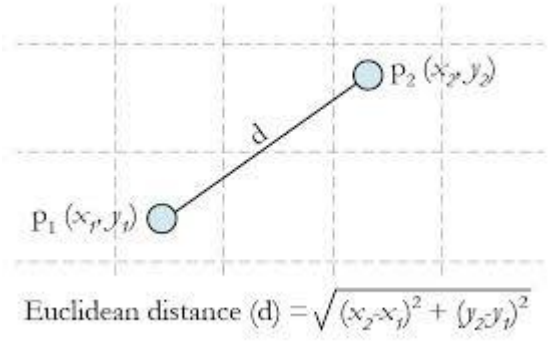
Data Mining K-Nearest Neighbour Coursework

Cross Validation

To find the best value of K for the "Adult Census" data I used cross validation. Cross validation is the process of splitting up the training data into folds where one-fold is tested against all others. Then once all folds have been tested the average error is taken. The data was split into 5 folds, then when testing one-fold would be evaluated with the other 4 folds. The folds were chosen using the folds column inside the "adult.train.5fold.csv"

Once the data was split into folds, I started the K-Nearest Neighbour algorithm. The first step was finding the Euclidean distances between the input and all the data in the other folds. In the categorical/discrete columns, if the column variable from the input instance didn't match the training instance, I gave it a distance of 1. I did this because we do not know for example if a certain "native country" is more different to some than others.

To calculate the Euclidean Distance between instances I squared the difference between the instances then summed them together and square rooted the result as seen in figure 2. The Euclidean distance gives the shortest/diagonal path between two points as seen in figure 1.



(Figure 1, http://www.ieee.ma/uaesb/pdf/distances-in-classification.pdf)

```
public static double CalculateEuclideanDistance(AdultTrain a, AdultTrain b)
{
    List<double> dist = new List<double>();
    //here I am adding the squared difference between the instance variables in a (the test instance)
    //and b (the training instance) to a list
    dist.Add(Square(a.age,b.age));
    dist.Add(Square(1, Match(a.workclass, b.workclass)));
    dist.Add(Square(a.fnlwgt, b.fnlwgt));
    dist.Add(Square(1, Match(a.education, b.education)));
    dist.Add(Square(a.educationNum, b.educationNum));
    dist.Add(Square(1, Match(a.maritalStatus, b.maritalStatus)));
    dist.Add(Square(1, Match(a.occupation, b.occupation)));
    dist.Add(Square(1, Match(a.relationship, b.relationship)));
    dist.Add(Square(1, Match(a.race, b.race)));
    dist.Add(Square(1, Match(a.sex, b.sex)));
    dist.Add(Square(a.capitalGain, b.capitalGain));
    dist.Add(Square(a.capitalLoss, b.capitalLoss));
    dist.Add(Square(a.hoursPerWeek, b.hoursPerWeek));
    dist.Add(Square(1, Match(a.nativeCountry, b.nativeCountry)));

    double sum = 0;
    //here I am summing all the squared distances together
    foreach(var dis in dist)
    {
        sum += dis;
    }
    //here I am calculating the square root of the sum and returning it which is the
    //euclidean distance
    return Math.Sqrt(sum);
}
```

(Figure 2, Code to calculate Euclidean Distance)

After this I sorted all the distances from each input to all instances in the validation set. I did this, so I could easily take the closest K distances and find the majority label from the closest instances. Using the sorted list, I looped through it stopping at all tested Ks (1,3,5,7 [..] 37,39). When stopping I would check the majority label from K instances compared with the input label and then incrementing the total number of errors for the current fold if the labels did not match. This method can be seen in the code in figure 3.

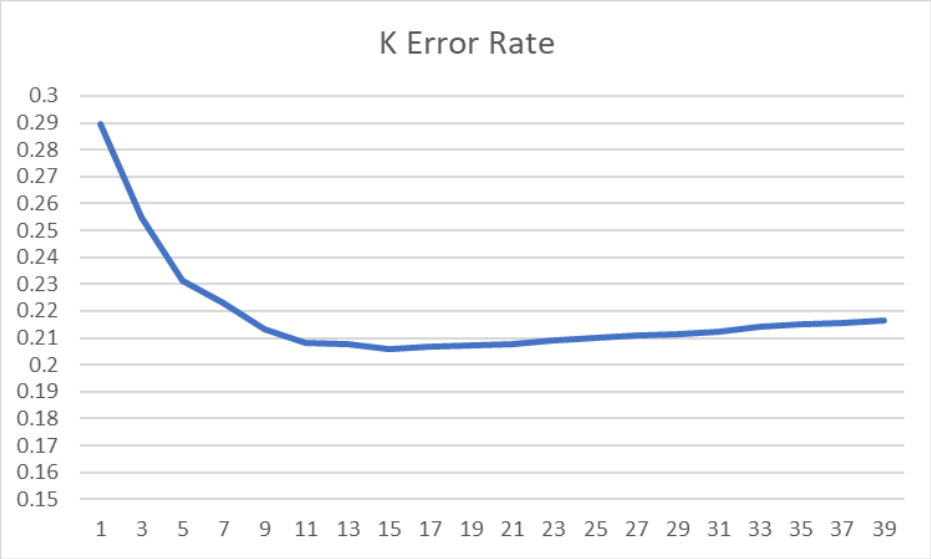
```
public static void CountError(AdultTrain input, TupleList<double, int> distLabels, int inputLabel, int split, int k)
{
    //in this class I loop through the sorted distances and add all distances labelled <=50K and >50K
    //to seperate lists, then when i equals a desired tested K, I classify the test instance as the class list
    //with more distances, I then increment the number of errors in the current fold for the K if the test instance was incorrectly classified
    int i = 1;
    List<double> lThan50 = new List<double>();
    List<double> gThan50 = new List<double>();

    foreach (var dl in distLabels)
    {
        if (dl.Item2 == 0)
        {
            lThan50.Add(dl.Item1);
        }
        else
        {
            gThan50.Add(dl.Item1);
        }
            if (i % 2 != 0 && i != 0)
            {
                int label = 1;
                if (lThan50.Count > gThan50.Count)
                {
                    label = 0;
                }
                if (inputLabel != label)
                {
                    kAccs[i][split - 1]++;
                }
            }
            if (i == 39)
                break;


         i++;
    }
}
```

(Figure 3)

Once I had collected the number of errors for each K in each fold, I calculated the error percentage for each fold. I did this by dividing the number of misclassified instances in a test fold by the total number of instances in the test fold. I then averaged all 5 folds percentages to give an overall error for each K. I outputted the results to a text file and created the graph labelled figure 4.



(Figure 4, showing the K errors using the results found in the "grid.results.txt" file)

In figure 4 you can see initially at the number of K increases the error decreases. This is because with low numbers of K there isn't a big enough sample size to give a confident classification. At 11 K the graph levels off then starts to increase. I suspect the increase is related there being 4 times as many <=50K instances as there is to >50. So as you increase K there is higher probability that <=50K will be on of the K nearest neighbours simply because they is more instances of them.

The best K found from cross validation was 15 K with an error rate of 0.205818181818182.

Testing

After the best K was found I began testing with the test set "adult.test.csv". I performed KNN on it in the same way but this time only taking results for 15 nearest neighbours. This mean I would take the closest 15 by distance from the training set to a test instance then classify the test instance with the majority class from the 15. While testing I recorded the number of errors for each class, so I could create the following confusion matrix (Figure 6) with the code in Figure 5.

```
public static double[,] ComputeConfusionMatrix()
{
    double [,] cm = new double[2, 2];
    //the total number of class 1 minus the number of class 1 incorrectly classified as class 2
    cm[0, 0] = c1Count - testErr[0];
    //the of class 1 incorrectly classified as class 2
    cm[0, 1] =  testErr[0];
    //the of class 2 incorrectly classified as class 1
    cm[1, 0] = testErr[1];
    //the total number of class 2 minus the number of class 1 incorrectly classified as class 1
    cm[0, 1] = c2Count - testErr[1];
    return cm;
}
```

(Figure 5)

	<=50 (Predicted)	>50 (Predicted)
<=50 (True)	12288 (c11)	147 (c12)
>50 (True)	3136 (c21)	710 (c22)

(Figure 6)

Key:

c11 – number of correctly predicted <=50 instances.

c12 – number of falsely predicted <=50 as >50 instances

c21 – number of falsely predicted >50 as <=50 instances

c22 –number of correctly predicted >50 instances

I then used these values to compute the Accuracy, Error, Sensitivity and Specificity. In the following code snippets "cm" is a 2D array representing the confusion matrix. All the code to compute these measures are found in the "TestFunctions.cs" class.

Accuracy: 0.798353909465021

This was computed by doing (c11 + c22) / (c11 + c12 + c21 + c22) which is the number of correctly classified instances divided by the total tested instances. In the following code:

```
public static double ComputeAccuracy(double [,] cm)
{
    double acc = (cm[0, 0] + cm[1, 1]) / (cm[0, 0] + cm[1, 1] + cm[0, 1] + cm[1, 0]);
    return acc;
}
```

The accuracy shows how well the model performed at classifying the test data.

Error: 0.201646090534979

The error is simply 1 – the accuracy. It gives a percentage of the number of incorrectly classified test instances.

Precision:

Precision shows the accuracy for a specific class. The precisions calculated that the KNN was better at classifying things as >50K as less mistakes were made when doing so by around 3% when compared with <=50K.

<=50K: 0.796680497925311 c11 / (c11 + c21)

```
public static double ComputePrecision0(double [,] cm)
{
    double prec = cm[0, 0] / (cm[0, 0] + cm[1, 0]);
    return prec;
}
```

>50K: 0.828471411901984 c22 / (c22 + c12)

```
public static double ComputePrecision1(double[,] cm)
{
    double prec = cm[1, 1] / (cm[1, 1] + cm[0, 1]);
    return prec;
}
```

Specificity:

Specificity is the ability of the KNN to correctly classify <=50K class. As you can see the KNN is excellent at classifying <=50K class with a nearly 99% specificity.

<=50K: 0.988178528347406 c11 / (c11 + c12)

```
public static double ComputeSensitivity0 (double [,] cm)
{
    double sens = cm[0, 0] / (cm[0, 0] + cm[0, 1]);
    return sens;
}
```

Sensitivity:

Sensitivity is the ability of the KNN to correctly classify >50K class. As you can see the KNN is very bad at classifying >50K class with a nearly 18% sensitivity.

>50K: 0.184607384295372 c21 / (c21 + c22)

```
public static double ComputeSpecificity(double [,] cm)
{
    double spec = cm[1, 1] / (cm[1, 1] + cm[1, 0]);
    return spec;
}
```