)} = \boldsymbol{x} \qquad S(1)' = \texttt{ones}$$

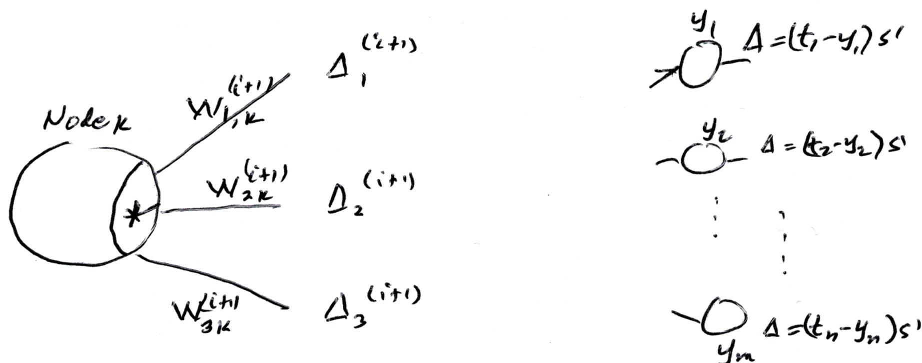
- At the output layer, we will make the assumption that the activation function is the identity. If we have L layers, then:

$$P^{(L)} = W^{(L)} S^{(L-1)} + \boldsymbol{b}^{(L)}, \qquad S^{(L)} = P^{(L-1)}, \qquad {S^{(L)}}' = \texttt{ones}$$

And the computation at the output layer finishes the forward phase.

The "backwards phase" for the network is to move information from right to left, where our goal is now to compute the values of Δ that were left blank.

Starting at the output layer, $\Delta^{(L)} = \boldsymbol{t} - \boldsymbol{y}$, which is the error between the target value and the output of the net. Notice that in that equation, Δ is a vector of values, one value for each output node, and in the picture to the right below, we included S', but if the activation function is the identity, this is just equal to 1.



The asterisk in the image above and to the left represents the computation we make for that node's Δ value. In this particular case, think of the $\Delta's$ in layer $i+1$ moving to the left so we have to multiply by the corresponding weight. We then sum those up and multiply by the derivative there. That is:

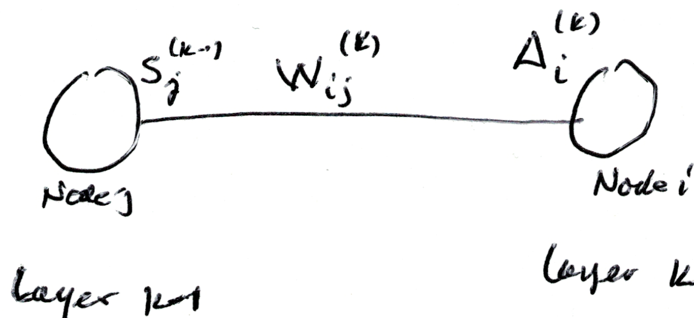
$$\Delta_k^{(i)} = S_k^{(i)'} \sum_{j=1}^{N_{i+1}} \Delta_j^{(i+1)} W_{jk}^{(i+1)}$$

This may look complicated, but think of it in terms of the reverse affine map from $\mathbb{R}^{N_{i+1}}$ to $\mathbb{R}^{N_i}$. Then we can compute all of the $\Delta's$ in layer i all at once:

$$\Delta^{(i)} = ((W^{(i+1)})^T \Delta^{(i+1)}) .* S^{(i)'}$$

Somewhere a linear algebra teacher is crying. OK, so just to be crystal clear, our multiplication is not a dot product, we are performing the multiplication (using the Matlab symbol `.*`) between two vectors *element-wise* so the result is also a vector of the same size. Also to be very clear, let's figure out the dimensions of everything there: $W^{(i+1)}$ is $N_{i+1} \times N_i$ (so the transpose reverses those), $\Delta^{(i)}$ has N_{i+1} elements, and $S^{(i)'}$ has N_i elements.

Once we reach the input layer, we're done. Now what do we do with these Δ values? I like to use the following memory device to help me remember how this goes. Below we see an edge, and we want to use the **state on the left** and the **Δ on the right** with the edge and weight connecting the two.



Then here is the big conclusion:

Backpropagation of Error

$$\Delta W_{ij}^{(k)} = S_j^{(k-1)} \Delta_i^{(k)} \tag{13.1}$$

The update rule for the weights is now (using gradient descent):

$$\text{new } W_{ij}^{(k)} = \text{old } W_{ij}^{(k)} + \alpha \Delta W_{ij}^{(k)}$$

where the plus sign is not a typo, and α is the learning rate.

Thinking back, this kind of update rule was a lot like Widrow-Hoff, and in that case, we were able to write the update using linear algebra so we could update weights all at once.

Update Rule Using Linear Algebra

Remember that matrix $W^{(i)}$ connects layer $i-1$ to layer i, so that $W^{(i)}$ is $N_i \times N_{i-1}$. In our update rule, we are taking states from layer $i-1$ (so we have a vector with N_{i-1} entries) and the deltas from layer i (so that is a vector with N_i entries. To give away the punch line, if we take the outer product between these two vectors (in the right order), we'll have a matrix the same size as $W^{(i)}$. You should verify this works by looking at the dimensions of the three objects below.

$$\Delta W^{(i)} = \Delta^{(i)} (S^{(i-1)})^T$$

And finally,

$$W_{\text{new}}^{(i)} = W^{(i)} + \alpha \Delta W^{(i)}$$

You might think we're done, but there are still three big questions we have to answer:

1. These computations were for a single data point. What do I do with p data points?
2. Is this really gradient descent? Prove it!
3. Where did the bias terms go??

These questions will be answered in the next section, where we do actually show that our computations do result in gradient descent.

13.6 Backprop Proved

The answer to the **first question** goes pretty quickly. With p data points, if we use the sum of squares as the error, then each of the 4 numbers we computed per node should be summed over all the input (that's before the backward phase). Some people use the average error, and in that case, you would average each of the 4 numbers over all the input.

For the **second question**, we'll need to break out our Calculus. The function we want to minimize is the error function. We'll put a 1/2 in the front so we don't have to deal with putting 2 in front of everything when we differentiate.

$$E = \frac{1}{2} \sum_{i=1}^{p} \| \boldsymbol{t}^{(i)} - \boldsymbol{y}^{(i)} \|^2$$

We'll show that our rules do indeed produce the gradient descent. Recall that W^l_{mn} connects node n in the layer to the left to node m in the layer to the right. Therefore, S^l_m is the state of node m in layer l (to the right of the edge labeled W^l_{mn}). Using these relationships, we can write:

$$\frac{\partial E}{\partial W^l_{mn}} = \frac{\partial E}{\partial S^l_m} \frac{\partial S^l_m}{\partial P^l_m} \frac{\partial P^l_m}{\partial W^l_{mn}} \tag{13.2}$$

The two values on the right can readily be computed:

$$\frac{\partial S^l_m}{\partial P^l_m} = \sigma'\left(P^l_m\right) \qquad \frac{\partial P^l_m}{\partial W^l_{mn}} = S^{l-1}_n \tag{13.3}$$

This leaves the first term which can be evaluated on the output layer L:

$$\frac{\partial E}{\partial S^L_m} = \frac{\partial E}{\partial y_m} = (t_m - y_m)(-1)$$

On the rest of the layers, the term can be defined recursively,

$$\frac{\partial E}{\partial S^l_m} = \sum_{j \in \text{nextlayer}} \frac{\partial E}{\partial S^{l+1}_j} \frac{\partial S^{l+1}_j}{\partial S^l_m} \tag{13.4}$$

Since $S^{l+1}_j = \sigma(P^{l+1}_j) = \sigma(W^{l+1}_{jm} S^l_m + \text{ other terms})$, the derivative will be

$$\frac{\partial S^{l+1}_j}{\partial S^l_m} = \sigma'(P^{l+1}_m) W^{l+1}_{jm}$$

Now we'll connect up the two sets of notation:

Definition: We'll define Δ as the product of the first two terms of Equation (13.2):

$$\Delta^l_m = -\frac{\partial E}{\partial S^l_m} \frac{\partial S^l_m}{\partial P^l_m} = -\frac{\partial E}{\partial S^l_m} \sigma'(P^l_m)$$

Therefore, on the output layer,

$$\Delta^L_m = \frac{\partial E}{\partial S^L_m} \frac{\partial S^L_m}{\partial P^L_m} = -(t_m - y_m)(-1)\sigma'(P^L_m)$$

which matches Equation (**??**). Now, using Equation (13.4), the nodes on the previous layer are computed:

$$\Delta^l_m = -\frac{\partial E}{\partial S^l_m} \frac{\partial S^l_m}{\partial P^l_m} = \left(\sum_{j \in \text{layer} l+1} -\frac{\partial E}{\partial S^{l+1}_j} \frac{\partial S^{l+1}_j}{\partial S^l_m} \right) \sigma'(P^l_m) =$$

$$\sigma'(P^l_m) \left(\sum_{j \in \text{layer} l+1} -\frac{\partial E}{\partial S^{l+1}_j} \sigma'(P^{l+1}_m) W^{l+1}_{jm} \right) =$$

$$\sigma'(P^l_m) \left(\sum_{j \in \text{ layer} l+1} \Delta^{l+1}_j W^{l+1}_{jm} \right)$$

And this is Equation (**??**). Finally, by Equation (13.3), we can write:

$$-\frac{\partial E}{\partial W^l_{mn}} = \left(-\frac{\partial E}{\partial S^l_m} \frac{\partial S^l_m}{\partial P^l_m} \right) \frac{\partial P^l_m}{\partial W^l_{mn}} = \Delta^l_m S^{l-1}_n$$

which proves that the update rule in Equation (13.1) is indeed gradient descent. Now for the second question: How should we deal with bias terms?

For the bias term, we will add a node to each layer, but for these nodes, the state is the constant function, $S = \sigma(x) = 1$, and the weight connecting this node to node k of the next layer could be labeled b_k^l. That also means that Δ for a bias node is 0, but now Equation (**??**) becomes:

$$b_k^l = b_k^l + \epsilon \Delta_k^l$$

where the Δ_k^l is the value of Δ to the node to which b_k^l is connected.

13.7 Simple Construction of a Feed Forward Neural Net

The neural net is simple to construct if we view it in terms of its layers.

- Initialize the $n - k - m$ network:

  Build the weights and biases with random numbers. Be sure to pay attention to dimensions.

  $$W^{(1)} \text{ is } k \times n \qquad \boldsymbol{b}^{(1)} \text{ is } k \times 1$$
  $$W^{(2)} \text{ is } m \times k \qquad \boldsymbol{b}^{(2)} \text{ is } m \times 1$$

- Compute the output of a neural net given the weights, biases. Here we'll assume that the activation function is $\sigma(x)$, and has been determined. The computation proceeds in layers. For the three layer network (layers 0, 1, and 2), we'll have the series of computations. This is a forward pass:
  - $P^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$
  - $S^{(1)} = \sigma(P^{(1)})$. Compute $S(1)'$ as well.
  - $P^{(2)} = W^{(2)}S^{(1)} + \boldsymbol{b}^{(2)}$
  - $S^{(2)} = P^{(1)}$. Then $S(1)'$ is a vector of ones. The output is $S^{(2)}$.
- A backwards pass for backpropagation of error:
  - $\Delta^{(2)} = (\mathbf{t} - \mathbf{y})$ (This is a vector in $\mathbb{R}^m$).
  - $\Delta^{(1)} = (W^{(2)})^T \Delta^{(2)} \texttt{ .* } S^{(i)'}$ (We use Matlab's `.*` to denote elementwise mutliplication)
- Compute the changes to the weights and biases:
  - $\Delta W^{(1)} = \Delta^{(1)} \mathbf{x}^T$
  - $\Delta W^{(2)} = \Delta^{(2)} (S^{(1)})^T$
  - $\Delta \mathbf{b}^{(1)} = \Delta^{(1)}$
  - $\Delta \mathbf{b}^{(2)} = \Delta^{(2)}$
- Apply the changes:

  $$W^{(1)} = W^{(1)} + \alpha \Delta W^{(1)}, \quad W^{(2)} = W^{(2)} + \alpha \Delta W^{(2)}, \quad \mathbf{b}^{(1)} = \mathbf{b}^{(1)} + \alpha \Delta \mathbf{b}^{(1)} \quad \mathbf{b}^{(2)} = \mathbf{b}^{(2)} + \alpha \Delta \mathbf{b}^{(2)}$$

Chapter 14

Deep Learning

Part III

Appendices

Appendix A

An Introduction to Matlab

Intro To Matlab

Matlab (short for Matrix-Laboratory) was originally designed as a front end for numerical linear algebra routines (numerically finding eigenvalues and eigenvectors, etc.). Now it is an industry standard for fast development of numerical solutions to mathematics.

Matlab is different than Maple: Maple is known as a specific example of a *computer algebra system*, which means that Maple was designed to work symbolically (e.g., factor a polynomial, compute derivatives and antiderivatives symbolically, etc.). On the other hand, Matlab was developed for numerical work. In fact, there are ways of calling Matlab from inside Maple, and vice versa (although we won't be doing that).

If you have had a class in computer programming, then Matlab will look very familiar. There are three ways of running Matlab, and we will consider each:

- Run Matlab "live", by typing in commands.
- Run Matlab by running a script file. A script file is just a text file with the Matlab commands typed in.
- Run Matlab by calling programs that you have written. We'll see how to do this in just a moment.

Starting the Matlab Program

First you need an account in the Mathematics computer lab- See your instructor if you need an account of if you've forgotten your password.

Once you have logged in, open a shell window and type `matlab` at the prompt. The splashscreen will come up, and you'll get a command window that looks like:

```
                              < M A T L A B >
                  Copyright 1984-2001 The MathWorks, Inc.
                       Version 6.1.0.450 Release 12.1
                                May 18 2001

  To get started, select "MATLAB Help" from the Help menu.
>>
```

To exit from Matlab, either type `exit` from the command window, or choose `Exit` from the window menu (under File).

Saving your work

Typically, homework and larger projects should be done by writing a script rather than typing in commands live. It makes it easy to change and debug, rather than typing all your work over again!

Matlab has a very nice text editor that you can use to type out and save Matlab functions and scripts- To access the editor, type `edit` in the Matlab command window.

All of the Matlab files that we write will end in a `*.m` suffix (like `Example01.m`, etc.). Before we go much further, let's open up Matlab and try some commands live:

Introductory Commands

1. **Arithmetic**

   Matlab understands all of the basic arithmetic functions, `+, -, *, /, ^` are addition, subtraction, multiplication, division and exponentiation. Type them in just as you would write them. For example, 2^5 would be typed as `2^5`.

2. **Trigonometric Functions**

   Matlab understands the basic trig functions sine, cosine and tangent as `sin` , `cos` , `tan` . So, for example, the sine of 3.1 would be typed as: `sin(3.1)`

   The number π is used so frequently that Matlab has its (approximate) value built-in as the constant `pi`. For example, $\sin(\pi)$ is typed as `sin(pi)`. Note that π uses a lowercase "P".

3. **Exponential and Logarithmic Functions**

   Matlab does not have the number e built-in as a constant (like π). To take the number e to a power, use the functional form: $\mathrm{e}^x =$ `exp(x)` So if I want the number e, I would type `exp(1)`, and so on.

   For the natural log (log base e), use the notation `log`. For example, $\ln(3)$ is written as `log(3)`. We will only use the natural log- if in the future you want a different base, look up the log command by typing `help log`.

4. **Complex Numbers and Arithmetic**

   Matlab has complex arithmetic built-in. Either the letter i or j can be used to represent $\sqrt{-1}$, but a word of caution is in order here: You can only use i or j for $\sqrt{-1}$ ONLY if you have not previously defined them. If you think you're going to use complex numbers, do not use the letter i for anything but complex arithmetic! Example: `(0.2+3*i)*(5+2*i)` will multiply the two complex numbers together (using complex arithmetic).

Helpful Administrative Commands

The following commands are useful as you begin to use Matlab more and more:

`who` List all variables currently in use.

`whos` List all variables, and their sizes.

`ls` or `dir` List the contents of the current directory.

`cd` Change the directory. For example, `cd examples` would change the current directory to your file named `examples`. To go up the structure instead of down, type `cd ..`

`pwd` Tell me where in the directory structure I'm currently at.

`where` *command* Tell me where *command.m* is located.

`help` *command* List the help file for the function *command*. For example, to get help on the sine function, type `help sin`.

`demo` Lists all the demonstration programs that Matlab came with- This is fun to look at. We don't have all of them; you can go to Matlab's website to look at more: `www.mathworks.com`.

A Programming Note: Assignment v. Equality

In computer programming, the equal sign does not mean mathematical equality. We use the equal sign as an assignment operator. For example,

```
A=3;
```

means to assign the value of 3 to the variable A- If A has not been assigned to anything before, this command will also create that variable.

Definition: In computer programming, $A = B$ means that we will assign the value of B to the variable A.

Using this, what will the following commands do?

```
x=5;
x=x+3;
```

Answer: First, the value 5 is assigned to the variable x. Next, $x + 3$ is computed as 8, and finally, the value of 8 is assigned to the variable x.

Not an example: `5=x;`, you would get an error- the number 5 is not a variable.

The assignment operator is also used to label the output of a function. For example, the following commands stores a vector $[1, 2, 3]$ into the variable x, then we apply the sine function to each of those numbers and store them in y to get $[\sin(1), \sin(2), \sin(3)]$:

```
x=[1,2,3];
y=sin(x);
```

Matrices

Matlab was originally designed as a "front end" to access LINPACK and EISPACK, which are numerical linear algebra packages written in FORTRAN. From this beginning, Matlab's basic data type is the matrix.

I enter the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

as:

```
A=[1 2 3 4; 5 6 7 8];
```

or as:

```
A=[1 2 3 4
5 6 7 8];
```

Note the use of the semicolon: Inside a matrix, the semicolon indicates the end of a row. Outside the matrix, the semicolon suppresses Matlab output. You can also separate numbers using a comma if you'd prefer that. Rows and columns are entered in a corresponding way, as either a $1 \times n$ matrix or as a $n \times 1$ matrix.

We access elements of the matrix in a natural way. For example, the $(2, 3)$ element of A is written as `A(2,3)` in Matlab (in this case, $A(2, 3)$ is 7). You can change the elements using the assignment operator `=`. For example, if we want to change the $(1, 3)$ element of A from 3 to 6, type:

```
A(1,3)=6;
```

Special Commands: The colon operator

- `a:b`

  Produces a listing from a to b in a row:

  $$\texttt{a:b} \text{ gives } [a, a+1, a+2, \ldots, a+n]$$

  where n is the largest integer so that $a+n \leq b$. For example, $x = 2:9$ puts x as a row vector whose elements are the integers from 2 to 9.

- `a:b:c`

  Produces the numbers from a to c by adding b each time:

  $$\texttt{a:b:c} \quad \text{gives} \qquad [a, a+b, a+2b, \ldots, a+nb]$$

  where n is the largest integer so that $a+nb \leq c$. For example, $1:2:7$ returns the numbers $1, 3, 5, 7$. Type the following into Matlab to see what you get: `1:2:8` and `1:0.5:6`

Matlab commands associated with Arrays

- `linspace(a,b,c)`

  Produces c numbers evenly spaced from the number a to the number b (inclusive). For example, `x=linspace(2,3.5,40)` produces 40 numbers evenly spaced beginning with 2 and ending with 3.5.

  SHORTCUT: Leaving off the third number c will give you 100 numbers between a and b (That is, $c = 100$ is the default value.)

  Compare this with the colon operator. We would use the colon operator if we want to define the length between numbers, and use `linspace` if we want to define the endpoints.

- Random arrays (handy if you just need some quick data!)

  `A=rand(m,n)` Produces an $m \times n$ array of random numbers (uniformly distributed) between 0 and 1. If you just want a single random number between 0 and 1, just type `rand`

  `A=randn(m,n)` produces an $m \times n$ array of random numbers (with a normal distribution) with zero mean and unit variance. If you want a single random number (with a normal distribution), just type `randn`

- `A=zeros(m,n)` Produces an $m \times n$ array of zeros.

- `A=ones(m,n)` Produces an $m \times n$ array of ones.

- `A=eye(n)` Produces an $n \times n$ identity matrix.

- `A=repmat(B,m,n)` Matrix A is constructed from matrix (or vector) B by replicating B m times down and n times across.

  Example: Let $B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. Then `A=repmat(B,2,3)` creates the array:

```
A =
     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4
```

Matrix Arithmetic

- Transposition is denoted by the single quote character `'`. That is, `A'` $= A^T$. (CAUTION: If A contains complex numbers, then `A'` is the *conjugate transpose* of A, sometimes denoted as $A^* = \bar{A}^T$)
- Matrix addition and subtraction is performed automatically and is only defined for matrices of the same size.
- Scalar addition. If we want to add a constant c to every item in an array A, type: `A+c`
- Scalar Multiplication: We can multiply every number in the array by a constant: If A is the array and c is the constant, we would write: `B=c*A`
- Matrix Multiplication: Use the regular multiplication sign for standard matrix multiplication. If A is $m \times n$ and B is $n \times p$, then `A*B` is an $m \times p$ matrix, as we did in linear algebra.
- Elementwise Multiplication. We can multiply and divide the elements of an array A and an array B *elementwise* by `A.*B` and `A./B`

  Exponentiation is done in a similar way. To square every element of an array A, we would write: `A.^2` This is the same as saying `A.*A`
- Functions applied to arrays: Matlab will automatically apply a given function to each element of the array. For example, `sin(A)` will apply the sine function to each element of the array A, and `exp(A)` will apply e^x to each element of the array. If you write your own functions, you should always decide ahead of time how you want the function to operate on a matrix.

Accessing Submatrices

Let A be an $m \times n$ array of numbers. Then:

The notation:	Yields:
`A(i,j)`	The (i, j)th element
`A(i,:)`	The entire ith row
`A(:,j)`	The entire jth column
`A(:,2:5)`	The 2d to fifth columns, all rows
`A(1:4,2:3)`	A 4×2 submatrix

Example: What kind of an array would the following command produce?

`A([1,3,6],[2,5])`

A 3×2 matrix consisting of the elements:

$$\begin{matrix} A(1,2) & A(1,5) \\ A(3,2) & A(3,5) \\ A(6,2) & A(6,5) \end{matrix}$$

Example: Create a 5×5 zero array, and change it to:

$$\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}$$

Answer:

```
A=zeros(5,5);  %Create the matrix of zeros
b=[1 2 3;4 5 6; 7 8 9];
A(2:4,2:4)=b;
```

Note also the use of the `%` sign. It is used to denote *comments*; that is, Matlab would ignore everything on the same line after the `%` sign.

Adding/Deleting Columns and Rows:

Its straightforward to insert and/or delete rows and columns into a matrix. Before doing it, we define `[ ]` as "the empty array": the array with nothing in it.

In the following, let A be a 4×5 array, let b be a 1×5 row, and c be a 4×1 column.

Examples of use (each of these are independent from the previous):

The command:	Produces:
`A(1,:)=[];`	Delete the first row.
`A([1,3],:)=[];`	Delete rows 1 and 3.
`A(:,3)=[];`	Delete column 3.
`A(:,1:2:5)=[];`	Delete the odd columns.
`A(1,:)=b;`	Put **b** as row 1.
`A(:,6)=c;`	Add c as the last column.
`d=[c , A(:,1:3)];`	d is c and columns $1-3$ of A.
`A=[A(:,1), c, A(:,2:5)];`	Insert c as column 2 of A, others shift 1 over.
`A=[A(1,:); b; A(2:4,:)];`	Insert b as row 2 of A, others shifted 1 down.

Example: Matlab comes with some built-in data sets. One such set is the image of a clown. For fun, we'll load the array in, display it, then we'll remove all of the even rows and columns, then re-display it:

```
X=load clown.mat
whos
image(X);
colormap(map);
X(2:2:200,:)=[];
X(:,2:2:320)=[];
image(X);
```

Once you're done, you may want to clear the memory and the screen:

```
clear
clc
```

If you want to re-do the clown again, you do not need to retype it! Use the up-arrow key to bring back the commands you typed. You can also type the first few characters, then use the up-arrow key:

```
X=(up arrow)
```

Solving $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$

To solve $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, Matlab has two basic commands: `x=A\b` or `x=pinv(A)*b`. The command `pinv(A)` computes the *pseudoinverse* of A, which we will discuss later in the section dealing with the Singular Value Decomposition.

In linear algebra, there were three possible outcomes for solving $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. They were:

1. A unique solution.
2. An infinite number of solutions.
3. No solution.

Matlab will always give exactly one solution. We need to interpret that solution in the second two cases. In the case of an infinite number of solutions (we have free variables in this case, also called *an underdetermined system*), the two methods may give different answers:

- `x=A\b` provides the most zeros in $\mathbf{x}$.
- `x=pinv(A)*b` gives $\mathbf{x}$ with the smallest norm.

Example: Let $A = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}$, with $b = \begin{bmatrix} 9 \\ 3 \end{bmatrix}$. Then the full solution is:

$$\mathbf{x} = \begin{bmatrix} 9+2t \\ 3-t \\ t \end{bmatrix} \tag{A.1}$$

In Matlab, the result of typing `x=A\b` is $x = [0, -15/2, -9/2]^T$ and the result of typing `x=pinv(A)*b` is $x = [4, 11/2, -5/2]$.

(MATLAB HINT: You can get Matlab to return numbers as fractions by typing `format rat`)

In the case of an overdetermined system (a system with no solution), Matlab will automatically return the *least squares* solution- that is, the answer $\mathbf{x}^*$ will be the minimum of $\|A\mathbf{x} - \mathbf{b}\|$:

$$\|A\mathbf{x}^* - \mathbf{b}\| \leq \|A\mathbf{x} - \mathbf{b}\|, \text{ for all } \mathbf{x}$$

In general, you should always use the forward slash (for help, type `help slash`): `x=A\b` which automatically determines a best numerical method. That is, sometimes its best (numerically) not to explicitly compute the pseudoinverse first.

Exercise Set I

1. What is the Matlab command to create the array x which holds the integers: $2, 5, 8, 11, \dots 89$

2. (Referring to the array above) What would the Matlab command be that zeros out the even-numbered indices (That is, $x(2), x(4), x(6), \dots$)?

3. What is the difference in Matlab between typing: `x=[1 2 3]` and `x=[1,2,3]` and `x=[1;2;3]`? What happens if you type a semicolon at the end of the commands (i.e., `x=[1 2 3];`)?

4. (Referring to the last question) For each of those, what happens if you type `x.^2+3`? What happens if you forget the period (e.g., `x^2+3`)

5. What do the following commands do: `x=2;3;6;`, `x=2:3:6;`, `a=pi:pi:8*pi;`

6. Describe the output for each of the following Matlab commands. Recall that typing a semicolon at the end of the line suppresses Matlab output- to see the results, leave off the semicolon.

```
A=rand(3,4);
A([1,2],3)=zeros(2,1);
B=sin(A);
C=B+6;
D=2*B';
E=A./2;
F=sum(A.*A);
```

7. What will Matlab do if you type in:

```
A=rand(3,4);
A(:)
A(7)
```

   NOTE: This is very bad programming style! Don't do it unless you know what you're doing!!

8. What is the Matlab command to perform the following:

   (a) Given an array x, add 3 to each of its values.

   (b) Given an array A, remove its first column and assign the result to a new array B.

9. What will the following code fragment do?

```
a=1:10;
for k=1:10
  h=ceil(length(a)*rand);
  b(k)=a(h);
  a(h)=[];
end
```

   Compare this with `a=ceil(10*rand(10,1))` and `a=randperm(10)`

10. Use the Quick Summary sheet to help you write a code fragment that takes a random matrix X and re-sorts the columns so that the first column has the smallest size and the last column has the greatest size.

How do I get a Plot?

Here's a quick example to get us started:

```
x=linspace(-pi,3*pi,200);
y=sin(x);
plot(x,y);
```

You'll see that we had to create a domain array and a range array for the function. We then plot the arrays. For example,

```
plot([1,2],[3,4]);
```

will plot a line segment between the points $(1,3)$ and $(2,4)$. So, Matlab's plotting feature is drawing small line segments between data points in the plane.

Examples

1. Matlab can also plot multiple functions on one graph. For example:

   ```
   x1=linspace(-2,2);
   y1=sin(x1);
   y2=x1.^2;
   x2=linspace(-2,1);
   y3=exp(x2);
   plot(x1,y1,x1,y2,x2,y3);
   ```

   produces a single plot with all three functions.

2. `plot(x1,y1,'*-');`

   Plots the function `y1`, and also plots the symbol `*` where the data points are.

3. `plot(x1,y1,'k*-',x2,y3,'r^-');`

   Plots the function `y1` using a black (k) line with the asterisk at each data point, PLUS plots the function `y2` using a red line with red triangles at each data point.

   The following lists all of the built in colors and symbols that Matlab can use in plotting: (NOTE: You can see this list anytime in Matlab by typing: `help plot`)

Code	Color	Symbol	
y	yellow	.	point
m	magenta	o	circle
c	cyan	x	x-mark
r	red	+	plus
g	green	−	solid
b	blue	∗	star
w	white	:	dotted
k	black	−.	dashdot
		−−	dashed

4. The following sequence of commands also puts on a legend, a title, and relabels the $x-$ and $y-$axes: Try it!

```
x=linspace(-2,2);
y1=sin(x);
y2=x.^2;
plot(x,y1,'g*-',x,y2,'k-.');
title('Example One');
legend('The Sine Function','A Quadratic');
xlabel('Dollars');
ylabel('Sense');
```

5. Other Things: If you look at the plotting window from the last example, you'll see lots of things that you can do. For example, there's a zoom in and a zoom out feature. You can also edit the colors and symbols of your plot, and the title, legend and axis labels. Try them out!

Plotting in Three Dimensions

Matlab uses the `plot3` command to plot in three dimensions. We won't be using this feature here. To get more information, either type `help plot3` or refer to the Matlab Graphics Manual.

M-Files: Functions and Scripts

What is a Matlab Function? A Matlab function is a sequence of commands that uses some input variables and outputs some variables. The following is a very simple Matlab function:

```
function y=square(x)
%FUNCTION Y=SQUARE(X)
%This function inputs a number or an array, and
%  outputs the squares of the numbers.

y=x.^2;
```

You would type this in the editor, then save it as `square.m` (the filename must be the same name as the function, and it must use the `.m` extension).

You'll notice that the first line states "function". This is always the first line of a Matlab function. The remarks that follow the first line are very important. When you type `help square`, these three lines appear. So when you write your own functions, you should include comments so that you can remember how to use it.

The rest of the first line defines what the input variable is (x), and what the output variable is (y).

A Matlab function can produce multiple outputs. For example:

```
function [A,b]=randmatrix(n)
%FUNCTION [A,b]=RANDMATRIX(N)
%Produces an 2n x 2n random matrix A and a random
%column vector b.
nn=2*n;
A=rand(nn,nn);
b=rand(nn,1);
```

To call this function from Matlab, you would write, for example,

`[X,y]=randmatrix(10);`

You'll notice that after running this program, the variables internal to the function (in this case `nn`) disappear. This is one major difference between a *script* and a *function*:

- A **script file** is a text file with a sequence of Matlab commands. It is used by Matlab just as if you were typing the commands in from the keyboard. You should use a script file whenever you are experimenting in Matlab- it makes life a lot easier!
- A **function** in Matlab is like a subroutine in programming. That is, once the function has been called, all of its variables are local to that function- you cannot access them from the keyboard, and the variables are erased once the function is finished.

Both scripts and functions should have the `.m` file extension. We'll look at the difference in the Exercise set below.

Exercise Set II

1. The following script file is an example of Newton's method applied to a function $f(x) = xe^x - \cos(x)$. Recall that Newton's Method solves for x: $f(x) = 0$ by taking an initial guess, x_0, and refines the guess by:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

```
x=0.2;  %Initial guess for solution to f(x)=0
for k=1:5
  y=x*exp(x)-cos(x);
  dy=(x+1)*exp(x)+sin(x);
  x=x-(y/dy)
end
```

Notes about the code:

- We see our first `for` loop. We'll discuss what this does in class.
- Note the use of `%` to make comments.
- Note that `x=x-(y/dy)` does NOT have a semicolon at the end.

(a) Use the `edit` feature to type it in and save it as `newton1.m`
(b) Run the code after you've saved it by typing `newton1` in the command window.
(c) Write down Matlab's output.
(d) To see more significant digits, type `format long`
(e) Type `whos` and write down Matlab's answer.

Before continuing, clear the memory by typing `clear`

2. Now we'll change the script file into a Matlab function: Edit the file we created earlier as `newton1.m` so that it looks like this:

```
function [z,y,dy]=newton2(x,n)
%FUNCTION [y,dy]=newton2(x,n)
% performs Newton's Method on x*exp(x)-cos(x)
% using initial value x and n iterations.
% The output z gives the refined solution,
% the output y gives the function values and dy
% the corresponding derivatives.
```

```
for k=1:n
  y(k) = x*exp(x)-cos(x);
  dy(k) = (x+1)*exp(x)+sin(x);
  z(k) = x-(y(k)/dy(k));
  x = z(k);
end
```

Now save the file as `newton2.m`. In the command window, type `help newton2`. You should see the help lines come up. Now run the function by typing in the command window: `[x,y,z]=newton2(0.2,5);`

3. Let x be a row. What happens if you type `plot(x)`?

4. Let A be a 4×3 matrix. What happens if you type `plot(A)`? Compare this with `plot(A')`.

5. Write a Matlab command to plot $y = e^{5x}$, where $-2 \leq x \leq 2$ using 30 points. Plot both the curve and the actual data points themselves, both in magenta.

6. Write a Matlab function to plot $y = \sin(x)$ in red, $y = \sin(2x)$ in black, and $y = \sin(3x)$ in green, all on the same plot. You can assume that $x \in [-4, 8]$.

7. When we compute with numbers, some errors can occur. Try typing each of the following into Matlab, and see what happens:

   - `1/0` (Think about what this means before trying it!)
   - `-1/0`
   - `0/0`
   - `1/Inf`

   These give us some extensions to arithmetic- Matlab has ways of dealing with infinity (`Inf`) and "Not-A-Number" (`NaN`).

8. Try to reason out what you think Matlab will do with each of the following, then type it in and record what you get:

```
x=[1 3 2 1 3];
max(x)
find(x==max(x))
sort(x)
mean(x)
sum(x)
```

Appendix B

The Derivative

B.1 The Derivative of f

In this chapter, we give a short summary of the derivative. Specifically, we want to compare/contrast how the derivative appears for functions whose domain is $\mathbb{R}^n$ and whose range is $\mathbb{R}^m$, for any m, n. We begin by reviewing the definitions found in Calculus:

B.1.1 Mappings from $\mathbb{R}$ to $\mathbb{R}$

Definition: Let $f : \mathbb{R} \to \mathbb{R}$. Then define

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

We interpret this quantity as the slope of the tangent line at a given x, or as the velocity at time x. Given this definition, we can give a local linear approximation of a nonlinear function f at $x = a$:

$$L(x) = f(a) + f'(a)(x - a)$$

which is simply the equation of the tangent line to f at $x = a$. For comparison purposes, note that the graph of this function is in $\mathbb{R}^2$, and if $u = x - a, v = f(x) - f(a)$, this function behaves as the linear function $v = f'(a)u$.

Furthermore, we know the basic Taylor series expansion about $x = a$ is an extension of the linearization:

$$f(x) = f(a) + f'(a)(x - a) + \frac{1}{2} f''(a)(x - a)^2 + \ldots + \frac{f^{(k)}(a)}{k!}(x - a)^k + \ldots$$

We have also seen the derivative when f has some different forms:

B.1.2 Mappings from $\mathbb{R}$ to $\mathbb{R}^n$ (Parametrized Curves)

Definition: Let $\boldsymbol{f} : \mathbb{R} \to \mathbb{R}^n$ via:

$$\boldsymbol{f}(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \\ \vdots \\ f_n(t) \end{bmatrix} \quad \text{so that } \boldsymbol{f}'(t) = \begin{bmatrix} f_1'(t) \\ f_2'(t) \\ \vdots \\ f_n'(t) \end{bmatrix}$$

We normally think of the graph of f as a *parameterized curve*, and we differentiate (and integrate) componentwise. In this case, the linearization of f at $x = a$ is a matrix $(n \times 1)$ mapping:

$$L(x) = \boldsymbol{f}(a) + \boldsymbol{f}'(a)(x - a)$$

which takes a scalar x and maps it to a vector starting at $\boldsymbol{f}(a)$ and moves it in the direction of $\boldsymbol{f}'(a)$. The graph of this function lies in $\mathbb{R}^{n+1}$. If $u = x - a, v = \boldsymbol{f}(x) - \boldsymbol{f}(a)$, this function behaves like: $v = \boldsymbol{f}'(a)u$

In differential equations, we considered functions of this form when we looked at systems of differential equations. For example,

$$\dot{\boldsymbol{x}}(t) = A\boldsymbol{x}(t)$$

In this case, the origin is a critical point (fixed point), and we were able to classify the origin according to what the eigenvalues of A were (i.e., positive/negative, complex).

In the more general setting, we also considered the form: $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$ In this setting, $f : \mathbb{R}^n \to \mathbb{R}^n$, which we look at in the last section.

B.1.3 Mappings from $\mathbb{R}^n$ to $\mathbb{R}$: Surfaces

Definition: Let $f : \mathbb{R}^n \to \mathbb{R}$. Then the derivative in this case is the *gradient* of f:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

where

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{1}{h} f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)$$

and measures the rate of change in the direction of x_i. The linearization of f at $\boldsymbol{x} = \boldsymbol{a}$ is now a $1 \times n$ matrix mapping:

$$L(\boldsymbol{x}) = f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

The graph of this function lies in $\mathbb{R}^{n+1}$, and if $\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{a}, v = f(\boldsymbol{x}) - f(\boldsymbol{a})$, then this function behaves like: $v = \nabla f(\boldsymbol{a})u$.

We use the gradient to measure the rate of change of f in the direction of a unit-length vector u by computing the directional derivative of f at $\boldsymbol{a}$:

$$D_u f = \nabla f(\boldsymbol{a}) \cdot \boldsymbol{u}$$

In the exercises, you are asked to verify that the direction u of fastest increase is in the direction of the gradient.

Geometrically, suppose we are looking at the contours of a function, $y = f(x_1, \dots, x_n)$: That is, we plot $k = f(x_1, \dots, x_n)$ for different values of k. Since a contour line is where f is constant, the gradient in the direction of the contour is zero. On the other hand, a vector in the direction of the gradient is orthogonal to the contour, and is the direction of fastest increase.

For example, consider $f(x, y) = x^2 + 2y^2$. Its gradient is $\nabla f = [2x, 4y]$. At the point $x = 0.5, y = 0.5$, the gradient vector is $\nabla f(0.5, 0.5) = [1, 2]$. In Figure B.1, we plot several contours of f, including the contour going through the point $(0.5, 0.5)$. Next, we plot several unit vectors emanating from that point, alongside of which we show the numerical values of the corresponding directional derivatives.

From this, we verify that the direction of maximum increase is in the direction of the gradient, the gradient is orthogonal to the direction tangent to the contour, and the direction of fastest decrease is the negative of the gradient.

This particular class of functions is especially important to us, since:

- Learning can be thought of as the process of minimizing error.
- All error functions can be cast as functions from $\mathbb{R}^n$ to $\mathbb{R}$.

But, before going into the details, let us finish our comparisons of the derivative.

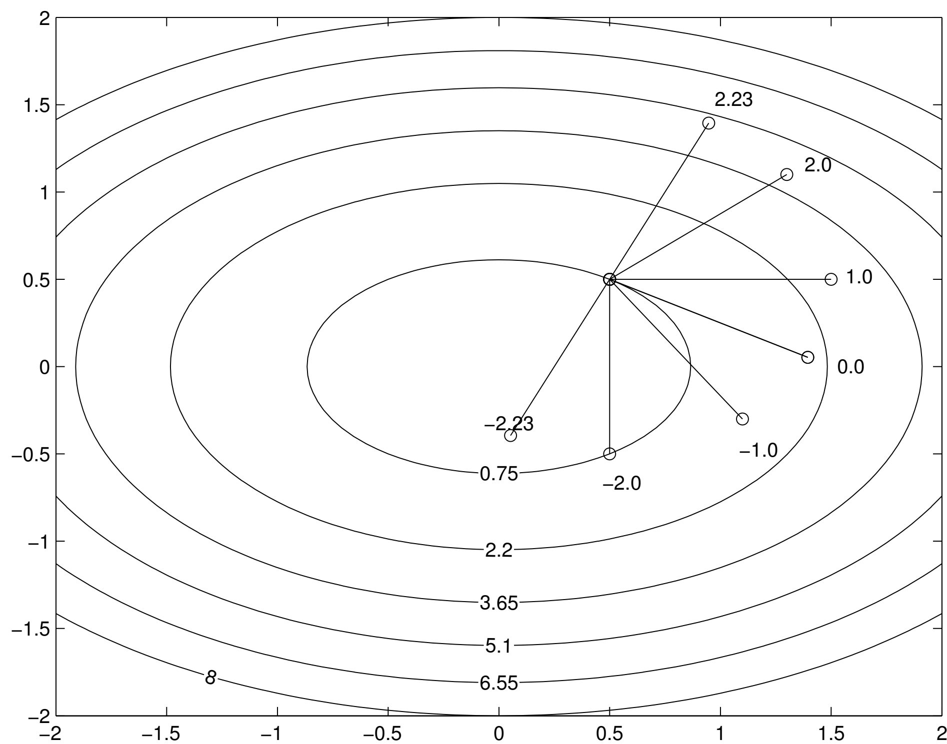


Figure B.1: The plot shows several contours of the function $f(x, y) = x^2 + 2y^2$, with the contour values listed vertically down the center of the plot. We also show several unit vectors emanating from the point $(0.5, 0.5)$, with their associated directional derivative values.

B.1.4 Mappings from $\mathbb{R}^n$ to $\mathbb{R}^m$

The last, most general, class of function is the function that goes from $\mathbb{R}^n$ to $\mathbb{R}^m$. Such a function can always be defined coordinate-wise:

Definition: If $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$, then f can be written as:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f^1(x_1, \ldots, x_n) \\ f^2(x_1, \ldots, x_n) \\ \vdots \\ f^m(x_1, \ldots, x_n) \end{bmatrix}$$

where each of the f^i are mapping $\mathbb{R}^n$ to $\mathbb{R}$. So, for example, a mapping of $\mathbb{R}^2$ to $\mathbb{R}^3$ might look like:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} x_1 + x_2 \\ \cos(x_1) \\ x_1 x_2 + \mathrm{e}^{x_1} \end{bmatrix}$$

In this case, the derivative of $\boldsymbol{f}$ has a special name: The Jacobian of $\boldsymbol{f}$:

Definition: Let $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$. Then the Jacobian of $\boldsymbol{f}$ at $\boldsymbol{x}$ is the $m \times n$ matrix:

$$Df = \begin{bmatrix} \nabla f^1 \\ \nabla f^2 \\ \vdots \\ \nabla f^m \end{bmatrix} = \begin{bmatrix} f^1_1 & f^1_2 & \ldots & f^1_n \\ f^2_1 & f^2_2 & \ldots & f^2_n \\ \vdots & & & \vdots \\ f^m_1 & f^m_2 & \ldots & f^m_n \end{bmatrix}$$

with $f^i_j = \frac{\partial f_i}{\partial x_j}$. You should look this over- it is consistent with all of our previous definitions of the derivative.

The linearization of $\boldsymbol{f}$ at $\boldsymbol{x} = \boldsymbol{a}$ is the affine map:

$$L(x) = \boldsymbol{f}(\boldsymbol{a}) + Df(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

The graph of this function is in $\mathbb{R}^{n+m}$, and if $\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{a}, v = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{a})$, then this function behaves like: $v = Df(\boldsymbol{a})u$.

B.2 Worked Examples:

Find the linearization of the given function at the given value of a:

1. $f(x) = 3x^2 + 4, x = 2$
2. $f(t) = (3t^2 + 4, \sin(t))^T, t = \pi$
3. $f(\boldsymbol{x}) = 3x_1 x_2 + x_1^2, \boldsymbol{x} = (0, 1)^T$
4. $f(\boldsymbol{x}) = (x_1 + x_2, \cos(x_1), x_1 x_2 + \mathrm{e}^{x_1})^T, \boldsymbol{x} = (0, 1)^T$

SOLUTIONS:

1. $f(2) = 16$, $f'(x) = 6x$, so $f'(2) = 12$. Thus,

$$L(x) = 16 + 12(x - 2)$$

Locally, this function is like: $v = 12u$.

2. $f(\pi) = (3\pi^2 + 4, 0)^T$, $f'(t) = (6t, \cos(t))^T$, $f'(\pi) = (6\pi, -1)^T$

$$L(x) = \begin{bmatrix} 3\pi^2 + 4 \\ 0 \end{bmatrix} + \begin{bmatrix} 6\pi \\ -1 \end{bmatrix} (x - \pi)$$

Locally, this means that f multiplies x by 6π in the first coordinate direction, and flips the second coordinate.

3. $f((0,1)^T) = 0$, $\nabla f = (3x_2 + 2x_1, 3x_1)$, $\nabla f((0,1)^T) = (3, 0)$

$$L(x) = 0 + (3, 0) \begin{bmatrix} x_1 - 0 \\ x_2 - 1 \end{bmatrix}$$

Locally (at $(0,1)^T$), this means that f is increasing in the direction parallel to x_1, and is constant in the direction parallel to x_2.

4.

$$f((0,1)^T) = (1,1,1)^T, Df = \begin{bmatrix} 1 & 1 \\ -\sin(x_1) & 0 \\ x_2 + \mathrm{e}^{x_1} & x_1 \end{bmatrix}, Df((0,1)^T) = \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 2 & 0 \end{bmatrix}$$

So the linearization is:

$$L(x) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 - 0 \\ x_2 - 1 \end{bmatrix}$$

B.3 Optimality

When we talk about "optimizing" a function, that function is necessarily a function from $\mathbb{R}^n$ to $\mathbb{R}$, since this is the only way that we can compare two outputs- that is, we need the range to be well-ordered. In this section, we will therefore concentrate on this class of functions.

In Calculus, we had a second derivative test for determining if we had a local maximum/minimum. That was:

> Let f be differentiable, and $f'(a) = 0$. If $f''(a) > 0$, then f has a local minimum at $x = a$. If $f''(a) < 0$, then f has a local maximum at $x = a$. If $f''(a) = 0$, this test is inconclusive.

We can do something similar for higher dimensions:

Definition: Let $f : \mathbb{R}^n \to R$. Then the Hessian of f is the $n \times n$ matrix:

$$Hf = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & & & \vdots \\ f_{n1} & f_{n2} & \cdots & f_{nn} \end{bmatrix}$$

where $f_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$, and so Hf is the matrix of all the second partial derivatives, and takes the place of the second derivative.

We might recall Clairaut's Theorem: If f_{ij}, f_{ji} are continuous in a ball containing $\boldsymbol{x} = \boldsymbol{a}$, then $f_{ij}(\boldsymbol{a}) = f_{ji}(\boldsymbol{a})$. In this case, the Hessian is a symmetric matrix at $\boldsymbol{x} = \boldsymbol{a}$. This will have some consequences that we'll discuss after The Spectral Theorem.

Example: If $f(x_1, x_2) = 3x_1^2 + 2x_1x_2 + x_2^2 - 3x_2 + 4$,

$$\nabla f = [6x_1 + 2x_2, 2x_1 + 2x_2 - 3]$$

and

$$Hf = \begin{bmatrix} 6 & 2 \\ 2 & 2 \end{bmatrix}$$

As in the case of the directional derivative, we can also check the second derivative in the direction of u by computing:

$$\frac{\boldsymbol{u}^T Hf(\boldsymbol{a})\boldsymbol{u}}{\boldsymbol{u}^T\boldsymbol{u}} \text{ or, if we use a unit vector, } \boldsymbol{u}^T Hf(\boldsymbol{a})\boldsymbol{u}$$

In Calc I, we said that, if a is a stationary point, then we can check to see if $f''(a) > 0$ to see if $x = a$ is a local minimum. How does that translate to the Hessian?

Definition: The matrix A is said to be positive definite if $\boldsymbol{x}^T A\boldsymbol{x} > 0$ for all $\boldsymbol{x} \neq 0$. A weaker statement is to say that A is positive semidefinite, where in this case, $\boldsymbol{x}^T A\boldsymbol{x} \geq 0$ for $\boldsymbol{x} \neq 0$. Compare this computation to was we said was the second derivative in the direction of $\boldsymbol{u}$.

Example: Any diagonal matrix with positive values along the diagonal is a positive definite matrix. As a specific example, consider:

$$[x, y] \begin{bmatrix} 2 & 0 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = [x, y] \begin{bmatrix} 2x \\ 5y \end{bmatrix} = 2x^2 + 5y^2$$

which is only zero if $x = y = 0$.

Example: If a matrix A has all positive eigenvalues, it is positive definite. We'll go through this in more detail in the section on linear algebra, but heuristically, you might imagine that the proof will use a diagonalization of A so that we're back to the previous example.

Let us consider the quadratic function in detail, as we will use this as a model for non-quadratic functions.

Let $F(x) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x} + c$, where A is symmetric and $\boldsymbol{b}, c$ are constants. Then

$$\nabla F(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$$

If we assume A is invertible, then the stationary point of F:

$$\boldsymbol{x} = -A^{-1}\boldsymbol{b}$$

and the Hessian does not depend on this value:

$$HF = A$$

So, if A is positive definite, the stationary point holds the unique minimum of F. Compare these computations to those we do in Calc I:

$$y = \frac{1}{2}ax^2 + bx + c$$

B.3.1 Necessary and Sufficient Conditions

Theorem: If f attains its local minimum (maximum) at $\boldsymbol{x} = \boldsymbol{a}$, then $\nabla f(\boldsymbol{a}) = \mathbf{0}$. The vector $\boldsymbol{a}$ is called a *stationary point* of f.

This is just the multidimensional version of Fermat's Theorem from Calculus I- we require all partial derivatives to vanish if we are at a local maximum or minimum.

Theorem: Necessary and Sufficient Conditions for Extrema

- First Order Condition: A necessary condition for $f : \mathbb{R}^n \to \mathbb{R}$ to have a local minimum (maximum) at $\boldsymbol{x} = \boldsymbol{a}$ is that $\boldsymbol{a}$ is a stationary point.
- Second Order Condition: Let $\boldsymbol{x} = \boldsymbol{a}$ be a stationary point. Then a necessary condition for f to have a local minimum (maximum) at $\boldsymbol{x} = \boldsymbol{a}$ is that $Hf(\boldsymbol{a})$ is positive semidefinite (negative semidefinite).

  A sufficient condition for a minimum (maximum) is that $Hf(\boldsymbol{a})$ is positive definite (negative definite).

We saw that the minimum was easy to compute if the function f was in a special form. We can force our function to have the quadratic form by appealing to the Taylor expansion.

Taylor's Theorem: Let $f : \mathbb{R}^n \to \mathbb{R}$ be analytic (That is, we'll assume all derivatives are continuous). Then the Taylor's series for f at $\boldsymbol{x} = \boldsymbol{a}$ is:

$$f(\boldsymbol{x}) = f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{a})^T Hf(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a}) + h.o.t.$$

where h.o.t. stands for "higher order terms". Recall that $\nabla f(\boldsymbol{a})$ is an $1 \times n$ vector, and $Hf(\boldsymbol{a})$ is an $n \times n$ symmetric matrix.

We will only compute Taylor's expansion up to second order, but for the sake of completeness (and your amusement!), we give the definition of the full Taylor expansion below.

In order to understand the notation, think about all the different partial derivatives we will have to compute. For example, if f depends on x, y, z, then there are 3 first partials, 9 second partials, 27 third partials, etc. Thus, we need a way of organizing all of these. Probably the best way to do this is with the following notation:

$$f_{i_1 i_2 \ldots i_k} = \frac{\partial^k f}{\partial x_{i_1} \partial x_{i_2} \ldots \partial x_{i_k}}$$

so the indices $i_1, i_2, \ldots, i_k$ are some permutation of the indices of $\boldsymbol{x}$ taken k at a time. Now we're ready to go!

The Taylor series expansion for f at $\boldsymbol{x} = \boldsymbol{a}$, where $\boldsymbol{x} \in \mathbb{R}^n$ is given by:

$$f(\boldsymbol{x}) = f(\boldsymbol{a}) + \sum_{i=1}^{n} f_i(\boldsymbol{a})(x_i - a_i) + \frac{1}{2!}\sum_{i_1=1}^{n}\sum_{i_2=1}^{n} f_{i_1 i_2}(\boldsymbol{a})(x_{i_1} - a_{i_1})(x_{i_2} - a_{i_2}) +$$

$$\ldots + \frac{1}{k!}\sum_{i_1=1}^{n} \ldots \sum_{i_k=1}^{n} f_{i_1 i_2 \ldots i_k}(\boldsymbol{a})(x_{i_1} - a_{i_1}) \cdots (x_{i_k} - a_{i_k}) + \ldots$$

B.4 Worked Examples

1. Compute the (full) Taylor series expansion for

   $$f(x, y) = x^2 + 2xy + 3y^2 + xy^2$$

   at $(1, 1)$.
2. Suppose we have a function f so that $f((0, 1)) = 3$, and $\nabla f((0, 1)) = [1, -2]$. Use the linear approximation to f to get an approximation to $f((0.5, 1.5))$.

3. Let $f(x, y) = xy + y^2$. At the point $(2, 1)$, in which direction is f increasing the fastest? How fast is it changing?

4. Suppose $A = \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix}$. Show that A is neither positive definite or negative definite by finding an $\boldsymbol{x}$ so that $\boldsymbol{x}^T A\boldsymbol{x} > 0$, and another $\boldsymbol{x}$ so that $\boldsymbol{x}^T A\boldsymbol{x} < 0$.

SOLUTIONS TO THE WORKED EXAMPLES:

1. First we see that f is a third degree polynomial (because of the xy^2 term). Therefore, we will need to compute the partial derivatives up to the third partials:

$$f_x = 2x + 2y + y^2, \quad f_y = 2x + 6y + 2xy$$

$$\text{so at } x = 1, y = 1 : f_x = 5, \quad f_y = 10$$

$$f_{xx} = 2, f_{xy} = f_{yx} = 2 + 2y = 4, f_{yy} = 6 + 2x = 8$$

$$f_{xxy} = 0 \quad f_{xxx} = 0 \quad f_{xyx} = 0 \quad f_{xyy} = 2$$

$$f_{yxx} = 0 \quad f_{yxy} = 2 \quad f_{yyx} = 2 \quad f_{yyy} = 0$$

Now we get that:

$$f(x) = 7 + 5(x-1) + 10(y-1) + \frac{2}{2}(x-1)^2 + \frac{8}{2}(x-1)(y-1) +$$
$$\frac{8}{2}(y-1)^2 + \frac{6}{6}(x-1)(y-1)^2$$

Note that the coefficient for $(x-1)(y-1)$ came from both f_{xy} and f_{yx}, and Note that the $(x-1)(y-1)^2$ coefficient came from all three partials: f_{xyy}, f_{yxy} and f_{yyx}.

2. The linearization of f:

$$f(x, y) = 3 + [1, -2] \begin{bmatrix} x \\ y \end{bmatrix} = 3 + x - 2y$$

so $f(0.5, 1.5)$ is approximately $3 + 0.5 - 3 = 0.5$.

3. The gradient is $[y, x + 2y]$, so at $(2, 1)$, we get $[1, 4]$. The direction in which f is increasing the fastest is in the direction of $[1, 4]^T$, and the rate of change in that direction is $\sqrt{17}$.

4. We see that $\boldsymbol{x}^T A\boldsymbol{x} = 2x^2 + xy - y^2$, so we can choose, for example, $x = 0, y = 1$ for a negative value, and $x = 1, y = 0$ for a positive.

B.5 Exercises

1. Let $f(x, y) = x^2y + 3y$. Compute the linearization of f at $x = 1, y = 1$.

2. Let

$$f(x, y, z, w) = \begin{bmatrix} x + zw - w^2 \\ 4 - 3xy - z^2 \\ e^{xyzw} \\ x^2 + y^2 - 2xyw \end{bmatrix}$$

Compute the linearization of f at $(1, 2, 3, 0)^T$.

3. Definition: Let $f : \mathbb{R}^n \to \mathbb{R}$. The Directional Derivative of f in the direction of the vector $\boldsymbol{u}$ is given by:

$$D_{\boldsymbol{u}} f = \nabla f \cdot \boldsymbol{u}$$

which is the dot product of the gradient with the vector $\boldsymbol{u}$. We interpret this quantity (which is a scalar) as "The rate of change of f in the direction of $\boldsymbol{u}$".

   (a) What does the directional derivative give if $\boldsymbol{u} = \boldsymbol{e}^{(i)}$, the ith standard basis vector?

   (b) Consider f at some fixed value, $x = a$. Over all possible *unit* vectors $\boldsymbol{u}$, in which direction will f have it's maximum (minimum) rate of change? Hint: Think about how we can write the dot product in terms of $\cos(\theta)$, for an appropriate θ.

4. Let $\boldsymbol{x}$ be the variable, $A, \boldsymbol{b}$ be a constant matrix, vector respectively. Show that:

   (a) $D\boldsymbol{x} = I$

   (b) $\nabla \left(\boldsymbol{x}^T \boldsymbol{x}\right) = 2\boldsymbol{x}$

   (c) $\nabla \left(\boldsymbol{b}^T \boldsymbol{x}\right) = \boldsymbol{b}$

   (d) $D(A\boldsymbol{x}) = A$

5. Show that, if $\boldsymbol{u}, \boldsymbol{v}$ are vector-valued functions from $\mathbb{R}^n \to \mathbb{R}^n$, then:

$$\nabla \left(\boldsymbol{u}^T \boldsymbol{v}\right) = (D\boldsymbol{u})^T \boldsymbol{v} + (D\boldsymbol{v})^T \boldsymbol{u}$$

6. Use the previous exercises to show that, if G is a symmetric matrix, then $\nabla \left(\frac{1}{2}\boldsymbol{x}^T G\boldsymbol{x}\right) = G\boldsymbol{x}$.

7. Multiply the following expression out to its scalar representation:

$$[x, y] \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

   Use this result to write: $5x^2 - xy + 6y^2$ as $\boldsymbol{x}^T A\boldsymbol{x}$, for a suitable matrix A.

8. Let $f(x, y) = 3x^2 + xy - y^2 + 3x - 5y$.

   (a) At the point $(1, 1)$, in which direction is f increasing the fastest? Decreasing the fastest? What is its rate of change in the fastest direction?

   (b) Compute the Hessian of f.

   (c) Rewrite f in the form: $\frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x}$, for a suitable matrix A and vector $\boldsymbol{b}$.

   (d) Find the stationary point of f using our previous formula, and verify that the gradient is zero there. Show that we do not have a minimum or a maximum at the stationary point.

Appendix C

Optimization

Appendix D

Matlab and Radial Basis Functions

The purpose of this chapter is to demonstrate the Matlab[1] implementation of the radial basis function solver. This will be worthwhile in two respects. We first see how the orthogonal least squares algorithm is extended to the multidimensional case, and also will learn some of Matlab's vectorization techniques.

If we have p centers, m points of dimension n as input data, and m points of dimension k for output data, then the training the RBF network is equivalent to solving the following equation for $\boldsymbol{A}$:

$$\Phi \boldsymbol{A} = \boldsymbol{Y} \tag{D.1}$$

where $\Phi_{i,j}$ is the ϕ acting on the ith data point using the jth center:

$$\Phi_{i,j} = \phi(\|\boldsymbol{x}^i - \boldsymbol{c}_j\|)$$

so that Φ has dimensions m by p.

The matrix values in A are the unknown coefficients, specifically, $A_{i,j}$ is the coefficient for the ith center, j^{th} dimension, α_i^j, so that A has dimensions p by k.

Finally, $\boldsymbol{Y}$ is the target matrix, with dimensions m by k.

The Solution to the RBF Equation

Given the transfer function ϕ and a list of centers, $\boldsymbol{c}_i$, then the solution to Equation (D.1) is:

$$\boldsymbol{A} = \Phi^{+} \boldsymbol{Y}$$

where Φ^{+} is the Moore-Penrose pseudoinverse. There are several methods available to solve for the pseudoinverse of a matrix. Two methods implementable in Matlab are given below:

1. The "pinv" command:
$$\Phi^{+} = \text{pinv}(\Phi)$$
2. The Singular Value Decomposition, Case I: Φ is full rank.

   Let Φ be m by p. Then the reduced singular value decomposition of Φ is given by:
$$\Phi = USV^{\text{T}}$$
   which in Matlab is produced by the command:
$$[U, S, V] = \text{svd}(\Phi, 0)$$

[1]Matlab is the property of Mathworks, Inc. The algorithms presented here are written by Mark Beale and are copyrighted by Mathworks.

where U is an m by p unitary matrix ($U^{\mathrm{T}}U = I$), V is an p by p unitary matrix, and S is an p by p diagonal matrix:

$$S = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ 0 & \vdots & \dots & \vdots \\ 0 & \dots & & \sigma_p \end{pmatrix} \qquad S^{-1} = \begin{pmatrix} \frac{1}{\sigma_1} & 0 & \dots & 0 \\ 0 & \frac{1}{\sigma_2} & \dots & 0 \\ 0 & \vdots & \dots & \vdots \\ 0 & \dots & & \frac{1}{\sigma_p} \end{pmatrix}$$

The pseudoinverse of Φ is then:

$$\Phi^+ = VS^{-1}U^{\mathrm{T}}$$

In this case, either "pinv" or this SVD routine gives adequate numerical values. The special case is when Φ is NOT full rank.

3. Singular Value Decomposition, Case II: Φ is not full rank.

   In this case, S only has k non-zero singular values (or in practice, only k values above a "noise floor"). We note that the Matlab SVD routine returns singular values in descending order, so that the largest singular value is σ_1. Therefore, S has a diagonal k by k submatrix, and the rest of the matrix should be set to 0.

   If Φ is not full rank, then the "pinv" command has a default to choose the rank. This is why the singular value decomposition is sometimes preferable; you directly choose the values for which $\sigma_i = 0$. Once the noise floor cutoff has been determined, set k equal to the number of singular values above the cutoff. Then the psuedo inverse is given by (I'm using Matlab notation here):

$$\Phi^+ = V(:, 1:k) * S^{-1}(1:k, 1:k) * U(:, 1:k)'$$

So the idea is that once the transfer function is chosen and the appropriate centers are found, then training the Radial Basis Function Network is just computing the pseudoinverse Φ.

Current Matlab Implementation

The following discusses the suite of commands that Matlab currently uses to solve the interpolation problem given the centers. Note that this is *not* the solverb routine, which finds the centers.

First, the Matlab implementation is transposed from our original setup. That is,

$$\Phi\tilde{\boldsymbol{A}} = \tilde{\boldsymbol{Y}} \Rightarrow \mathtt{AP} = \mathtt{Y}$$

Furthermore, the interpolation matrix is appended with a column (for Φ, or a row for P):

$$\Phi \Rightarrow \left[\Phi | \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \right]$$

In Matlab, let M by N be the size of P:

```
P = [P;ones(1,N)]
```

and the matrix A is split into two pieces:

```
A=[w2 b2]
```

The following commands mimic the approximation to Y:

```
a2 = ( Y / P) * P;
```

In solverb, we see the following suite of commands:

- Construct P using radbas. The constant b is for the Gaussian:

  ```
  P=radbas(dist(p',p)*b)
  ```

- Find the matrix A:

  ```
  [w2,b2]=solvelin(P,Y);
  ```

- Construct the approximation to Y:

  ```
  a2=purelin(w2*P,b2);
  ```

We now discuss the issues associated with Radial Basis Function interpolation: Finding a suitable transfer function and finding suitable centers.

Choosing the Centers

In a lot of research work, centers are chosen randomly from the data. It is also possible to perform clustering on the data, then choose centers from the clusters (a.k.a. codebook vectors). This method is not optimized for this purpose, however, and can be computationally expensive.

For many applications, randomly choosing the centers may be sufficient. In what follows, we apply a method known as Orthogonal Least Squares to choose the centers from the data. This algorithm is implemented in the Matlab routine "solverb", which is also discussed below.

The Matlab Implementation

We now discuss the Matlab implementation of this algorithm. Keep in mind that the default number of centers in "solverb" is the number of data points. This can cause some problems since the first iterate of the interpolation matrix has dimensions *Number of points* by *Number of points*!

Again, denote the interpolation matrix by its columns:

$$\Phi = [\Phi_1, \ldots, \Phi_p], \quad \Phi_i \in \mathbb{R}^n$$

which in Matlab will be denoted by P, and the output matrix, $\boldsymbol{Y}$ in terms of its rows:

$$\boldsymbol{Y}^{\mathrm{T}} = \begin{pmatrix} y_1^{\mathrm{T}} \\ \vdots \\ y_n^{\mathrm{T}} \end{pmatrix} = [\boldsymbol{d}_1, \ldots, \boldsymbol{d}_k] = \boldsymbol{d}$$

So that $y_i \in \mathbb{R}^k$, and $\boldsymbol{d}_i \in \mathbb{R}^n$.

We are now ready to construct the algorithm. Key is the computation of error:

$$\frac{(\Phi_k^{\mathrm{T}} \boldsymbol{d}_j)^2}{\boldsymbol{d}_j^{\mathrm{T}} \boldsymbol{d}_j \Phi_k^{\mathrm{T}} \Phi_k} \tag{D.2}$$

which again is called the error associated with using the kth column of Φ for the jth data point.

Remark. First, we construct the matrix of elements for the denominator:

Let

$$\mathtt{PP} = \begin{pmatrix} \Phi_1^{\mathrm{T}}\Phi_1 \\ \vdots \\ \Phi_m^{\mathrm{T}}\Phi_m \end{pmatrix} = \mathtt{sum}(\mathtt{P}.*\mathtt{P}')'$$

and

$$\mathtt{dd} = \begin{pmatrix} \boldsymbol{d}_1^{\mathrm{T}}\boldsymbol{d}_1 \\ \vdots \\ \boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{d}_k \end{pmatrix} = \mathtt{sum}(\mathtt{d}.*\mathtt{d})'$$

Now the matrix of elements needed for the denominators of the error calculation in Equation (D.2) is:

$$\mathtt{dd}*\mathtt{PP}' = \begin{pmatrix} \boldsymbol{d}_1^{\mathrm{T}}\boldsymbol{d}_1 \\ \vdots \\ \boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{d}_k \end{pmatrix} \left(\Phi_1^{\mathrm{T}}\Phi_1 \dots \Phi_m^{\mathrm{T}}\Phi_m\right)$$

The $(j,k)^{\mathrm{th}}$ element of this matrix is given by:

$$\boldsymbol{d}_j^{\mathrm{T}}\boldsymbol{d}_j\Phi_k^{\mathrm{T}}\Phi_k$$

Remark. We now construct a matrix that has the numerator elements from Equation (D.2).

First, note that:

$$\begin{pmatrix} \Phi_1^{\mathrm{T}} \\ \vdots \\ \Phi_m^{\mathrm{T}} \end{pmatrix} (\boldsymbol{d}_1,\dots,\boldsymbol{d}_k) = \Phi^{\mathrm{T}}\boldsymbol{d} = \mathtt{P}'*\mathtt{d}$$

so that, in Matlab, the $(j,k)^{\mathrm{th}}$ component of

```
((P'*d)'.^2)
```

is the numerator we needed:

$$\left(\Phi_k^{\mathrm{T}}d_j\right)^2$$

In conclusion, we construct the error value in Equation (D.2) by computing:

```
( (P'*d)'.^2 ) ./ (dd * PP')
```

Other Matlab "Tricks"

1. Truncate the Matrix X by removing the kth column:

   ```
   X(:,k)=[];
   ```

2. This also works to remove elements from a vector:

   ```
   x(k)=[];
   ```

3. Increase the Matrix X by adding the kth column of Y:

```
X=[X, Y(:,k)];
```

4. And the vector version of adding an element of y to a vector x:

```
x=[x y(k)]
```

5. Construct a matrix X by using the odd columns (1-5) of Y:

```
I=1:2:5
X=Y(:,I);
```

6. For projections, if we would like to remove the component along a vector w_j from the column space of matrix P:

```
a=wj'*P/(wj'*wj);
P=P-wj*a;
```

Part IV

Bibliography

Bibliography

[1] Pierre Baldi, Yves Chauvin, and Kurt Hornik. Back-propagation and unsupervised learning in neural networks. In Yves Chauvin and David E. Rumelhart, editors, *Backpropagation: Theory, Architectures and Applications*, Developments in Connections Theory, chapter 4, pages 99–135. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1995.

[2] Keith Ball. Eigenvalues of euclidean distance matrices. *Journal of Approximation Theory*, 68(1):74–82, 1992.

[3] Andrew R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Neural Networks*, 39(3):930–945, 1993.

[4] Andrew R. Barron. Approximation and estimation bounds for artificial neural networks. *Machine Learning*, 14:115–133, 1994.

[5] D.S. Broomhead and David Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

[6] G.S. Brown. Point density in stems per acre. *New Zealand Forestry Research Notes*, (38), 1965.

[7] S. Chen, C.F.N. Cowan, and P.M. Grant. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks*, 2(2):302–309, 1991.

[8] Visionics Corporation. Faceit technology. http://www.visionics.com/faceit/.

[9] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control Signals Systems*, 2:303–314, 1989.

[10] M.N. Dailey, G.W. Cottrell, and T.A. Busey. Eigenfaces for familiarity. In *Proceedings of the Twentieth annual conference of the Cognitive Science Society*, pages 273–278. Erlbaum, 1998.

[11] R. Fadeley. Oregon malignancy pattern physiographically related to hanford, washington, radioisotope storage. *Journal of Environmental Health*, 27(6):883–897, June 1965.

[12] K. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183–191, 1989.

[13] K. Funahashi. Approximate realization of identity mappings by three layer neural networks. *Electronics and Communications in Japan, Part 3*, 73(11):61–68, 1990.

[14] Robert V. Hogg and Allen T. Craig. *Introduction to Mathematical Statistics*. Macmillan Publishing, 1978.

[15] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:251–257, 1991.

[16] Kurt Hornik. Some new results on neural network approximation. *Neural Networks*, 6:1069–1072, 1993.

[17] Kurt Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[18] Kurt Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3:551–560, 1989.

[19] D. Hundley, M. Kirby, and R. Miranda. Spherical nodes in neural networks with applications. In S.H. Dagli, B.R. Fernandez, J. Ghosh, and R.T. Soundar Kumara, editors, *Intelligent Engineering Through Artificial Neural Networks*, number 5. The American Society of Mechanical Engineers, 1995.

[20] M. Kirby and R. Miranda. Circular nodes in neural networks. *Neural Computation*, 8(2):390–402, 1996.

[21] M. Kirby and R. Miranda. Empirical dynamical systems reduction Part I: Global nonlinear transformations. In K. Coughlin, editor, *Semi-analytic methods for the Navier-Stokes Equations*, volume 20 of *CRM Proceedings and Lecture Notes*. American Mathematical Society, Providence, RI, 1999.

[22] M. Kirby and L. Sirovich. Application of the Karhunen-Loève procedure for the characterization of human faces. *IEEE trans. PAMI*, 12(1):103., 1990.

[23] Michael Kirby. *Geometric Data Analysis: An empirical approach to dimensionality reduction and the study of patterns*. John Wiley and Son, New York, 2001.

[24] T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, Berlin, 1995.

[25] Peter Kruizinga. Face recognition home page. http://www.cs.rug.nl/~peterkr/FACE/face.html.

[26] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantization design. *IEEE transactions on Communications*, 28(1):84–95, January 1980.

[27] T. Martinetz and K. Schulten. A "neural-gas" network learns topologies. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 397–402. Elsevier Science Publishers, 1991.

[28] T. Martinetz and K. Schulten. Topology representing networks. *Neural Networks*, 7(3):507–522, 1994.

[29] Thomas M. Martinetz, Stanislav G. Berkovich, and Kluas J. Schulten. "neural-gas" network for vector quantization and its application to time-series prediction. *IEEE Transactions on Neural Networks*, 4(4):558–569, 1993.

[30] Charles A. Micchelli. Interpolation of scattered data: Distance matrices and conditionally positive definite functions. *Constr. Approx.*, 2:11–22, 1986.

[31] MIT. Mit artificial intelligence lab. http://www.ai.mit.edu.

[32] E. Oja. Data compression, feature extraction, and autoassociation in feedforward neural networks. In T. Kohonen, K. Mäkisara., O. Simula, and J. Kangas, editors, *"Artificial Neural Networks"*, pages 737–745, NY, 1991. Elsevier Science Publishers.

[33] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, England, 1989.

[34] L. Sirovich and M. Kirby. A low-dimensional procedure for the characterization of human faces. *J. of the Optical Society of America A*, 4:529., 1987.

[35] Viisage Technology. Face recognition software. http://www.viisage.com/facialrecog.htm.

[36] Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

Part V

Index

Index

Best Approximation Theorem, 44

centers
  RBF, 144
chromosomes, 25
correlation coefficient, 21
covariance
  between two sets, 20

eigenvalue
  EDM, 142
Euclidean Distance Matrix (EDM), 140

fundamental subspaces, 35, 41, 50

Genetic Algorithm, 25

hyperbolic tangent, 156

interpolation, 135

log sigmoidal, 156
low dimensional representation, 36

mean
  discrete one-dimensional, 17
multidimensional scaling, 142

prestate, 156
Pythagorean Theorem, 43

radial basis function, 143
rank, 42
regression, 135
rule of thumb
  Gaussian width, 148

saturation, 157
Singular Value Decomposition, 48
  Reduced, 51
singular vectors
  left, 48
  right, 48
Spectral Decomposition, 46
Spectral Theorem, 46
SVD, 35

testing set, 135
training set, 135
transfer function, 156
  RBF, 143
transfer matrix
  RBF, 143

Vandermonde matrix, 136, 137
variance, 20