The age prediction of Abalone using K-nearest neighbor

Machine Learning Assignment 2

Joel Satkauskas
R00116315

Abstract:

In this project I will attempt to predict the classification of abalones age by using K-nearest neighbor, distance weighted K-nearest neighbor using both the Euclidean and Manhattan distances and a one vs all approach.

Its age will be gotten from the ‘Rings’ feature by adding 1.5 onto it therefore ‘Rings’ will be the class to predict for this algorithm.

I am going to ignore the gender feature as it is not numeric and seems to be irrelevant to age.

The results conclude that the Manhattan distance is of higher accuracy then the Euclidean distance by 0.3-0.9% and that the distance weighted algorithm is more accurate than a normal Knn algorithm by 3%. The one vs all approached showed interesting results but in the end was extremely skewed as the number of records vastly outnumbered the records with a classification as there are 28 of them.

Data Set:

I have chosen a data set that details the numeric properties of abalone characteristics. The characteristics are:

Name / Data Type / Measurement Unit / Description

Sex / nominal / -- / M, F, and I (infant)
Length / continuous / mm / Longest shell measurement
Diameter / continuous / mm / perpendicular to length
Height / continuous / mm / with meat in shell
Whole weight / continuous / grams / whole abalone
Shucked weight / continuous / grams / weight of meat
Viscera weight / continuous / grams / gut weight (after bleeding)
Shell weight / continuous / grams / after being dried
Rings / integer / -- / +1.5 gives the age in years

There are no missing values for this data set and there are a total of 4,177 rows.

```
Records with classification
1.0  =  1
2.0  =  1
3.0  =  15
4.0  =  57
5.0  =  115
6.0  =  259
7.0  =  391
8.0  =  568
9.0  =  689
10.0  =  634
11.0  =  487
12.0  =  267
13.0  =  203
14.0  =  126
15.0  =  103
16.0  =  67
17.0  =  58
18.0  =  42
19.0  =  32
20.0  =  26
21.0  =  14
22.0  =  6
23.0  =  9
24.0  =  2
25.0  =  1
26.0  =  1
27.0  =  2
29.0  =  1
```

The amount of records of a certain classification.

Algorithm/Code:

Normal Knn:

Its implementation is split into the following methods.

- *doNormalKnnWithKFold(data, distanceMethod, drawCM)*

  This method will take in the data set, a number that indicates which distance method to use and a Boolean to draw the confusion matrix or not.

  This method is in charge of getting the accuracy of the Knn algorithm on the data set. It will keep track of the fold of the data set and get their average.

  Explanation:
  It will start by creating the folds for cross fold validation and then call `doKnnForFold()` for each fold. It will store the result of each fold and get the average.

- *doKnnForFold(folds, data, testFold, knn, distanceMethod, drawCM)*

  This method is in charge of evaluating the accuracy of Knn on the data set using the specified fold.

  `Folds` is an array of integers that indicate the finishing points for folds. Eg, `Folds = [835, 1670, 2505, 3340, 4177]`, 0-835 are the records for the first fold, 835-1670 are the records for the second fold, ect.

  `Data` is the abalone data set, `testFold` is which fold to use as the test data for the model, `knn` is the amount of neighbors to take into consideration, `distanceMethod` is to indicate which distance method to use and `drawCM` is to indicate whether to draw a confusion matrix for the fold or not.

  Explanation:
  It begins by copying the data set for the training data and finding a range of records based on the `folds` array and the `testFold` value. Once it has this range of integers, it will use them to add the respective records to the testing data set and removing them from the training data set.

Once this is done it will use `getNeighbors()` to get the nearest records, `getResponse()` to get the most frequent class of those records and `getAccuracy()` to get the accuracy of the records it predicted. Then based on `drawCM` it will draw a confusion matrix for that fold.

- *`getNeighbors(trainingSet, testInstance, k, ditanceMethod)`*

The method will get the nearest neighbors of the test record provided.

`TrainingSet` is a list of the records to use as the training model. `testInstance` is the record to get the nearest neighbors of, `k` is the amount of neighbors to take into consideration and the `distanceMethod` is an indication of which method to use.

This method will get the K nearest neighbors using either the Euclidian or Manhattan distance and return the closest k neighbors.

Explanation:
Depending on `ditanceMethod` it will either use the `euclideanDistance()` or `manhatanDistance()` method. It will compare the test instance/record against each record in the training set, sort by value and return the first k records.

- *`getResponse(neighbors)`*

A method that will take in a list of neighbors and return the neighbor that occurs most frequently

- *`getAccuracy(testData, predictions)`*

A method that will take in the `testData` and its `prediction` and will evaluate the accuracy of these predictions.

- *`euclideanDistance(row1, row2, length)`*

This method will get the Euclidean distance between 2 records.

`Row1` is the test record to compare to, row2 passes in the next record of the training set and `length` is how many columns of the records to compare to, it is the arrays length – 1

because we do not want to compare the last column since this is the value we are trying to predict.

This method will get the square root of the sum of the difference between each columns value, squared and return that squared distance.

- *manhatanDistance(row1, row2, length)*

This method will get the Manhattan distance between 2 records.

`Row1` is the test record to compare to, row2 passes in the next record of the training set and `length` is how many columns of the records to compare to, it is the arrays length – 1 because we do not want to compare the last column since this is the value we are trying to predict.

This method will get the sum of the absolute difference between each column of the 2 rows and return it.

Distance weighted Knn:

Its implementation is quite similar to the normal Knn implementation, it is split into the following method:

- *doWeightedKnnWithKFold(data, distanceMethod, drawCm)*

This method is in charge of evaluating the weighted distance knn algorithm.

`Data` is the abalone data set, `distanceMethod` is the indicator of which distance method to use and `drawCm` is a Boolean to whether draw a confusion matrix or not.

Explanation:

This method follows the same logic as `doNormalKnnWithKFold()` except that instead of calling `doKnnForFold()` it calls `doKnnDWForFold()`.

- *doKnnDWForFold(folds, data, testFold, distanceMethod, drawCm)*

This method will run the weighted distance algorithm on a fold of the dataset.

`Folds` is an array of integers that indicate the finishing points for folds. Eg, `Folds = [835, 1670, 2505, 3340, 4177]`, 0-835 is the records for the first fold, 835-1670 is the records for the second fold, ect.

`Data` is the abalone data set, `testFold` is which fold to use as the test data for the model, `distanceMethod` is to indicate which distance method to use and `drawCM` is to indicate whether to draw a confusion matrix for the fold or not.

Explanation:

Again this method follows the same logic as `doKnnForFold()` except instead of calling `getNeighbors()` it calls `getWeightedNeighbour()`.

- *getWeightedNeighbour(trainingSet, testInstance, distanceMethod)*

This method will get the class that has the highest score of distance weight for the record instance provided.

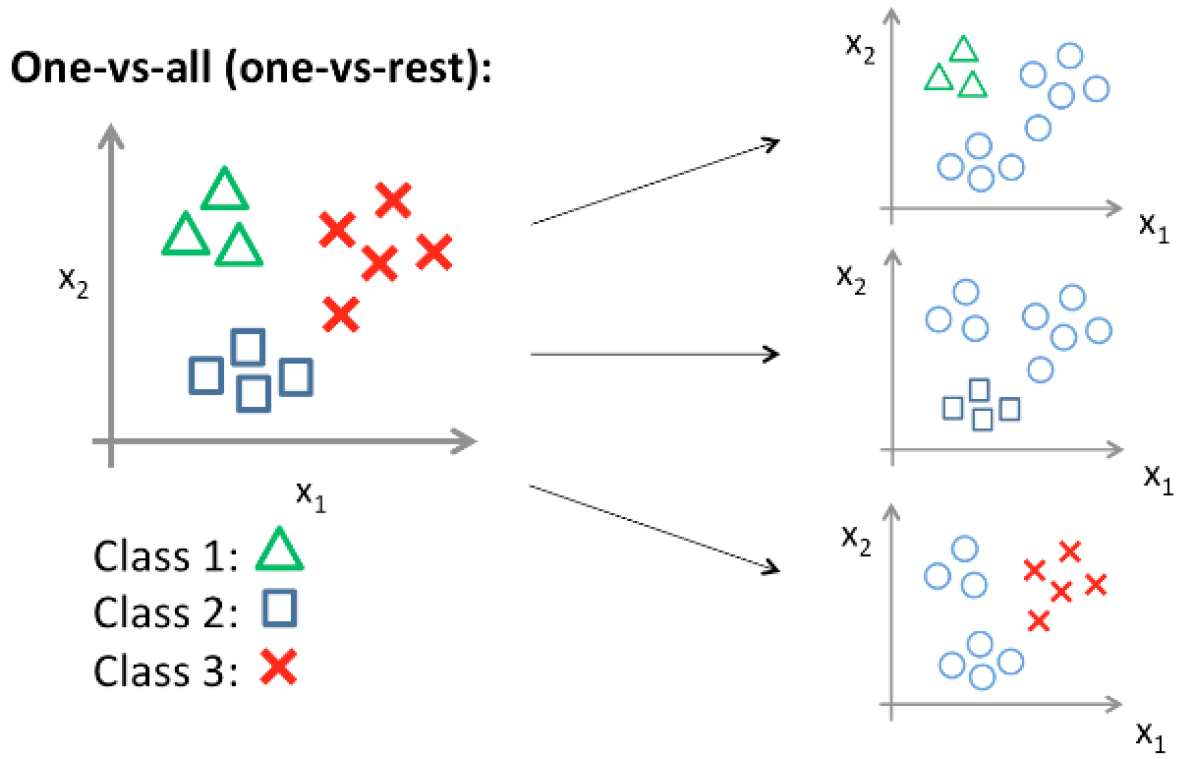
`trainingSet` is a list of records that are used for training, `testInstance` is the next test record to find the class of, `distanceMethod` is which distance method to use.

Explanation:
Depending on `ditanceMethod` it will either use the `euclideanDistance()` or `manhatanDistance()` method. It will compare the distance of the test instance/record against each record in the training set then add up the distance of each classification.

It will sort this list by highest value and return the first instance therefore returning the classification with the highest score in distance-weight.

One vs All Knn:

This approach is implemented by copying the abalone data set for each classification and turning each data set into a binary classification were one class is the class we want to predict and the other class is rest.



The methods that are involved in its algorithm are:

- *doOneVsAllWithKFold(classes, data)*

This method is in charge of running the one vs all algorithm. It will run k-fold validation for each classification to predict.

`Classes` is a list of the abalone data sets, `data` is the original data set.

Explanation:

The method will start off by finding the amount of records per class in the entire data set and using `DataOneVsMany()` to change each data set into a binary classification. Then, for each data classification, it will gets the folds and call `doKnnForFold()` for each fold, getting the average of the folds and at the end getting the average of the binary classifications.

- *DataOneVsMany(dataset, classifcation)*

A worker method that will take in a data set and a number, and change the all records who's classification is not that number to 0. Essentially splitting the classifications into 2 classes.

The normal Knn algorithm is then used to get the accuracy.

Results/Evaluation:

Normal Knn:

For a simple Knn algorithm I tried using both the Euclidean and Manhattan distances. For k I used the square root of 29 which rounds down to 5. To save computation time when implementing the algorithm I used a 5 fold for cross fold validation with 10 only giving slightly lower accuracies, at most 0.5% lower.

The result was that the Manhattan distance was more accurate than the Euclidean distance by 0.93%.

Manhattan:

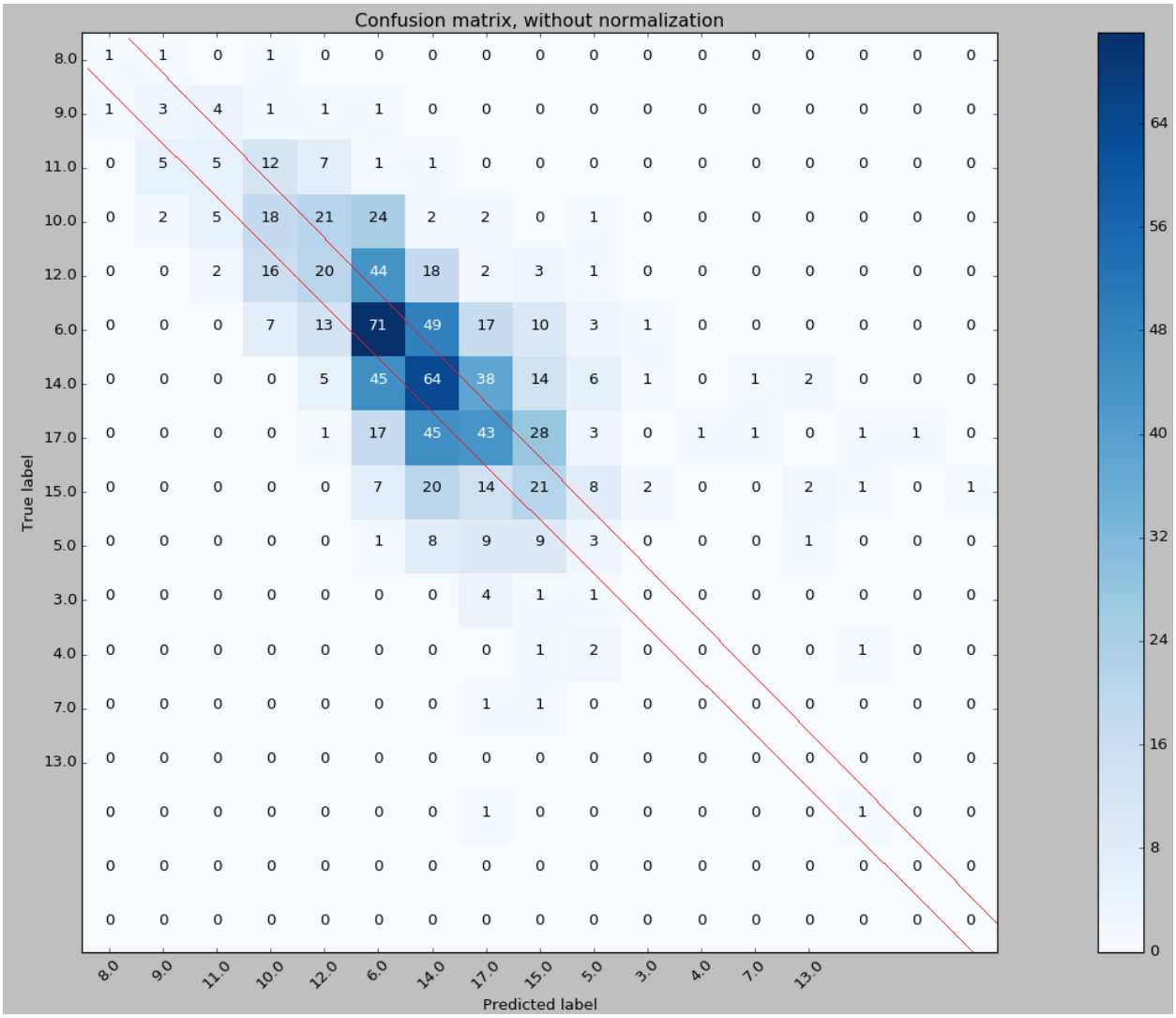
```
------------------------------------------------------------
Normal Knn
Manhatan Distance
Average accuracy = 23.36%
Knn =  5
k-Fold =  5
------------------------------------------------------------
```

Euclidean:

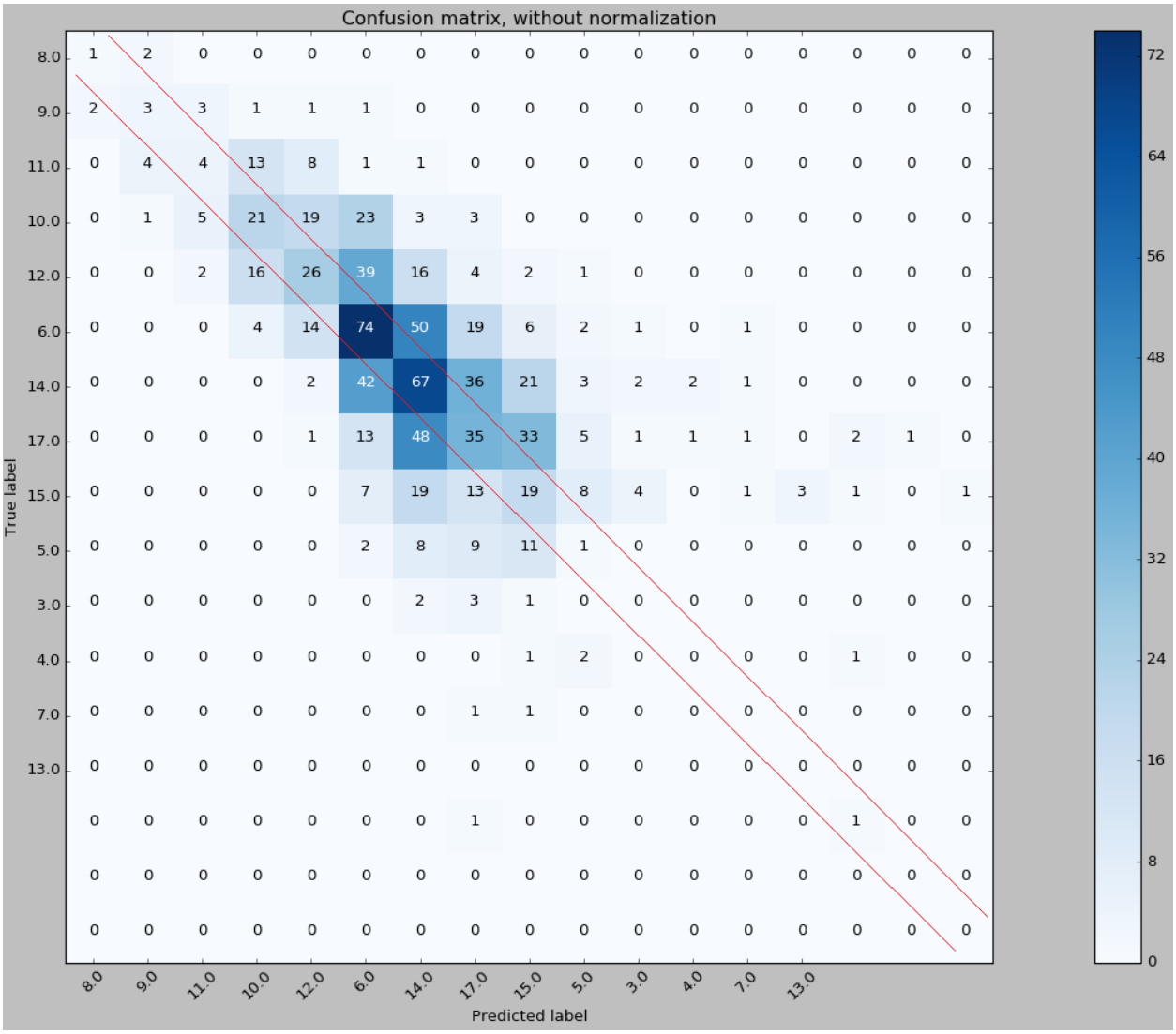
```
------------------------------------------------------------
Normal Knn
Euclidean Distance
Average accuracy = 22.43%
Knn =  5
k-Fold =  5
------------------------------------------------------------
```

Taking a look at the confusion matrix of the third fold of both distances we can see that in both cases, the highest classification prediction accuracy is with the class 6, 12, 14 and 17 with many classes surrounding them indicating that they were almost predicted correctly and were only off by 1-3 classes.

Manhattan:


Confusion matrix, without normalization
True label
Predicted label
8.0 9.0 11.0 10.0 12.0 6.0 14.0 17.0 15.0 5.0 3.0 4.0 7.0 13.0
64 56 48 40 32 24 16 8 0

Euclidean:


Confusion matrix, without normalization
True label
Predicted label
8.0 9.0 11.0 10.0 12.0 6.0 14.0 17.0 15.0 5.0 3.0 4.0 7.0 13.0
72 64 56 48 40 32 24 16 8 0

Distance weighted Knn:

For the distance weighted knn algorithm, I again tried both the Manhattan and Euclidean distances. I left k and the amount of folds the same as the normal knn algorithm.

The result was an overall improvement over the normal knn algorithm by 2.5-3.1%. Again the Manhattan distance methodology was more accurate than the Euclidean distance by 0.35%

Manhattan:

```
------------------------------------------------------------
Weighted-Distance Knn
Manhatan Distance
Average accuracy = 25.88%
Knn =  5
k-Fold =  5
------------------------------------------------------------
```

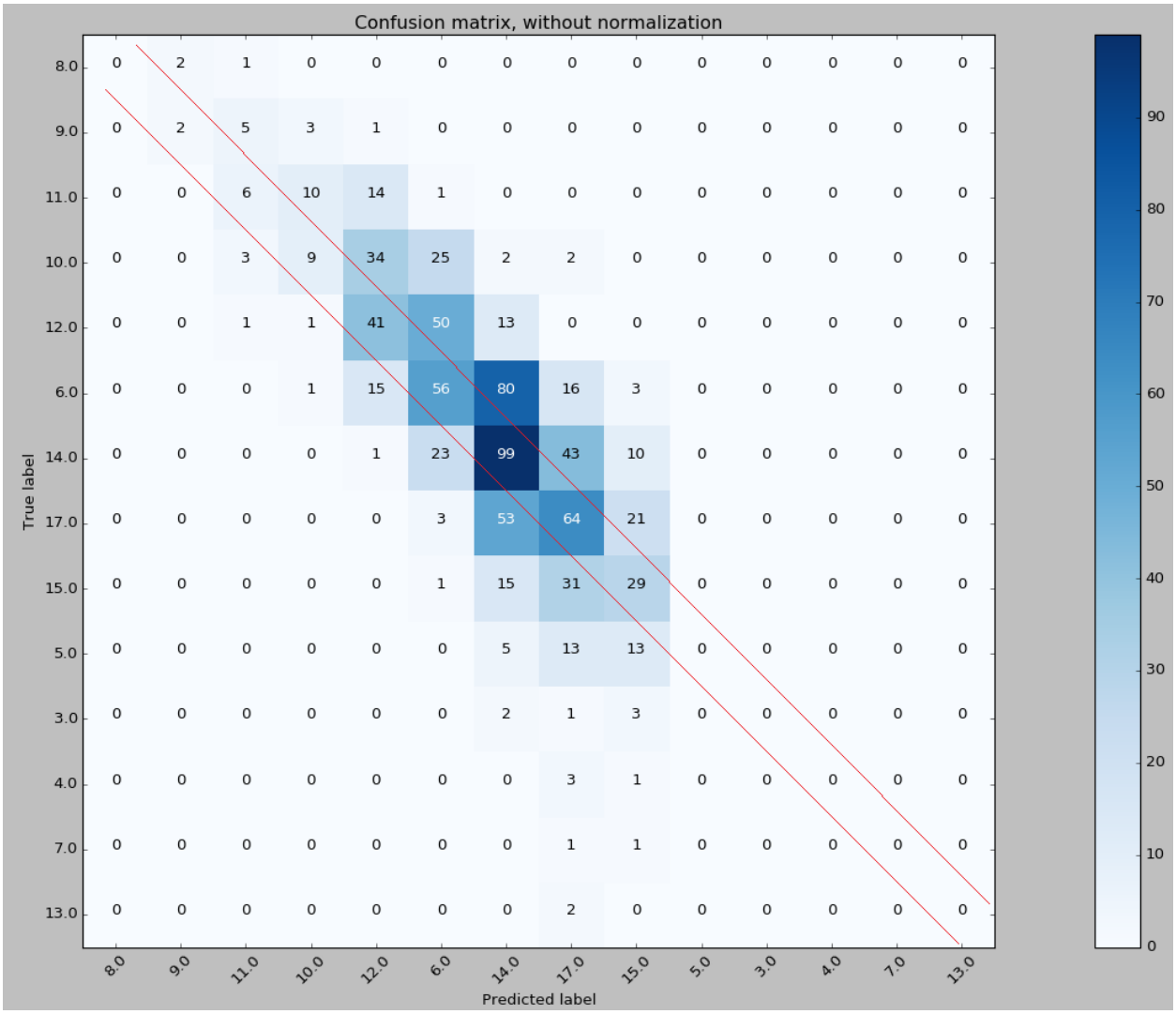
Euclidean:

```
------------------------------------------------------------
Weighted-Distance Knn
Euclidean Distance
Average accuracy = 25.52%
Knn =  5
k-Fold =  5
------------------------------------------------------------
```

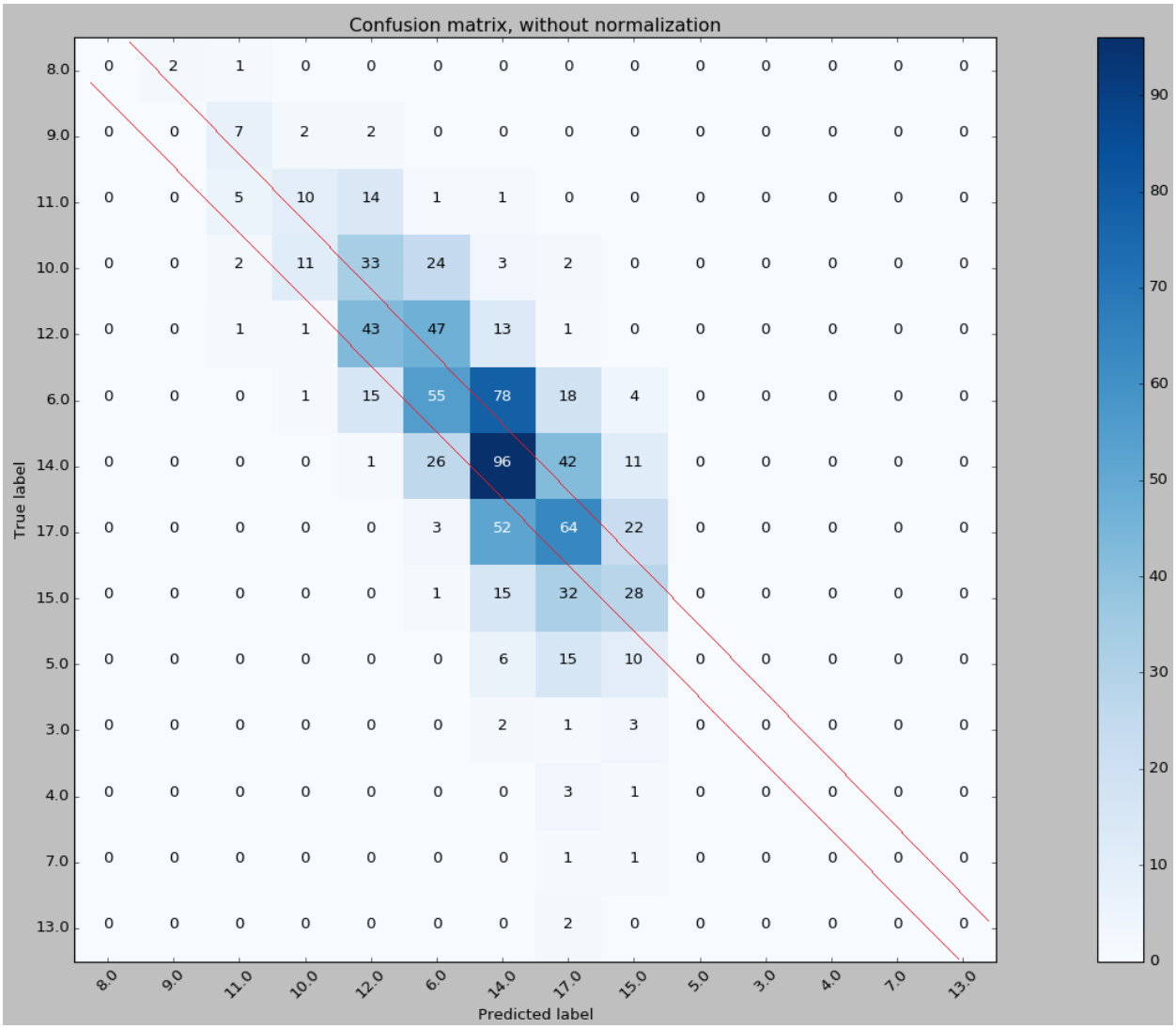
As we can see from the confusion matrixes, distance weighted followed a similar outcome as the normal knn algorithm, both having a strong accuracy for classifying the 6, 12, 14 and 17 classifications.

An interesting thing to note, the distance weighted confusion matrixes spread around the line of credit is more concentrated and closer towards the line then the normal knn algorithm. This might indicate that even if a record was classified incorrectly, its real classification is highly likely to be very close.

Manhattan:


Confusion matrix, without normalization
True label
Predicted label
8.0 9.0 11.0 10.0 12.0 6.0 14.0 17.0 15.0 5.0 3.0 4.0 7.0 13.0
0 2 1 0 0 0 0 0 0 0 0 0 0 0
0 2 5 3 1 0 0 0 0 0 0 0 0 0
0 0 6 10 14 1 0 0 0 0 0 0 0 0
0 0 3 9 34 25 2 2 0 0 0 0 0 0
0 0 1 1 41 50 13 0 0 0 0 0 0 0
0 0 0 1 15 56 80 16 3 0 0 0 0 0
0 0 0 0 1 23 99 43 10 0 0 0 0 0
0 0 0 0 0 3 53 64 21 0 0 0 0 0
0 0 0 0 0 1 15 31 29 0 0 0 0 0
0 0 0 0 0 0 5 13 13 0 0 0 0 0
0 0 0 0 0 0 2 1 3 0 0 0 0 0
0 0 0 0 0 0 0 3 1 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 2 0 0 0 0 0 0
90 80 70 60 50 40 30 20 10 0

Euclidean:


Confusion matrix, without normalization
True label
Predicted label
8.0
9.0
11.0
10.0
12.0
6.0
14.0
17.0
15.0
5.0
3.0
4.0
7.0
13.0
0 2 1 0 0 0 0 0 0 0 0 0 0 0
0 0 7 2 2 0 0 0 0 0 0 0 0 0
0 0 5 10 14 1 1 0 0 0 0 0 0 0
0 0 2 11 33 24 3 2 0 0 0 0 0 0
0 0 1 1 43 47 13 1 0 0 0 0 0 0
0 0 0 1 15 55 78 18 4 0 0 0 0 0
0 0 0 0 1 26 96 42 11 0 0 0 0 0
0 0 0 0 0 3 52 64 22 0 0 0 0 0
0 0 0 0 0 1 15 32 28 0 0 0 0 0
0 0 0 0 0 0 6 15 10 0 0 0 0 0
0 0 0 0 0 0 2 1 3 0 0 0 0 0
0 0 0 0 0 0 0 3 1 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 2 0 0 0 0 0 0
90
80
70
60
50
40
30
20
10
0

One vs All Knn:

For this approach I made a copy of the data set for each classification and changed each one into a binary classification for each one of the original classes. I then ran each of these binary classifications through the normal knn algorithm and got each ones accuracy. The accuracy of the one vs all approach is then the average accuracy of every classes data set that was cast into a binary classification.

```
--------------------------------------------------
Finished all classes for One vs All
Accuracy: 94.00%

Classifications =  3 - 21
> Euclidean Distance
Knn =  5
k-Fold =  5
--------------------------------------------------
```

At first this approach seems to greatly increase the accuracy by up to 94%. But taking a closer look it is clear that much of the data is skewed.

```
Records with classification
1.0  =  1
2.0  =  1
3.0  =  15
4.0  =  57
5.0  =  115
6.0  =  259
7.0  =  391
8.0  =  568
9.0  =  689
10.0  =  634
11.0  =  487
12.0  =  267
13.0  =  203
14.0  =  126
15.0  =  103
16.0  =  67
17.0  =  58
18.0  =  42
19.0  =  32
20.0  =  26
21.0  =  14
22.0  =  6
23.0  =  9
24.0  =  2
25.0  =  1
26.0  =  1
27.0  =  2
29.0  =  1
```

For many classifications the total records outnumber the records with the class by so much that the data is skewed in favor of the record not being of the class and so therefore appears more accurate than it actually is.

For example, the classification of 3:

```
Classification:  3
Average accuracy =  99.5451391125
Records with this classification =  15
```

Has a 99.5% accuracy but has only 15 records inside the data set. This is only 0.36% of the entire data set therefore the other 99.64% consists of the other class and would be picked over the 15 classes as there are far more of them.

Even with the class that has the highest amount of records of its classification, classification 9, it only accounts for 16.5% of the entire data set and so would also be quite skewed in its accuracy.

The problem is that the amount of classifications are splitting the records too thin, a solution might be to combine the classifications into groups therefore decreasing the amount of classifications and increase the amount of records for the groups and then further classifying them but only using the dataset from the previous classification. E.g.

Classifications:
1-5
5-10
10-15
15-20
20-25
25-30

If a record is identified as belonging to the class 5-10, then classify again for each one but only use records that have a classification of 5-10.