University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

CS6405 Data Mining Project Report

Assignment 1

Build a k-Nearest Neighbour algorithm that takes as input a training and test dataset and will predict the target variable.

James Henry Hehir

March 3rd 2021

Table of Contents

INTRODUCTION

The objective of this project is to build a k-Nearest Neighbour algorithm that takes as input a training and test dataset and will predict the target variable (excellent vs. poor quality) with a reasonable degree of accuracy. Typically you should be able to obtain an accuracy in excess of approx. 65% on this dataset.

In this project you must implement your own machine learning library and therefore **you are only allowed to use the following python packages and libraries: numpy, pandas, matplotlib.pyplot and seaborn.**

This datasets is related to red variants of the Portuguese "Vinho Verde" wine. For more details, consult the reference [Cortez et al., 2009]. In this project, we view this as a binary classification problem, i.e., excellent quality (+1) vs. poor quality (-1) wine. The original dataset is available from the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/wine+quality) and in Kaggle. However, please notice that this dataset has been modified to fit the purpose. Our modified dataset is available in Canvas.

INITIAL IMPLEMENTATION

Euclidean Distance

In my initial implementation of the KNN method, I calculated the Euclidean distance. This function is called calculateDistance() and it takes two points and minus the squared points from each other and gets the sum of this value and finally squares it again and returns the distance value.

```
def calculateDistance(dataSet, query_point):
    distance = np.square(dataSet - query_point) # (ai-bi)**2 for every point in the
vectors
    distance = np.sum(distance) # adds all values
    distance = np.sqrt(distance)
    return distance
```

After this I used a function that takes the distance found in the function mentioned above loops through the training feature vectors to find the distance between all training points. It returns these values in a numpy array. This array will be used in the knn function later to find the predictions.

```
def distance_from_all_training(test_point):
    dist_array = np.array([])
    for train_point in train_features:
        dist = calculateDistance(test_point, train_point)
        dist_array = np.append(dist_array, dist)
    return dist_array
```

In my code after this I read in the .csv files with training and test data. I did some data cleaning by declaring the features of both the training and test data and converting these into a numpy array. Similarly I did the same for the target values on both datasets.

```python
wine_test_df = pd.read_csv('wine-data-project-test.csv', sep = ',')
wine_train_df = pd.read_csv('wine-data-project-train.csv', sep = ',')

features = wine_train_df[['fixed acidity', 'volatile acidity', 'citric acid',
'residual sugar','chlorides', 'free sulfur dioxide', 'total sulfur dioxide',
'density', 'pH', 'sulphates', 'alcohol']]
train_features = features.to_numpy() # converts feature set to numpy array
features_test = wine_test_df[['fixed acidity', 'volatile acidity', 'citric acid',
'residual sugar','chlorides', 'free sulfur dioxide', 'total sulfur dioxide',
'density', 'pH', 'sulphates', 'alcohol']]
test_features = features_test.to_numpy()

train_target = wine_train_df['Quality'].to_numpy() # converts target column to
numpy array
test_target = wine_test_df['Quality'].to_numpy()
```

In my knn implementation it takes four arguments, these include train_features, train_target, test_features and a k value. I have a numpy array to store my predictions. I used a for loop to iterate through every test data point in the test features array. Then we calculate the distance from every training data point. This is store is dist_array. Then we calculate the neighbours and their distance to sort the training points on the basis of this distance. All of this is appended to predictions to return the predictions of the knn function.

```python
def knn(train_features, train_target, test_features, k):


    predictions = np.array([])
    train_target = train_target.reshape(-1,1)
    for test_point in test_features: # iterating through every test data point
        dist_array = distance_from_all_training(test_point).reshape(-1,1) #
calculating distance from every training data instance
        neighbors = np.concatenate((dist_array, train_target), axis = 1)
        neighbors_sorted = neighbors[neighbors[:, 0].argsort()] # sorts training
points on the basis of distance
        k_neighbors = neighbors_sorted[:k] # selects k-nearest neighbors
        frequency = np.unique(k_neighbors[:, 1], return_counts=True)
        target_class = frequency[0][frequency[1].argmax()] # selects label with
highest frequency
        predictions = np.append(predictions, target_class)

    return predictions
```

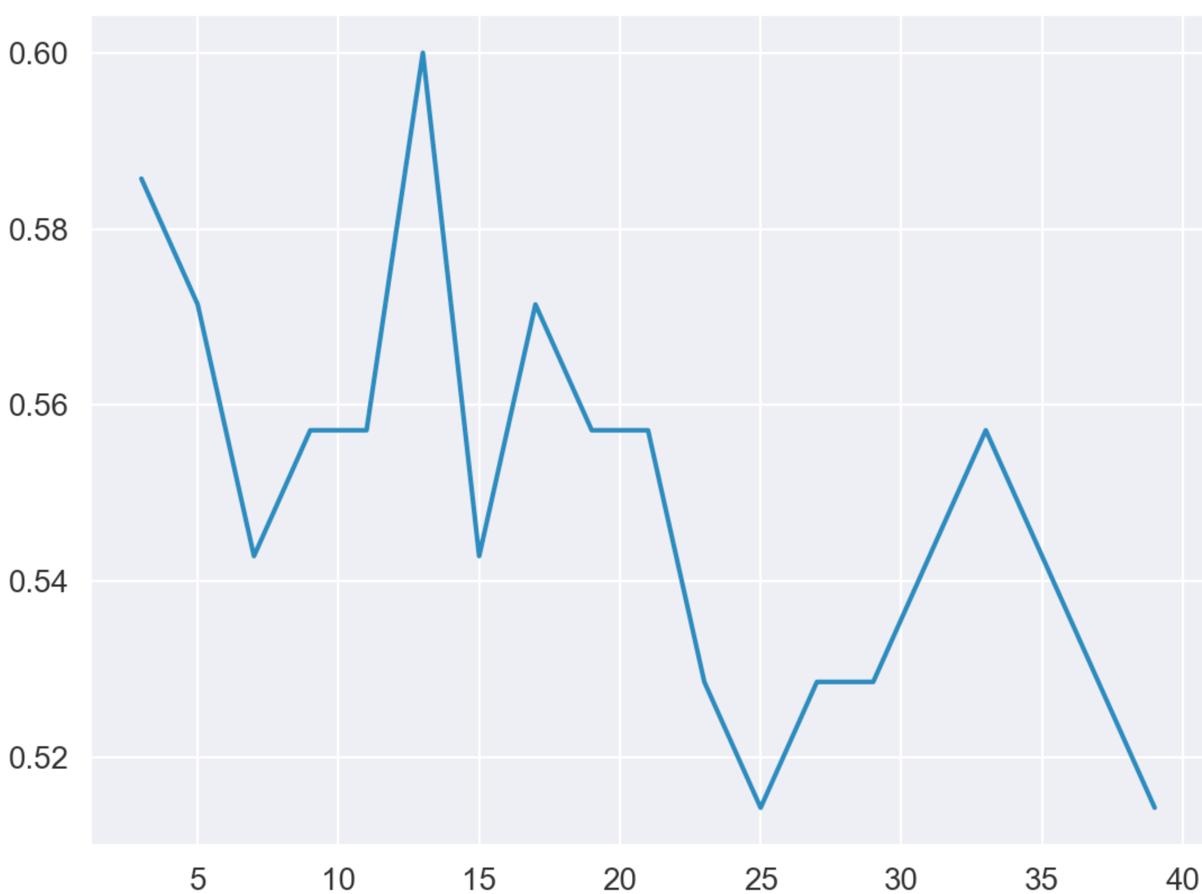
After this we used an accuracy function to test our results. This function compares our results with the test target values to see how many we got correct. We used this in our for loop that iterates through the odd values between 2 and 40. Finally it prints a plot to show our results.

```
def accuracy(y_test, y_preds):
    correctClassifications = 0
    for i in range(len(y_test)):
        if int(y_test[i]) == int(y_preds[i]):
            correctClassifications += 1
    acc = correctClassifications/len(y_test)
    return acc

allResults = []
for k in range(3, 40, 2):
    test_predictions = knn(train_features, train_target, test_features, k)
    allResults.append(accuracy(test_target, test_predictions))
    acc = accuracy(test_target, test_predictions)
    print('Model accuracy ' , k , '=', acc*100)

sns.set_style("darkgrid")
plt.plot( list(range(3, 40, 2)), allResults)
plt.show()
```

```
Jimmys-MacBook-Pro:Downloads jimmyhehir$ python3 TemplateKNN.py
Model accuracy  3 = 58.57142857142858
Model accuracy  5 = 57.14285714285714
Model accuracy  7 = 54.285714285714285
Model accuracy  9 = 55.714285714285715
Model accuracy  11 = 55.714285714285715
Model accuracy  13 = 60.0
Model accuracy  15 = 54.285714285714285
Model accuracy  17 = 57.14285714285714
Model accuracy  19 = 55.714285714285715
Model accuracy  21 = 55.714285714285715
Model accuracy  23 = 52.85714285714286
Model accuracy  25 = 51.42857142857142
Model accuracy  27 = 52.85714285714286
Model accuracy  29 = 52.85714285714286
Model accuracy  31 = 54.285714285714285
Model accuracy  33 = 55.714285714285715
Model accuracy  35 = 54.285714285714285
Model accuracy  37 = 52.85714285714286
Model accuracy  39 = 51.42857142857142
```



The K value 13 clearly gives us the best result. The 60% was the highest result I could achieve with this KNN method using the Euclidean distance. In the next section I will address my Manhattan implementation.

Manhattan Distance

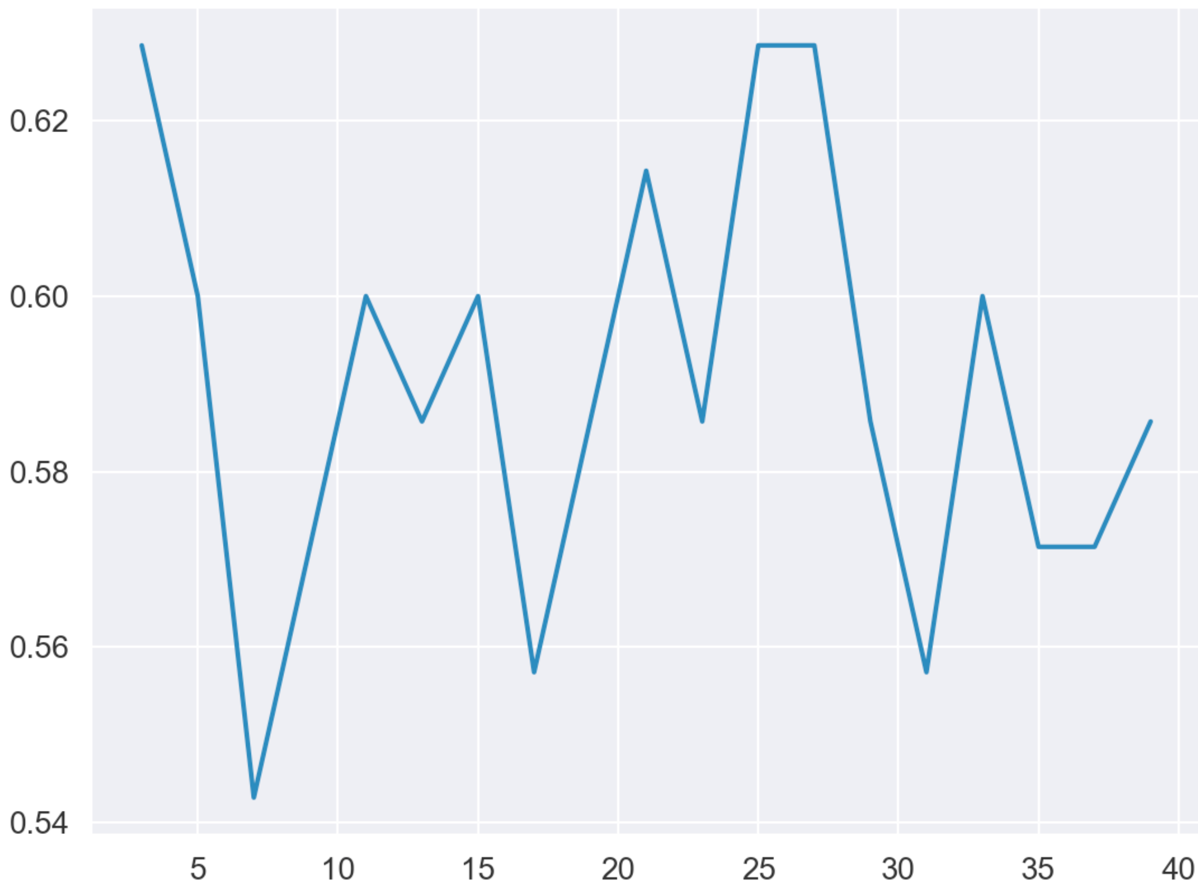
Manhattan Distance has the same approach as the initial KNN approach except for a different calculateDistance() function which I named ManhattanDistance().

```
def manhattanDistance(dataSet, query_point):
    return sum(abs(e1-e2) for e1, e2 in zip(dataSet,query_point))


def distance_from_all_training(test_point):
    dist_array = np.array([])
    for train_point in train_features:
        dist = manhattanDistance(test_point, train_point)
        dist_array = np.append(dist_array, dist)
    return dist_array
```

After implementing this distance I got far better results. The k value and 25 and 27 gave the best accuracy.

```
Jimmys-MacBook-Pro:Downloads jimmyhehir$ python3 TemplateKNN.pyModel accuracy  3 =
62.857142857142854
Model accuracy  5 = 60.0
Model accuracy  7 = 54.285714285714285
Model accuracy  9 = 57.14285714285714
Model accuracy  11 = 60.0
Model accuracy  13 = 58.57142857142858
Model accuracy  15 = 60.0
Model accuracy  17 = 55.714285714285715
Model accuracy  19 = 58.57142857142858
Model accuracy  21 = 61.42857142857143
Model accuracy  23 = 58.57142857142858
Model accuracy  25 = 62.857142857142854
Model accuracy  27 = 62.857142857142854
Model accuracy  29 = 58.57142857142858
Model accuracy  31 = 55.714285714285715
Model accuracy  33 = 60.0
Model accuracy  35 = 57.14285714285714
Model accuracy  37 = 57.14285714285714
Model accuracy  39 = 58.57142857142858
```

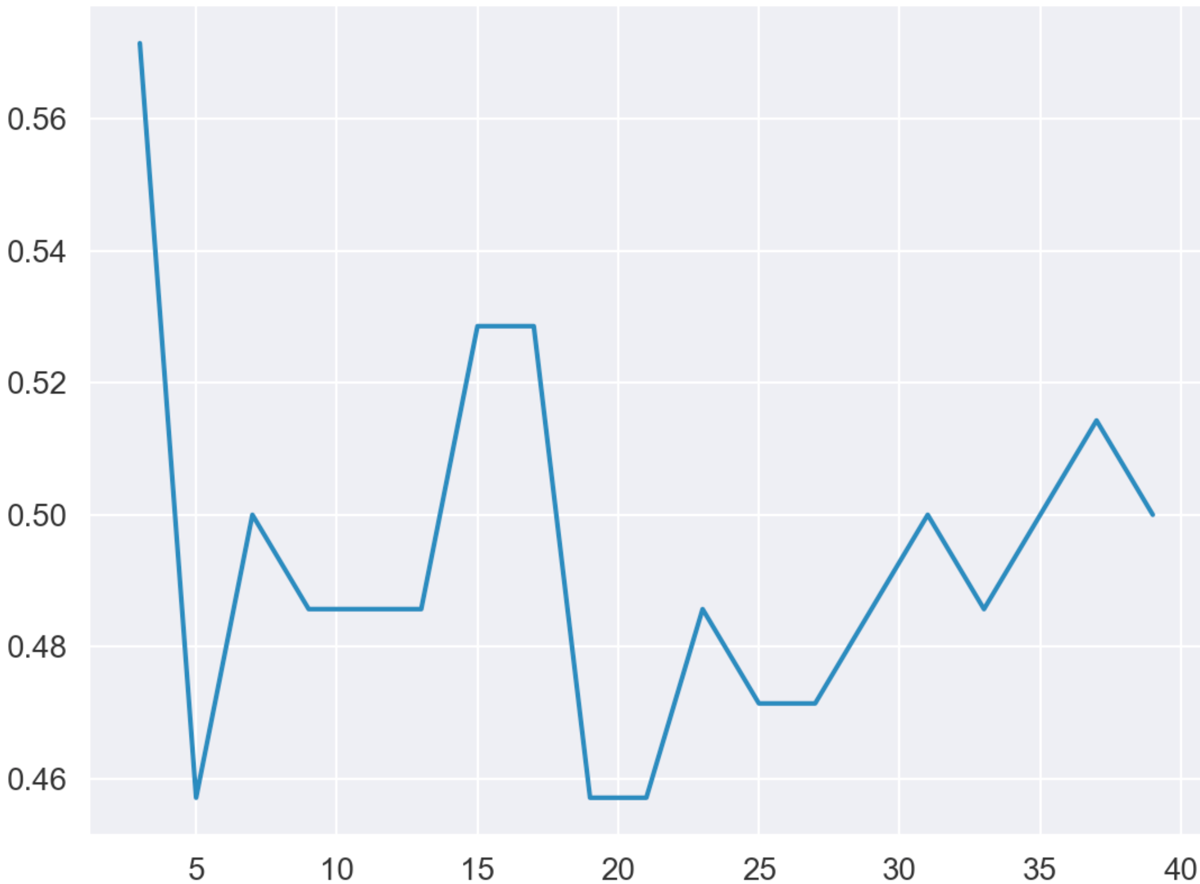

0.62
0.60
0.58
0.56
0.54
5
10
15
20
25
30
35
40

Weighted KNN implementation (Euclidean Distance)

Using the Euclidean distance I used a weighted KNN implementation by inversing and square rooting the distance. The approach was similar except the dist_array got inversed and square rooted in the Weightedknn function.

```
def Weightedknn(train_features, train_target, test_features, k):
    predictions = np.array([])
    train_target = train_target.reshape(-1,1)
    for test_point in test_features: # iterating through every test data point
        dist_array = distance_from_all_training(test_point).reshape(-1,1) #
calculating distance
        dist_array = np.square(dist_array[::-1]) #inverse square distance
        neighbors = np.concatenate((dist_array, train_target), axis = 1)
        neighbors_sorted = neighbors[neighbors[:, 0].argsort()] # sorts training
points on the basis of distance
        k_neighbors = neighbors_sorted[:k] # knn selected
        frequency = np.unique(k_neighbors[:, 1], return_counts=True)
        target_class = frequency[0][frequency[1].argmax()] # highest frequency
        predictions = np.append(predictions, target_class)
    return predictions
```

```
Jimmys-MacBook-Pro:Downloads jimmyhehir$ python3 TemplateKNN.py
Model accuracy  3 = 57.14285714285714
Model accuracy  5 = 45.714285714285715
Model accuracy  7 = 50.0
Model accuracy  9 = 48.57142857142857
Model accuracy  11 = 48.57142857142857
Model accuracy  13 = 48.57142857142857
Model accuracy  15 = 52.85714285714286
Model accuracy  17 = 52.85714285714286
Model accuracy  19 = 45.714285714285715
Model accuracy  21 = 45.714285714285715
Model accuracy  23 = 48.57142857142857
Model accuracy  25 = 47.14285714285714
Model accuracy  27 = 47.14285714285714
Model accuracy  29 = 48.57142857142857
Model accuracy  31 = 50.0
Model accuracy  33 = 48.57142857142857
Model accuracy  35 = 50.0
Model accuracy  37 = 51.42857142857142
Model accuracy  39 = 50.0
```



The result shows when the k value is 3 we get our best accuracy.

NORMALIZATION

In my implementation of normalizing my data I used the min max method on the data. My implementation is below.

```
normalized_test_features = (test_features - np.min(test_features)) /
(np.max(test_features) - np.min(test_features))
normalized_test_target = (test_target - np.min(test_target)) / (np.max(test_target)
- np.min(test_target))
normalized_train_features = (train_features - np.min(train_features)) /
(np.max(train_features) - np.min(train_features))
normalized_train_target = (train_target - np.min(train_target)) /
(np.max(train_target) - np.min(train_target))
```

The only issue was when I ran the KNN method with the new normalized data it kept returning an accuracy of 50 for every k value.

```
Jimmys-MacBook-Pro:Downloads jimmyhehir$ python3 TemplateKNN.py
Model accuracy  3 = 50.0
Model accuracy  5 = 50.0
Model accuracy  7 = 50.0
Model accuracy  9 = 50.0
Model accuracy  11 = 50.0
Model accuracy  13 = 50.0
Model accuracy  15 = 50.0
Model accuracy  17 = 50.0
Model accuracy  19 = 50.0
Model accuracy  21 = 50.0
Model accuracy  23 = 50.0
Model accuracy  25 = 50.0
Model accuracy  27 = 50.0
Model accuracy  29 = 50.0
Model accuracy  31 = 50.0
Model accuracy  33 = 50.0
Model accuracy  35 = 50.0
Model accuracy  37 = 50.0
Model accuracy  39 = 50.0
```

On further inspection the test_features were being read in as scientific notation values as compared to the train_features which were normal decimal values. I could not find a solution to this issue.

Similarly with the Z-score implementation I did not find a solution to this. I did try to implement a scalar method but it wouldn’t work with the data. Below is my implementation.

```
def standardScaler(feature_array):

    total_cols = feature_array.shape[0] # total number of columns
    for i in range(total_cols): # iterating through each column
        feature_col = feature_array[:, i]
        mean = feature_col.mean() # mean stores mean value for the column
        std = feature_col.std() # std stores standard deviation value for the
column
        feature_array[:, i] = (feature_array[:, i] - mean) / std # standard scaling
of each element of the column
    return feature_array

train_features_scaled = standardScaler(train_features)
test_features_scaled = standardScaler(test_features)
train_target_scaled = standardScaler(train_target)
test_target_scaled = standardScaler(test_target)
```

(Sharma, 2020)

WORKS CITED

Sharma, A. (2020, September 13). *ML from Scratch: K-Nearest Neighbors Classifier*. Retrieved from Towards Data Science: https://towardsdatascience.com/ml-from-scratch-k-nearest-neighbors-classifier-3fc51438346b