Classification

Classification is a problem that can be tackled with supervised learning. In supervised learning, we want to build a model using the training data with which we will then be able to make accurate predictions on new data that has the same characteristics as the training set that we used. The purpose of this project is to develop such algorithms from scratch. For that purpose a dataset with known labels was used (http://mlearn.ics.uci.edu/databases/yeast/yeast.data) where 1484 yeast proteins were classified into 10 classes with an accuracy of 55%.

The first algorithm that was developed is K-nearest neighbors (KNN). KNN is an instance-based learning algorithm which means that it doesn't generalize from the training dataset but instead memorizes it in order to predict the classes of new data. When new data is introduced, each data point will be matched to the label that corresponds to the most frequent label among the K nearest training data points. This basically means that the closest K data points to the new data point will define its class, by majority rule. Choosing a huge number of K reduces performance as it increases overfitting (imagine that choosing K equal to the number of the training data points will match every new data point to the most frequent class) while choosing a very low number causes underfitting. The right choice of k is crucial to find a good balance between over- and underfitting. A big downside of this algorithm is the computation time required which increases linearly as the training sample size increases.

The second algorithm that was developed is a Naive Bayes Classifier. Much more efficient in speed, the Naive Bayes generalizes by estimating the probability of belonging to a class given the values of the features. In order to estimate that probability, the probability density function of the Gaussian distribution is used. Within each class the data is summarized by the mean and standard deviation for each feature (in total, #features * #classes = #distributions). The probability density function will give us an estimate of how likely it is for a new data point to belong to the Gaussian distribution with these parameters (and therefore how likely it is to belong in that class). This is an estimate of the conditional probability to witness these values of the feature given that the data point belongs in that class. The product of these estimates is multiplied with the marginal probability of the class and finally a comparison of the final results will determine which class matches the data point.

To find which algorithm configuration performs best in our data 10-fold validation was used. The KNN was tested with k's from 1 to 25 and with 4 different distance metrics (euclidean, manhattan, chebyshev, cosine) and Naive Bayes performance was also tested. The highest accuracy output from 10-fold validation was 0.59953704 (basically 60%) which was managed by 10NN with manhattan as the distance metric. The leave one out method was also tried out, however since we have a big sample size it is not a good choice (for reference the accuracy of 10NN_manhattan was 0.5947565543071162 with LOO method).

Before doing any of the things mentioned above a 10% of the data was hidden and the rest 90% was used through all the algorithms and validations. This was done so that we can have a completely unbiased estimate of the accuracy given the configuration that the validation test has

proved to be the best. The accuracy with which the 10NN_manhattan algorithm managed to classify the test set was 0.5704697986577181 so basically 57% accuracy. To have a more accurate feel of the true accuracy, the whole process should be repeated as to try out different orientations of randomized data order which will result in different test sets each time (nested cross validation could achieve that purpose).

Below is the output of the 10Fold process. Two lists were used as it becomes very simple to match accuracy and configuration through the index.

```
>>>  ['10NN_1']
>>> configurations
array(['1NN_0', '1NN_1', '1NN_2', '1NN_3', '2NN_0', '2NN_1', '2NN_2',
       '2NN_3', '3NN_0', '3NN_1', '3NN_2', '3NN_3', '4NN_0', '4NN_1',
       '4NN_2', '4NN_3', '5NN_0', '5NN_1', '5NN_2', '5NN_3', '6NN_0',
       '6NN_1', '6NN_2', '6NN_3', '7NN_0', '7NN_1', '7NN_2', '7NN_3',
       '8NN_0', '8NN_1', '8NN_2', '8NN_3', '9NN_0', '9NN_1', '9NN_2',
       '9NN_3', '10NN_0', '10NN_1', '10NN_2', '10NN_3', '11NN_0', '11NN_1',
       '11NN_2', '11NN_3', '12NN_0', '12NN_1', '12NN_2', '12NN_3',
       '13NN_0', '13NN_1', '13NN_2', '13NN_3', '14NN_0', '14NN_1',
       '14NN_2', '14NN_3', '15NN_0', '15NN_1', '15NN_2', '15NN_3',
       '16NN_0', '16NN_1', '16NN_2', '16NN_3', '17NN_0', '17NN_1',
       '17NN_2', '17NN_3', '18NN_0', '18NN_1', '18NN_2', '18NN_3',
       '19NN_0', '19NN_1', '19NN_2', '19NN_3', '20NN_0', '20NN_1',
       '20NN_2', '20NN_3', '21NN_0', '21NN_1', '21NN_2', '21NN_3',
       '22NN_0', '22NN_1', '22NN_2', '22NN_3', '23NN_0', '23NN_1',
       '23NN_2', '23NN_3', '24NN_0', '24NN_1', '24NN_2', '24NN_3', 'NBayes'],
      dtype='<U6')
>>> accuracies
array([ 0.53611111,  0.53657407,  0.51740741,  0.53611111,  0.53611111,
        0.53657407,  0.51740741,  0.53611111,  0.54675926,  0.55175926,
        0.52740741,  0.54675926,  0.57574074,  0.56648148,  0.54814815,
        0.57574074,  0.57694444,  0.57398148,  0.55055556,  0.57694444,
        0.57861111,  0.57638889,  0.56740741,  0.57861111,  0.58898148,
        0.58685185,  0.56212963,  0.58898148,  0.58222222,  0.58037037,
        0.57185185,  0.58222222,  0.59527778,  0.58898148,  0.57842593,
        0.59527778,  0.58694444,  0.59953704,  0.56166667,  0.58694444,
        0.58916667,  0.58842593,  0.56296296,  0.58916667,  0.59666667,
        0.58462963,  0.55944444,  0.59666667,  0.58777778,  0.58444444,
        0.56768519,  0.58777778,  0.5937037 ,  0.5875    ,  0.56916667,
        0.5937037 ,  0.59305556,  0.58472222,  0.58055556,  0.59305556,
        0.58759259,  0.59509259,  0.56925926,  0.58759259,  0.58425926,
        0.58703704,  0.56842593,  0.58425926,  0.58574074,  0.59666667,
        0.57583333,  0.58574074,  0.58731481,  0.58935185,  0.57435185,
        0.58731481,  0.59101852,  0.59157407,  0.57962963,  0.59101852,
        0.5962963 ,  0.59222222,  0.5787037 ,  0.5962963 ,  0.59407407,
        0.59398148,  0.58111111,  0.59407407,  0.59731481,  0.5975    ,
        0.57944444,  0.59731481,  0.59101852,  0.59685185,  0.57111111,
        0.59101852,  0.51657407])
>>> accuracies[configurations=='10NN_1']
array([ 0.59953704])
```