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

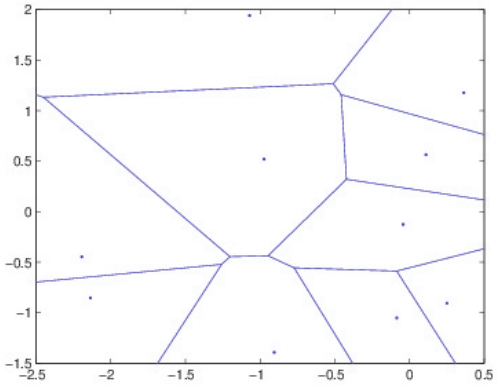
Given data in matrix X and class labels in $\boldsymbol{t}$, the class label for a new point $\hat{\boldsymbol{x}}$ is given by the class label of the point in X to which it is closest. That is, if i^* is the index of the point in X closest to $\hat{\boldsymbol{x}}$:

$$i^* = \min_i \|X(:, i) - \hat{\boldsymbol{x}}\|$$

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.
```

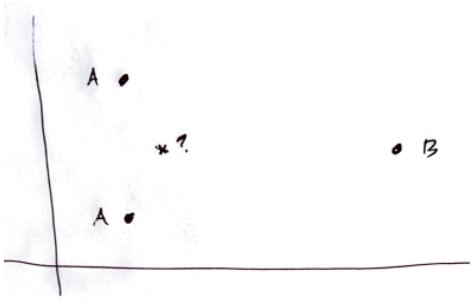
In Python, we'll have something similar.

```
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.

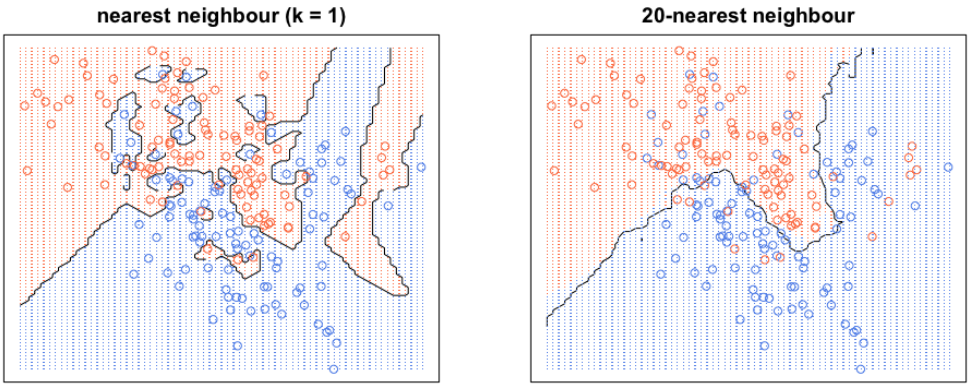
Extreme Cases

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

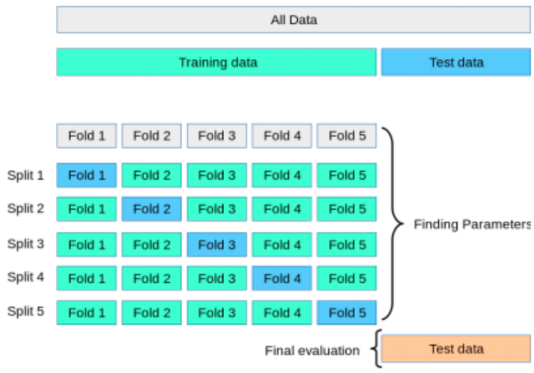
To begin with, suppose we have split our data into a **training set** and a **test set**. A common percentage might be 70-30, although this is problem dependent.

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 i^{th} 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:

$$f(\hat{\boldsymbol{x}}) = \sum_{i=1}^{k} \alpha_i t_i$$

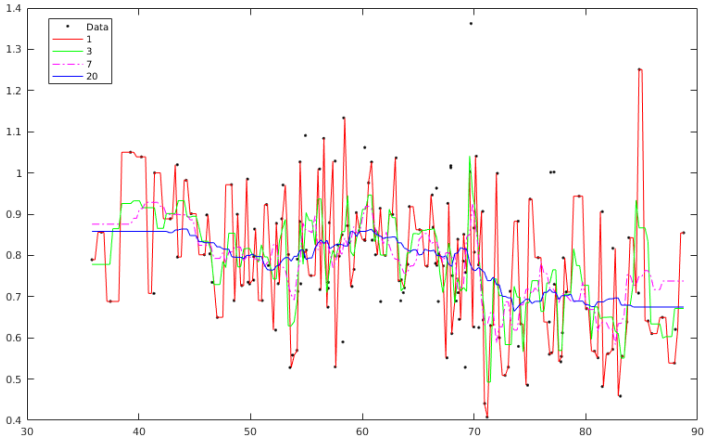
This technique could be extended to functions of more than one variable. For example, if the known target values are vectors $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p$, then we can compute the output just as before:

$$f(\hat{\boldsymbol{x}}) = \sum_{i=1}^{k} \alpha_i \boldsymbol{t}_i$$

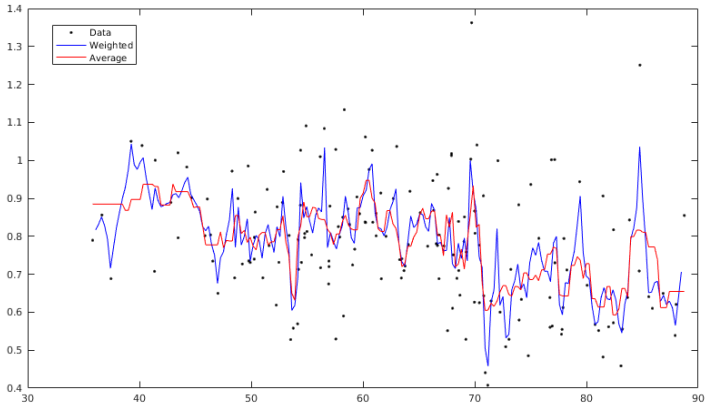
This kind of a linear combination (with non-negative weights that sum to 1) has a special name- This is called a **convex combination** of the vectors $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p$ (where the weights for the targets outside of k points are set to zero).

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.