Homework 1: K-nearest neighbors for time-series classification

Nam Ho Phan

April 20, 2021

Class: ALY6020 - Predictive Analytics

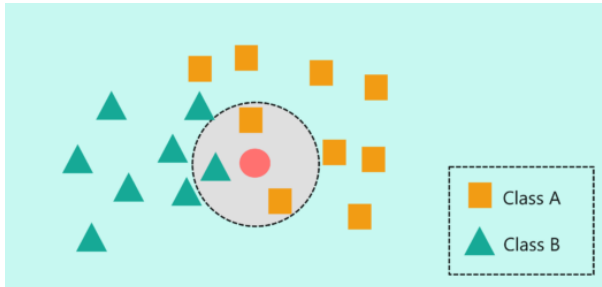
Professor: Marco Montes de Oca

- **A brief introduction to the topic.**

K-Nearest Neighbors algorithm (or KNN) is useful for machine learning because of its simplicity and flexibility. The simplicity of KNN can help model learn quickly and make it easier for analysts to interpret. The flexibility is that KNN can apply into both regression and classification model so analysts will be able to make use of it to optimize the result. The KNN model is determined by the parameter k, which classifies the data points based on the distance. The distance is calculated based on three distance metrics such as Euclidean Distance, Manhattan Distance, Chebyshev Distance. The Euclidean Distance is the most popular one so I will show its formula below and a brief explanation.

$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

The q1 is the unclassified data and p1 is the classified data, after the mathematic calculation, we can have the smaller results for each distance. For classification, we can sort the values to have the number of closest points according to the k parameter, then rank it with the majority of vote.



- **Code and output (or screenshots) with an explanation of what the code does.**

#LoadPackages and Dataset

Firstly, I want to load all necessary packages and data set to do the analysis on Python.

In [105]:

```
from scipy.io import arff
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn import metrics
from sklearn.metrics import confusion_matrix
```

The 'loadarff' function helps me to load dataset with file type as arff. After that, I use 'DataFrame' to create a table for data set with index and columns.

In [5]:

```
data_train = arff.loadarff('C:/Users/nickh/OneDrive - Northeastern Unive
data_test =arff.loadarff('C:/Users/nickh/OneDrive - Northeastern Univers

df_train = pd.DataFrame(data_train[0])
df_test = pd.DataFrame(data_test[0])
df_train.head()
```

Here is the brief 'DataFrame' of dataset with 721 columns representing for 721 attributes.

Out[5]:

	att1	att2	att3	att4	att5	att6	att7	att8
0	-0.099108	-0.099108	-0.099108	-0.099108	-0.099108	-0.099108	-0.099108	-0.099108
1	-0.155256	-0.155256	-0.155256	-0.155256	-0.155256	-0.155256	-0.155256	-0.155256
2	-0.100082	-0.100082	-0.100082	-0.100082	-0.100082	-0.100082	-0.100082	-0.100082
3	-0.140671	-0.140671	-0.140671	-0.140671	-0.140671	-0.140671	-0.140671	-0.140671
4	-0.140576	-0.140576	-0.140576	-0.140576	-0.140576	-0.140576	-0.140576	-0.140576

5 rows × 721 columns

‘Describe’ function will provide me the distribution of the attributes, so I can check whether the data is normal for predictive model.

In [31]:
```
df_train.describe()
```

Out[31]:

	att1	att2	att3	att4	att5	att6	a
count	375.000000	375.000000	375.000000	375.000000	375.000000	375.000000	375.0000
mean	0.088484	0.017864	-0.026741	-0.031323	-0.038759	-0.006532	-0.0013
std	1.790828	1.178260	0.938110	0.994872	1.028598	1.115328	1.0517
min	-0.686373	-0.686373	-0.686373	-0.686373	-0.686373	-0.686373	-0.6863
25%	-0.204997	-0.204997	-0.204997	-0.204566	-0.204997	-0.204997	-0.2057
50%	-0.168796	-0.168555	-0.168796	-0.168555	-0.168555	-0.168796	-0.1697
75%	-0.132184	-0.131642	-0.132184	-0.132660	-0.132660	-0.133968	-0.1326
max	26.781515	11.238558	10.630040	10.630040	10.374521	10.630040	8.0286

#Preprocessing Data for Model Building:

Before building model, we will split the data into train and test set to check the accuracy of train data when it is applied to the new instances. Because dataset is already split, I just need to load it and assign to 4 variables: X_train, y_train, X_test, y_test. Then, I have to convert the y label into numbers so it will not get type errors. ‘LabelEncoder’ is very easy for use, so I just need to import it, and use fit_transform to convert.

```
In [73]:  X_test = df_test.drop(columns='target',axis=1)
          y_test = df_test['target']
```

```
In [74]:  X_train = df_train.drop(columns='target',axis=1)
          y_train = df_train['target']
```

```
In [75]:  #creating labelEncoder
          le = preprocessing.LabelEncoder()
          # Converting string labels into numbers.
          y_train=le.fit_transform(y_train)
```

```
In [76]:  #creating labelEncoder
          le = preprocessing.LabelEncoder()
          # Converting string labels into numbers.
          y_test=le.fit_transform(y_test)
```

Now, I will be ready to start running predictive model with KNN function from sklearn package. I must apply the model of train set on test set to see how it works with new instances. The 'fit' function is used to apply the knn algorithm into model, and then we use 'predict' to find possible labels for x variables. The 'accuracy_score' is used to calculate the accuracy of model.

```
In [118]:  from sklearn.neighbors import KNeighborsClassifier
           from sklearn.metrics import accuracy_score
```

```
In [132]:  for i in range(1,21):
               knn = KNeighborsClassifier(n_neighbors=i)
               knn.fit(X_train, y_train)
               y_pred = knn.predict(X_train)
               # evaluate model
               accuracy = accuracy_score(y_train, y_pred)
               # report performance
               print('TRAIN:k',i,'Accuracy: %.3f' % (accuracy))
               y_pred = knn.predict(X_test)
               # evaluate model
               accuracy = accuracy_score(y_test, y_pred)
               # report performance
               print('TEST:k',i,'Accuracy: %.3f' % (accuracy))
               print('------------------------------------')
```

```
TRAIN: k1 Accu: 1.000
TEST: k1 Accu: 0.493
------------------------------------
TRAIN:k 2 Accu: 0.845
TEST:k 2 Accu: 0.475
------------------------------------
TRAIN:k 3 Accu: 0.771
TEST:k 3 Accu: 0.456
------------------------------------
TRAIN:k 4 Accu: 0.731
TEST:k 4 Accu: 0.440
------------------------------------
TRAIN:k 5 Accu: 0.699
TEST:k 5 Accu: 0.456
------------------------------------
TRAIN:k 6 Accu: 0.669
TEST:k 6 Accu: 0.429
------------------------------------
TRAIN:k 7 Accu: 0.600
TEST:k 7 Accu: 0.448
------------------------------------
```

TRAIN:k 8 Accu: 0.587
TEST:k 8 Accu: 0.448

TRAIN:k 9 Accu: 0.576
TEST:k 9 Accu: 0.437

TRAIN:k 10 Accu: 0.584
TEST:k 10 Accu: 0.419

TRAIN:k 11 Accu: 0.576
TEST:k 11 Accu: 0.435

TRAIN:k 12 Accu: 0.560
TEST:k 12 Accu: 0.448

TRAIN:k 13 Accu: 0.547
TEST:k 13 Accu: 0.427

TRAIN:k 14 Accu: 0.528
TEST:k 14 Accu: 0.421

TRAIN:k 15 Accu: 0.531
TEST:k 15 Accu: 0.427

TRAIN:k 16 Accu: 0.520
TEST:k 16 Accu: 0.405

TRAIN:k 17 Accu: 0.539
TEST:k 17 Accu: 0.408

TRAIN:k 18 Accu: 0.517
TEST:k 18 Accu: 0.397

TRAIN:k 19 Accu: 0.533
TEST:k 19 Accu: 0.421

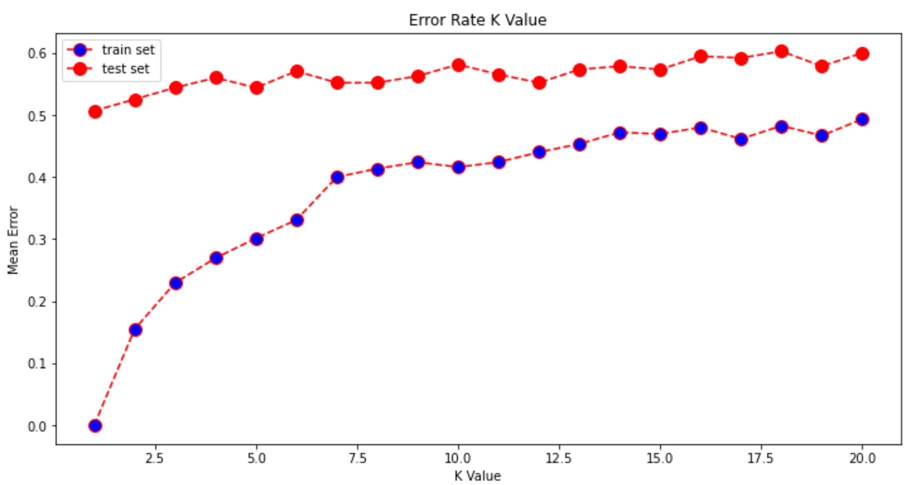
TRAIN:k 20 Accu: 0.507
TEST:k 20 Accu: 0.400

#Evaluate the performance of predictive model:

To see how the model performs well with different k, it is necessary to check its error rate. Firstly, I create the list of train error and test error, then I assign the mean of incorrect predictions between X variables and y predicted labels. After that, I plot it by the line chart to see the changes of errors rate within 20 different k parameters.

```
In [133]:  train_error = []
           test_error = []
           # Calculating error for K values between 1 and 21
           for i in range(1, 21):
               knn = KNeighborsClassifier(n_neighbors=i)
               knn.fit(X_train, y_train)
               pred_i = knn.predict(X_train)
               train_error.append(np.mean(pred_i != y_train))
               pred_i = knn.predict(X_test)
               test_error.append(np.mean(pred_i != y_test))
```

```
In [137]:  plt.figure(figsize=(12, 6))
           plt.plot(range(1, 21), train_error, color='red', linestyle='dashed', mar
                    markerfacecolor='blue', markersize=10,label='train set')
           plt.plot(range(1, 21), test_error, color='red', linestyle='dashed', mark
                    markerfacecolor='red', markersize=10,label='test set')
           plt.title('Error Rate K Value')
           plt.legend()
           plt.xlabel('K Value')
           plt.ylabel('Mean Error')
```



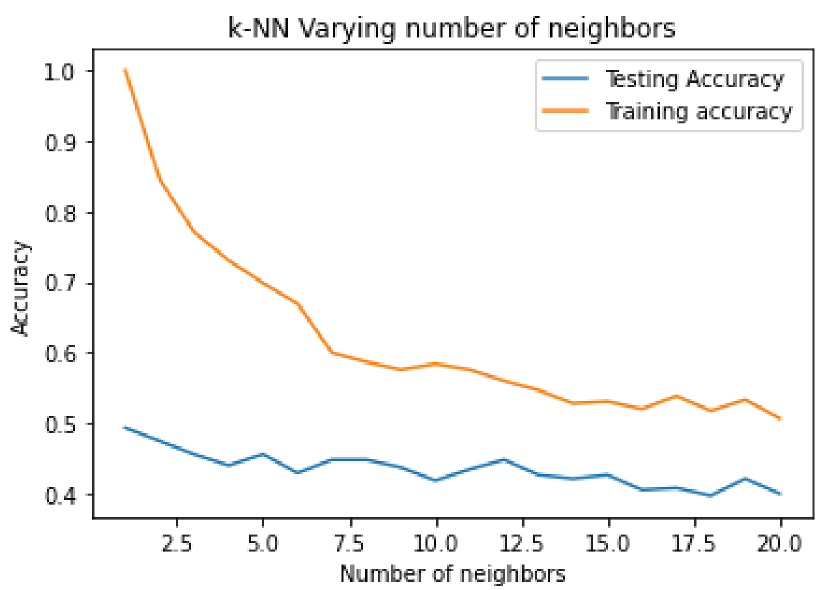
Another way is to test the accuracy of models, the method is quite similar but instead of calculating the error rate, I use ‘score’ from ‘accuracy_score’ to find the accuracy. So, I will be able to visualize the different accuracy results when running different k parameters on train and test set.

In [102]:

```
neighbors = np.arange(1,21)
train_accuracy =np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

for i,k in enumerate(neighbors):
    #Setup a knn classifier with k neighbors
    knn = KNeighborsClassifier(n_neighbors=k)
    #Fit the model
    knn.fit(X_train, y_train)
    #Compute accuracy on the training set
    train_accuracy[i] = knn.score(X_train, y_train)
    #Compute accuracy on the test set
    test_accuracy[i] = knn.score(X_test, y_test)
```

In [103]:

```
#Generate plot
plt.title('k-NN Varying number of neighbors')
plt.plot(neighbors, test_accuracy, label='Testing Accuracy')
plt.plot(neighbors, train_accuracy, label='Training accuracy')
plt.legend()
plt.xlabel('Number of neighbors')
plt.ylabel('Accuracy')
plt.show()
```



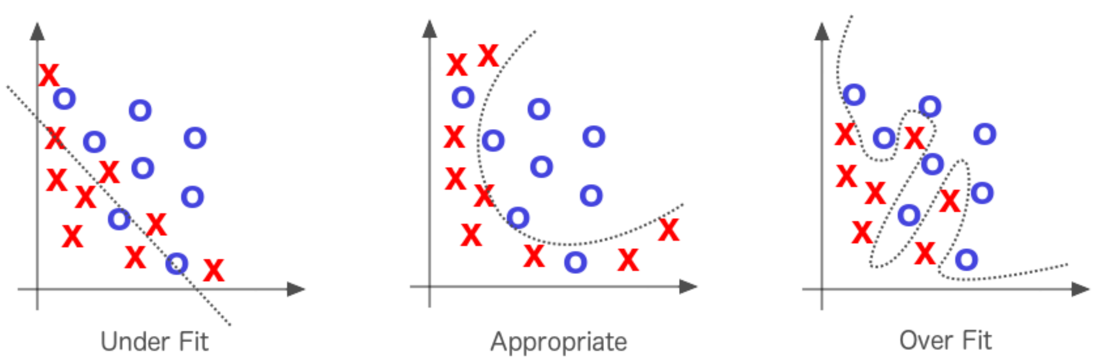
After deciding the k parameter, I want to create the confusion matrix to see how it performs on test set.

In [28]:
```
knn = KNeighborsClassifier(n_neighbors=15)
#Fit the model
knn.fit(X_train, y_train)
#Compute accuracy on the training set
y_pred = knn.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
```

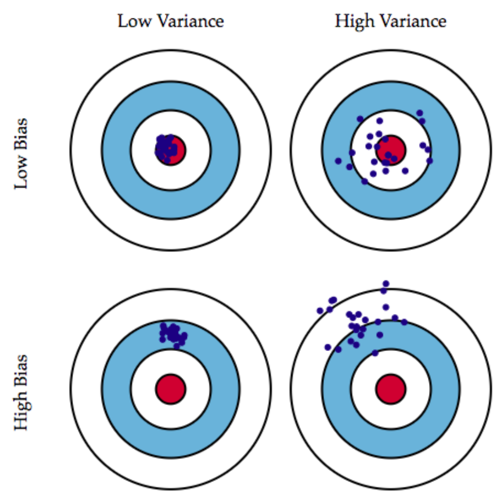
In [30]:
```
# Transform to df for easier plotting
cm_df = pd.DataFrame(cm,
                     index = ['b1','b2','b3'],
                     columns = ['b1','b2','b3'])

sns.heatmap(cm_df, annot=True)
plt.title('Accuracy:{0:.3f}'.format(accuracy_score(y_test, y_pre
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()
```

- ***A critical analysis of the results. This means that you must try to explain why the reported numbers behave the way they do. It is not enough to just report data.***

After having the accuracy result, we can learn that the smaller k parameter can make train model more accurate. For example, at K around 1-3, the model can perform with less than 50% errors for test set and 0% errors for train set. However, this is not what we need only to decide which model to be used in company. The fact is that our model is overfitting since the accuracy of train set is far higher than test set. The overfitting happens when the model fits data so well, but it will perform poor on new data set. Moreover, the small k could be the reason of noisier model so increasing the k will give smoother decision boundaries or increase bias. However, larger k also has problem with too high variance, which makes model complicated.

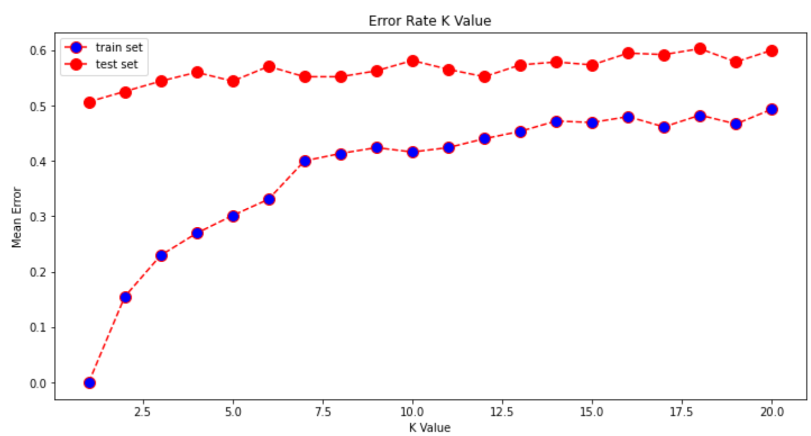


Therefore, when it comes to the trade-offs between bias and variance, we should find the most suitable k to lower variance with low bias.

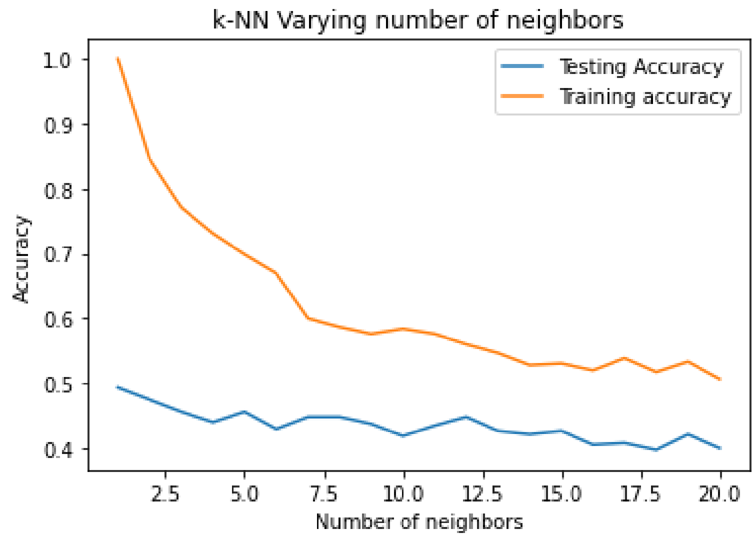


Graphical illustration of bias and variance. Credit: http://scott.fortmann-roe.com/docs/BiasVariance.html

#Best K for Trade-offs

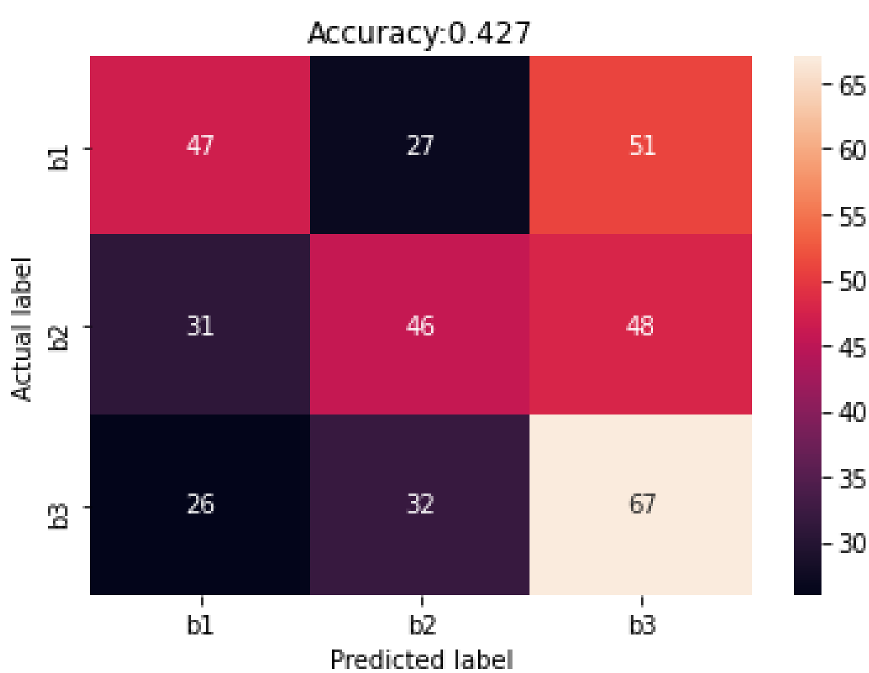


So, to find the good trade-offs between train set and test set, we can check where those two lines fit each other the most, so the model should perform better if the difference between both model's error rate is not much. Therefore, we can see the k from 12-15 is appropriate for model. The accuracy visualization also shows that the similar k will give us the balance of train and test set. In addition, if we go further with k, the accuracy will decrease because our data set is not too huge, and this can cause the complicate for model.




```
------------------------------------
TRAIN:k 12  Accu: 0.560
TEST:k 12  Accu: 0.448
------------------------------------
TRAIN:k 13  Accu: 0.547
TEST:k 13  Accu: 0.427
------------------------------------
TRAIN:k 14  Accu: 0.528
TEST:k 14  Accu: 0.421
------------------------------------
TRAIN:k 15  Accu: 0.531
TEST:k 15  Accu: 0.427
------------------------------------
```

As we can see, the k 15 is an ideal parameter to keep high accuracy for both data set and avoid overfitting.



With k = 15, the accuracy of test set is 0.427, and the confusion matrix also shows us there are many misclassifications. For example, there are 48 ‘b2’ misclassified as ‘b3’.

Reference:

Subramanian. D (2019). A Simple Introduction to K-Nearest Neighbors Algorithm. Retrieved from https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm-b3519ed98e