Lab 3 - K-Nearest Neighbors in Python

February 8, 2016

This lab on K-Nearest Neighbors is a python adaptation of p. 163-167 of "Introduction to Statistical Learning with Applications in R" by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. Originally adapted by Jordi Warmenhoven (github.com/JWarmenhoven/ISLR-python), modified by R. Jordan Crouser at Smith College for SDS293: Machine Learning (Spring 2016).

```
In [ ]: import pandas as pd
        import numpy as np
```

1 4.6.5: K-Nearest Neighbors

In this lab, we will perform KNN on the `Smarket` dataset from `ISLR`. This data set consists of percentage returns for the S&P 500 stock index over 1,250 days, from the beginning of 2001 until the end of 2005. For each date, we have recorded the percentage returns for each of the five previous trading days, `Lag1` through `Lag5`. We have also recorded `Volume` (the number of shares traded on the previous day, in billions), `Today` (the percentage return on the date in question) and `Direction` (whether the market was `Up` or `Down` on this date). We can use the `head`(...) function to look at the first few rows:

```
In [ ]: df = pd.read_csv('Smarket.csv', usecols=range(1,10), index_col=0, parse_dates=True)
        df.head()
```

Today we're going to try to predict `Direction` using percentage returns from the previous two days (`Lag1` and `Lag2`). We'll build our model using the `KNeighborsClassifier`() function, which is part of the `neighbors` submodule of SciKitLearn (`sklearn`). We'll also grab a couple of useful tools from the `metrics` submodule:

```
In [ ]: from sklearn import neighbors
        from sklearn.metrics import confusion_matrix, classification_report
```

This function works rather differently from the other model-fitting functions that we have encountered thus far. Rather than a two-step approach in which we first fit the model and then we use the model to make predictions, `knn`() forms predictions using a single command. The function requires four inputs. 1. A matrix containing the predictors associated with the training data, labeled `X_train` below. 2. A matrix containing the predictors associated with the data for which we wish to make predictions, labeled `X_test` below. 3. A vector containing the class labels for the training observations, labeled `y_train` below. 4. A value for K, the number of nearest neighbors to be used by the classifier.

We'll first create a vector corresponding to the observations from 2001 through 2004, which we'll use to train the model. We will then use this vector to create a held out data set of observations from 2005 on which we will test. We'll also pull out our training and test labels.

```
In [ ]: X_train = df[:'2004'][['Lag1','Lag2']]
        y_train = df[:'2004']['Direction']

        X_test = df['2005':][['Lag1','Lag2']]
        y_test = df['2005':]['Direction']
```

Now the `neighbors.KNeighborsClassifier()` function can be used to predict the market's movement for the dates in 2005.

```
In [ ]: knn = neighbors.KNeighborsClassifier(n_neighbors=1)
        pred = knn.fit(X_train, y_train).predict(X_test)
```

The `confusion_matrix()` function can be used to produce a **confusion matrix** in order to determine how many observations were correctly or incorrectly classified. The `classification_report()` function gives us some summary statistics on the classifier's performance:

```
In [ ]: print(confusion_matrix(y_test, pred).T)
        print(classification_report(y_test, pred, digits=3))
```

The results using $K = 1$ are not very good, since only 50% of the observations are correctly predicted. Of course, it may be that $K = 1$ results in an overly flexible fit to the data. Below, we repeat the analysis using $K = 3$.

```
In [ ]: knn = neighbors.KNeighborsClassifier(n_neighbors=3)
        pred = knn.fit(X_train, y_train).predict(X_test)
        print(confusion_matrix(y_test, pred).T)
        print(classification_report(y_test, pred, digits=3))
```

The results have improved slightly. Try looping through a few other K values to see if you can get any further improvement:

```
In [ ]: for k_val in range(10):
            # Your code here
```

It looks like for classifying this dataset, KNN might not be the right approach.

2 4.6.6: An Application to Caravan Insurance Data

Let's see how the `KNN` approach performs on the `Caravan` data set, which is part of the `ISLR` library. This data set includes 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is `Purchase`, which indicates whether or not a given individual purchases a caravan insurance policy. In this data set, only 6% of people purchased caravan insurance.

```
In [ ]: df2 = pd.read_csv('Caravan.csv')
        df2["Purchase"].value_counts()
```

Because the `KNN` classifier predicts the class of a given test observation by identifying the observations that are nearest to it, the scale of the variables matters. Any variables that are on a large scale will have a much larger effect on the distance between the observations, and hence on the `KNN` classifier, than variables that are on a small scale.

For instance, imagine a data set that contains two variables, salary and age (measured in dollars and years, respectively). As far as `KNN` is concerned, a difference of $1,000 in salary is enormous compared to a difference of 50 years in age. Consequently, salary will drive the `KNN` classification results, and age will have almost no effect.

This is contrary to our intuition that a salary difference of $1,000 is quite small compared to an age difference of 50 years. Furthermore, the importance of scale to the `KNN` classifier leads to another issue: if we measured salary in Japanese yen, or if we measured age in minutes, then we'd get quite different classification results from what we get if these two variables are measured in dollars and years.

A good way to handle this problem is to **standardize** the data so that all variables are given a mean of zero and a standard deviation of one. Then all variables will be on a comparable scale. The `scale()` function from the `preprocessing` submodule of SciKitLearn does just this. In standardizing the data, we exclude column 86, because that is the qualitative `Purchase` variable.

```
In [ ]: from sklearn import preprocessing
        y = df2.Purchase
        X = df2.drop('Purchase', axis=1).astype('float64')
        X_scaled = preprocessing.scale(X)
        print(np.std(X_scaled))
```

Now every column of `X_scaled` has a standard deviation of one and a mean of zero.

We'll now split the observations into a test set, containing the first 1,000 observations, and a training set, containing the remaining observations.

```
In [ ]: X_train = X_scaled[1000:,:]
        y_train = y[1000:]
        X_test = X_scaled[:1000,:]
        y_test = y[:1000]
```

Let's fit a `KNN` model on the training data using $K = 1$, and evaluate its performance on the test data.

```
In [ ]: knn = neighbors.KNeighborsClassifier(n_neighbors=1)
        pred = knn.fit(X_train, y_train).predict(X_test)
        print(classification_report(y_test, pred, digits=3))
```

The KNN error rate on the 1,000 test observations is just under 12%. At first glance, this may appear to be fairly good. However, since only 6% of customers purchased insurance, we could get the error rate down to 6% by always predicting `No` regardless of the values of the predictors!

Suppose that there is some non-trivial cost to trying to sell insurance to a given individual. For instance, perhaps a salesperson must visit each potential customer. If the company tries to sell insurance to a random selection of customers, then the success rate will be only 6%, which may be far too low given the costs involved.

Instead, the company would like to try to sell insurance only to customers who are likely to buy it. So the overall error rate is not of interest. Instead, the fraction of individuals that are correctly predicted to buy insurance is of interest.

It turns out that `KNN` with $K = 1$ does far better than random guessing among the customers that are predicted to buy insurance:

```
In [ ]: print(confusion_matrix(y_test, pred).T)
```

Among 77 such customers, 9, or 11.7%, actually do purchase insurance. This is double the rate that one would obtain from random guessing. Let's see if increasing K helps! Try out a few different K values below. Feeling adventurous? Write a function that figures out the best value for K.

```
In [ ]: # Your code here
```

It appears that `KNN` is finding some real patterns in a difficult data set! To get credit for this lab, post a response to the Piazza prompt available at: https://piazza.com/class/igwiv4w3ctb6rg?cid=10