QQP4 - ML Models - Random Model & K-Nearest Neighbour Model

April 27, 2018

In [1]:
```
# Loading the required libraries:
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cross_validation import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.cross_validation import cross_val_score
from collections import Counter
from sklearn import cross_validation
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\cross_validation.py:41: DeprecationWarning:
  "This module will be removed in 0.20.", DeprecationWarning)
```

In [2]:
```
from time import time

# Some helper functions:
def get_shape(seq):
    if type(seq) == type([]):
        print("The shape of data is:", len(seq),",",len(seq[0]))
    else:
        print("The shape of data is:", seq.shape)
    return

def time_taken(start):
    print("\nRuntime:", round(time()-start, 2), "seconds")
    return
```

1 4. Machine Learning Models

We will apply two ML Algorithms and generate 2 models: 1. Random Modelling Algorithm on a Sample of Quora Question Pairs Data. 2. K-Nearest Neighbour Algorithm on a Sample of Quora Question Pairs Data.

1.1 4.1 Loading Data

In [3]:
```
st = time()
# Load only a sample of the final features data:
quora_df = pd.read_csv('./final_features_100k.csv', nrows=25000)

# Data Info:
print(type(quora_df))
get_shape(quora_df)
time_taken(st)
quora_df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
The shape of data is: (25000, 797)

Runtime: 7.92 seconds
```

Out[3]:

	Unnamed: 0	id	is_duplicate	cwc_min	cwc_max	csc_min	csc_max	\
0	0	0	0	0.999980	0.833319	0.999983	0.999983	
1	1	1	0	0.799984	0.399996	0.749981	0.599988	
2	2	2	0	0.399992	0.333328	0.399992	0.249997	
3	3	3	0	0.000000	0.000000	0.000000	0.000000	
4	4	4	0	0.399992	0.199998	0.999950	0.666644	

	ctc_min	ctc_max	last_word_eq	...	374_y	375_y	\
0	0.916659	0.785709	0.0	...	16.188503	33.233713	
1	0.699993	0.466664	0.0	...	-4.432317	-4.367793	
2	0.399996	0.285712	0.0	...	8.264448	-2.244750	
3	0.000000	0.000000	0.0	...	3.488654	3.906499	
4	0.571420	0.307690	0.0	...	-2.440844	11.887040	

	376_y	377_y	378_y	379_y	380_y	381_y	\
0	6.971700	-14.820828	15.534945	8.205955	-25.256606	1.552828	
1	41.101273	-0.930737	-15.686246	-7.275999	2.756560	-7.351970	
2	11.084606	-16.741266	14.854023	15.726977	-1.298039	14.340431	
3	13.387563	-6.640244	6.378005	6.028185	2.511873	-3.830347	
4	8.019029	-15.028031	8.280575	1.703147	-6.503707	11.263387	

	382_y	383_y
0	1.651827	0.267462
1	3.103773	0.440425
2	11.669012	10.423255
3	5.421078	6.161891
4	11.556818	2.500520

[5 rows x 797 columns]

In [4]:
```
quora_df.tail()
```

Out[4]:

	Unnamed: 0	id	is_duplicate	cwc_min	cwc_max	csc_min	csc_max	ctc_min	ctc_max	last_word_eq	...	374_y	375_y	376_y	377_y	378_y	379_y	380_y	381_y	382_y	383_y
24995	24995	24995	0	0.999950	0.666644	0.999950	0.499988	0.999975	0.571420	1.0	...	7.164913	23.039302	5.387981	0.075391	5.558455	-2.099565	-2.638070	-7.089084	-5.356800	10.654196
24996	24996	24996	0	0.666644	0.666644	0.799984	0.799984	0.749991	0.749991	0.0	...	2.033602	15.150122	6.668069	-2.651128	13.367594	5.552914	-11.574743	0.417787	7.598512	1.206826
24997	24997	24997	0	0.599988	0.428565	0.000000	0.000000	0.299997	0.299997	0.0	...	-2.951352	0.726570	5.036810	-6.168143	-8.204763	6.214043	-4.996871	5.989402	12.023006	-12.509062
24998	24998	24998	1	0.499988	0.399992	0.499975	0.199996	0.499992	0.299997	0.0	...	9.845878	24.765205	5.713985	-8.277235	14.101339	3.312259	-7.348258	4.242257	22.634850	0.331471
24999	24999	24999	0	0.999967	0.749981	0.999983	0.857131	0.999989	0.818174	0.0	...	5.514413	2.062461	2.855732	-5.723221	-0.315934	8.459152	-7.826173	3.594841	1.028608	1.307878

[5 rows x 797 columns]

In [5]:
```
# We will drop some useless features:
y_class = quora_df['is_duplicate']
quora_df.drop(['id', 'is_duplicate', 'Unnamed: 0'], axis=1, inplace=True)
quora_df.head()
```

Out[5]:

	cwc_min	cwc_max	csc_min	csc_max	ctc_min	ctc_max	last_word_eq	first_word_eq	abs_len_diff	mean_len	...	374_y	375_y	376_y	377_y	378_y	379_y	380_y	381_y	382_y	383_y
0	0.999980	0.833319	0.999983	0.999983	0.916659	0.785709	0.0	1.0	0.0	13.0	...	16.188503	33.233713	5.387981	0.075391	5.558455	-2.099565	-2.638070	-7.089084	-5.356800	10.654196
1	0.799984	0.399996	0.749981	0.599988	0.699993	0.466664	0.0	1.0	0.0	12.5	...	-4.432317	-4.367793	6.668069	-2.651128	13.367594	5.552914	-11.574743	0.417787	7.598512	1.206826
2	0.399992	0.333328	0.399992	0.249997	0.399996	0.285712	0.0	1.0	0.0	12.0	...	8.264448	-2.244750	5.036810	-6.168143	-8.204763	6.214043	-4.996871	5.989402	12.023006	-12.509062
3	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	12.0	...	3.488654	3.906499	5.713985	-8.277235	14.101339	3.312259	-7.348258	4.242257	22.634850	0.331471
4	0.399992	0.199998	0.999950	0.666644	0.571420	0.307690	0.0	1.0	0.0	10.0	...	-2.440844	11.887040	2.855732	-5.723221	-0.315934	8.459152	-7.826173	3.594841	1.028608	1.307878

```
       376_y      377_y      378_y      379_y      380_y      381_y  \
0   6.971700 -14.820828  15.534945   8.205955 -25.256606   1.552828
1  41.101273  -0.930737 -15.686246  -7.275999   2.756560  -7.351970
2  11.084606 -16.741266  14.854023  15.726977  -1.298039  14.340431
3  13.387563  -6.640244   6.378005   6.028185   2.511873  -3.830347
4   8.019029 -15.028031   8.280575   1.703147  -6.503707  11.263387

       382_y      383_y
0   1.651827   0.267462
1   3.103773   0.440425
2  11.669012  10.423255
3   5.421078   6.161891
4  11.556818   2.500520

[5 rows x 794 columns]
```

In [6]:
```
# We have our class variable as:
y_class.head()
```

Out[6]:
```
0    0
1    0
2    0
3    0
4    0
Name: is_duplicate, dtype: int64
```

In [8]:
```
type(y_class)
```

Out[8]:
```
pandas.core.series.Series
```

1.2 4.2 Converting strings to numerics

In [7]:
```
st = time()

cols = list(quora_df.columns)
for i in cols:
    quora_df[i] = quora_df[i].apply(pd.to_numeric)
    print(i, end=", ")
```

```
cwc_min, cwc_max, csc_min, csc_max, ctc_min, ctc_max, last_word_eq, first_word_eq, abs_len_dif
```

In [8]:
```
time_taken(st)
```

```
Runtime: 87.6 seconds
```

1.3 4.3 Random Train-Test Split (70:30)

```
In [9]: # Our class variable is y_class:
        X_train, X_test, y_train, y_test = train_test_split(
            quora_df, y_class, stratify=y_class, test_size=0.3
        )

        print('Number of data points in train data:', X_train.shape)
        print('Number of data points in test data :', X_test.shape)
```

```
Number of data points in train data: (17500, 794)
Number of data points in test data : (7500, 794)
```

```
In [10]: print(type(X_train))
         X_train.tail()
```

```
<class 'pandas.core.frame.DataFrame'>
```

Out[10]:

	cwc_min	cwc_max	csc_min	csc_max	ctc_min	ctc_max	\
14738	0.749981	0.599988	0.666644	0.666644	0.714276	0.624992	
22866	0.749981	0.749981	0.749981	0.749981	0.749991	0.749991	
24591	0.499975	0.499975	0.999900	0.999900	0.666644	0.666644	
20393	0.499988	0.399992	0.599988	0.333330	0.555549	0.333331	
21438	0.749981	0.749981	0.749981	0.599988	0.749991	0.666659	

	last_word_eq	first_word_eq	abs_len_diff	mean_len	...	\
14738	0.0	1.0	0.0	7.5	...	
22866	1.0	1.0	0.0	8.0	...	
24591	1.0	1.0	0.0	3.0	...	
20393	0.0	0.0	0.0	12.0	...	
21438	0.0	1.0	0.0	8.5	...	

	374_y	375_y	376_y	377_y	378_y	379_y	\
14738	6.902403	11.493473	5.613606	-4.877812	13.043599	-1.132098	
22866	3.047488	2.449508	-5.921124	3.526991	10.510124	2.405635	
24591	1.473008	-2.153633	0.372134	3.989067	-2.932041	-1.881972	
20393	-1.791243	13.849159	1.789727	-7.466151	11.052734	5.252021	
21438	4.836686	9.265958	4.292188	-4.779898	13.177334	9.700841	

	380_y	381_y	382_y	383_y
14738	-4.538146	-3.301893	7.908649	-0.451902
22866	-7.946949	2.644402	4.655404	0.412322
24591	-0.737835	15.230009	-3.082860	-5.839749
20393	-16.688383	4.642186	12.817753	5.000661
21438	-5.874484	5.485653	7.848238	3.065879

[5 rows x 794 columns]

In [11]:
```
X_test.tail()
```

Out[11]:

```
        cwc_min   cwc_max   csc_min   csc_max   ctc_min   ctc_max  \
18463  0.333322  0.333322  0.999975  0.999975  0.714276  0.714276
4039   0.999980  0.999980  0.999950  0.666644  0.999986  0.874989
9761   0.666644  0.399992  0.749981  0.374995  0.714276  0.333331
12817  0.999967  0.499992  0.999950  0.399992  0.999980  0.454541
16519  0.142855  0.076922  0.499988  0.285710  0.272725  0.149999

       last_word_eq  first_word_eq  abs_len_diff  mean_len    ...     \
18463           0.0            1.0           0.0       7.0    ...
4039            1.0            1.0           0.0       7.5    ...
9761            0.0            0.0           0.0      11.0    ...
12817           0.0            1.0           0.0       8.0    ...
16519           0.0            0.0           0.0      15.5    ...

          374_y     375_y     376_y     377_y      378_y     379_y      380_y  \
18463  1.600818  8.941236  4.398032 -3.078369  13.817652  7.158675 -11.921655
4039   0.609111  8.085004  7.153885  1.884657 -11.188039  1.521977   0.628922
9761   6.810799  1.547490  3.335044 -3.722337  14.990891  8.770758  -7.148856
12817 -5.335906 -3.942303 -5.758041 -1.118189  11.369048 -5.043033 -10.508012
16519 -6.623911 -8.571070  3.896555 -2.758711  -9.771975 -3.437023 -14.918431

           381_y      382_y     383_y
18463   7.725214   4.514453 -0.636877
4039    2.914077  -0.467034  4.371045
9761    5.816961  11.357115  5.122552
12817  -4.148119 -15.397742  1.834784
16519  16.211051  17.536233 -2.204456

[5 rows x 794 columns]
```

In [12]:
```
get_shape(y_train)
y_train.tail()
```

```
The shape of data is: (17500,)
```

Out[12]:

```
14738    1
22866    0
24591    1
20393    1
21438    0
Name: is_duplicate, dtype: int64
```

In [13]:
```
get_shape(y_test)
y_test.tail()
```

```
The shape of data is: (7500,)
```

```
Out[13]: 18463    0
         4039     1
         9761     0
         12817    1
         16519    0
         Name: is_duplicate, dtype: int64
```

```
In [14]: # Now we will see the distribution of points classwise:
         print("-"*10, "Distribution of O/P Variable in train data", "-"*10)
         tr_disb = Counter(y_train)
         print("Number of data points that correspond to 'is_duplicate = 0' are:", tr_disb[0])
         print("Number of data points that correspond to 'is_duplicate = 1' are:", tr_disb[1])
         tr_len = len(y_train)
         print("Total Number of points in train:", tr_len, "\n")
         print("O/P (or) class-label: 'is_duplicate'")
         print("is_duplicate = 0:", float(tr_disb[0]/tr_len),
               "\nis_duplicate = 1:", float(tr_disb[1]/tr_len))
```

```
---------- Distribution of O/P Variable in train data ----------
Number of data points that correspond to 'is_duplicate = 0' are: 10986
Number of data points that correspond to 'is_duplicate = 1' are: 6514
Total Number of points in train: 17500

O/P (or) class-label: 'is_duplicate'
is_duplicate = 0: 0.6277714285714285
is_duplicate = 1: 0.3722285714285714
```

1.4 4.3 Building a Random Model

We will find the worst case accuracy score using a random model.

```
In [15]: # We will create a list which has exactly same size as our test data
         # and generate a non-uniform random model:
         random_y = np.random.choice(2, size=len(y_test),
                                     p = [0.1, 0.9]
                                    )
         random_y
```

```
Out[15]: array([1, 1, 1, ..., 1, 0, 1], dtype=int64)
```

```
In [16]: # Now we check the accuracy score:
         rand_acc = accuracy_score(random_y, y_test, normalize=True) * float(100)
         print("Accuracy Score for Random Model is {}%".format(rand_acc))
```

```
Accuracy Score for Random Model is 39.800000000000004%
```

With a Random Model, we are getting ~40% Accuracy, i.e., Our Random Model is able to predict whether 2 questions are similar or not, correctly, only 50% of the time. Therefore, this is the worst case Accuracy Score.

We want our k-NN to get an Accuracy Score > 40%.

1.5 4.4 Building k-Nearest Neighbours Model using Simple Cross Validation

In [17]:
```
# Split the train data into cross validation train and cross validation test
X_tr, X_cv, y_tr, y_cv = train_test_split(
    X_train, y_train, stratify=y_train, test_size=0.3
)

# train and cv data info:
print('Number of data points in train data:', X_tr.shape)
print('Number of data points in cross validation data :', X_cv.shape)
```

```
Number of data points in train data: (12250, 794)
Number of data points in cross validation data : (5250, 794)
```

In [18]:
```
# Now we will see the distribution of points classwise:
print("-"*15, "Distribution of O/P Variable in train data", "-"*15)
train_tr_disb = Counter(y_tr)
print("Number of data points that correspond to 'is_duplicate = 0' are:",
      train_tr_disb[0])
print("Number of data points that correspond to 'is_duplicate = 1' are:",
      train_tr_disb[1])
train_tr_len = len(y_tr)
print("Total Number of points in train:", train_tr_len, "\n")
print("O/P (or) class-label: 'is_duplicate'")
print("is_duplicate = 0:", float(train_tr_disb[0]/train_tr_len),
     "\nis_duplicate = 1:", float(train_tr_disb[1]/train_tr_len))
```

```
--------------- Distribution of O/P Variable in train data ---------------
Number of data points that correspond to 'is_duplicate = 0' are: 7690
Number of data points that correspond to 'is_duplicate = 1' are: 4560
Total Number of points in train: 12250

O/P (or) class-label: 'is_duplicate'
is_duplicate = 0: 0.6277551020408163
is_duplicate = 1: 0.3722448979591837
```

Hyper Parameter Selection (or) Selection of Optimal K

In [20]:
```
# Finding the right k and applying k-NN using simple cross-validation:
# Hyper parameter selection:

# Creating odd list of K for K-NN:
my_list = list(range(0,100))
neighbours = list(filter(lambda x: x%2 != 0, my_list))
print("We will test K-NN Algorithm for these values of K:\n")
for i in neighbours:
    print(i, end=' ')
```

We will test K-NN Algorithm for these values of K:

1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 6

In [21]:
```
# Now we have all the odd numbers, we can now apply the sklearn
# implementation of KNN to know the similarity/polarity between two questions:

st = time()

# Code for hyper parameter selection:
for k in neighbours:

    # Configured parameters are:-
    #
    # 1. algorithm = 'auto':
    #    automatically choose the algorithm (KDTree, BallTree or Brute Force)
    #
    # 2. metric = 'minkowski', p = 2:
    #    Use L2 Minkowski Distance which is nothing but Euclidean Distance.
    #
    # 3. n_jobs = -1:
    #    Use all the CPU cores to apply KNN Classfication.

    # Instantiate the learning model:
    knn = KNeighborsClassifier(
        n_neighbors = k,
        algorithm = 'auto',
        metric = 'minkowski',
        p = 2,
        n_jobs = -1
    )

    # Fitting the model on train:
    knn.fit(X_tr, y_tr)

    # Predict the response on cross validation:
    predict_y_cv = knn.predict(X_cv)

    # Evaluate the cross validation accuracy:
    acc = accuracy_score(predict_y_cv, y_cv, normalize=True) * float(100)
    print('\nCross Validation Accuracy for k={} is {}%'
          .format(k, acc))

time_taken(st)
```

```
Cross Validation Accuracy for k=1 is 61.96190476190476%
```

```
Cross Validation Accuracy for k=3 is 63.29523809523809%

Cross Validation Accuracy for k=5 is 63.21904761904762%

Cross Validation Accuracy for k=7 is 63.847619047619055%

Cross Validation Accuracy for k=9 is 64.07619047619048%

Cross Validation Accuracy for k=11 is 64.11428571428571%

Cross Validation Accuracy for k=13 is 64.32380952380953%

Cross Validation Accuracy for k=15 is 64.36190476190477%

Cross Validation Accuracy for k=17 is 64.4%

Cross Validation Accuracy for k=19 is 64.64761904761905%

Cross Validation Accuracy for k=21 is 64.72380952380952%

Cross Validation Accuracy for k=23 is 64.47619047619048%

Cross Validation Accuracy for k=25 is 64.64761904761905%

Cross Validation Accuracy for k=27 is 65.16190476190476%

Cross Validation Accuracy for k=29 is 65.18095238095239%

Cross Validation Accuracy for k=31 is 64.91428571428571%

Cross Validation Accuracy for k=33 is 65.04761904761904%

Cross Validation Accuracy for k=35 is 64.62857142857142%

Cross Validation Accuracy for k=37 is 64.32380952380953%

Cross Validation Accuracy for k=39 is 64.13333333333333%

Cross Validation Accuracy for k=41 is 64.22857142857143%

Cross Validation Accuracy for k=43 is 64.03809523809524%

Cross Validation Accuracy for k=45 is 64.4%

Cross Validation Accuracy for k=47 is 64.53333333333333%

Cross Validation Accuracy for k=49 is 64.15238095238095%
```

```
Cross Validation Accuracy for k=51 is 64.51428571428572%

Cross Validation Accuracy for k=53 is 64.59047619047618%

Cross Validation Accuracy for k=55 is 64.4%

Cross Validation Accuracy for k=57 is 64.28571428571429%

Cross Validation Accuracy for k=59 is 64.24761904761904%

Cross Validation Accuracy for k=61 is 64.28571428571429%

Cross Validation Accuracy for k=63 is 64.41904761904762%

Cross Validation Accuracy for k=65 is 64.24761904761904%

Cross Validation Accuracy for k=67 is 64.03809523809524%

Cross Validation Accuracy for k=69 is 64.15238095238095%

Cross Validation Accuracy for k=71 is 63.98095238095238%

Cross Validation Accuracy for k=73 is 63.866666666666674%

Cross Validation Accuracy for k=75 is 63.82857142857142%

Cross Validation Accuracy for k=77 is 63.94285714285714%

Cross Validation Accuracy for k=79 is 64.05714285714285%

Cross Validation Accuracy for k=81 is 64.15238095238095%

Cross Validation Accuracy for k=83 is 64.11428571428571%

Cross Validation Accuracy for k=85 is 63.714285714285715%

Cross Validation Accuracy for k=87 is 63.90476190476191%

Cross Validation Accuracy for k=89 is 63.82857142857142%

Cross Validation Accuracy for k=91 is 63.67619047619048%

Cross Validation Accuracy for k=93 is 63.542857142857144%

Cross Validation Accuracy for k=95 is 63.714285714285715%

Cross Validation Accuracy for k=97 is 63.61904761904762%
```

```
Cross Validation Accuracy for k=99 is 63.88571428571429%

Runtime: 2152.39 seconds
```

Cross Validation Accuracy for k=29 is 65.18095238095239%. This is highest accuracy score out of all the accuracy scores.

Therefore, we got our k=29, i.e., we will consider the majority vote of the classes of 29 nearest neighbours in the vicinity of a query point -> xq.

1.5.1 Applying the K Value from Simple Cross Validation on Test Data

In [23]:
```
# Configured parameters are:-
#
# 1. algorithm = 'auto':
#    automatically choose the algorithm (KDTree, BallTree or Brute Force)
#
# 2. metric = 'minkowski', p = 2:
#    Use L2 Minkowski Distance which is nothing but Euclidean Distance.
#
# 3. n_jobs = -1:
#    Use all the CPU cores to apply KNN Classfication.

# Instantiate the learning model with k=29:
k_simple = 29
knn_simple_cv = KNeighborsClassifier(
    n_neighbors = k_simple,
    algorithm = 'auto',
    metric = 'minkowski',
    p = 2,
    n_jobs = -1
)

# Fitting the model on train data:
knn_simple_cv.fit(X_tr, y_tr)

# Predict the response on test data:
predict_y_test_simple_cv = knn_simple_cv.predict(X_test)

# Evaluate the test accuracy:
acc_test_simple = accuracy_score(predict_y_test_simple_cv, y_test, normalize=True) * 
print('\n****** Test Accuracy for k={} is {}% *******'
      .format(k_simple, acc_test_simple))
```

```
****** Test Accuracy for k=29 is 64.62666666666667% *******
```

We will now apply K-NN using K-fold Cross Validation to get the best K, so that we can classify whether question1 is similar to question2 or not.

1.6 4.5 Building k-Nearest Neighbours Model using K-fold Cross Validation

Here, the k used for k-NN is a Hyper Parameter which tells us the number of neighbours that the algorithm is considering before making a decision about the class of a query point.

But, K used in K-fold Cross Validation is the number of folds/divisions we are making in our data, to consider the data as train and cross validation data with different division each time. After we get scores for each division, we take the mean of all of the scores, and that's our accuracy score of the k-fold cross validation.

K = 10: 10 fold Cross Validation

In [24]:
```
# Empty list to store the cross validation scores:
cv_scores = []

st = time()
# Perform 10-fold cross validation:
for k in neighbours:
        # Configured parameters are:-
    #
    # 1. algorithm = 'auto':
    #    automatically choose the algorithm (KDTree, BallTree or Brute Force)
    #
    # 2. metric = 'minkowski', p = 2:
    #    Use L2 Minkowski Distance which is nothing but Euclidean Distance.
    #
    # 3. n_jobs = -1:
    #    Use all the CPU cores to apply KNN Classfication.

    # Instantiate the learning model:
    knn = KNeighborsClassifier(
        n_neighbors = k,
        algorithm = 'auto',
        metric = 'minkowski',
        p = 2,
        n_jobs = 3
    )

    # cv = 10: meaning 10 folds in the given data to get combinations
    # of train and cross validation data
    scores = cross_val_score(
        knn, X_train, y_train, cv=10, scoring='accuracy'
    )

    # record all the scores until now:
    cv_scores.append(scores.mean())
```

```
In [25]: time_taken(st)
```

```
Runtime: 13361.46 seconds
```

```
In [32]: for i in cv_scores:
             print(i, end=', ')
```

```
0.629255119617, 0.639312820543, 0.641598535053, 0.640970257035, 0.645256919435, 0.644399122845
```

```
In [33]: # Changing to Misclassification error:
         MSE = [1-x for x in cv_scores]

         for i in MSE:
             print(i, end=', ')
```

```
0.370744880383, 0.360687179457, 0.358401464947, 0.359029742965, 0.354743080565, 0.355600877155
```

```
In [34]: # Now, we will determine the best k:
         optimal_k = neighbours[MSE.index(min(MSE))]
         print("The optimal number of neighbours is:", optimal_k)
```

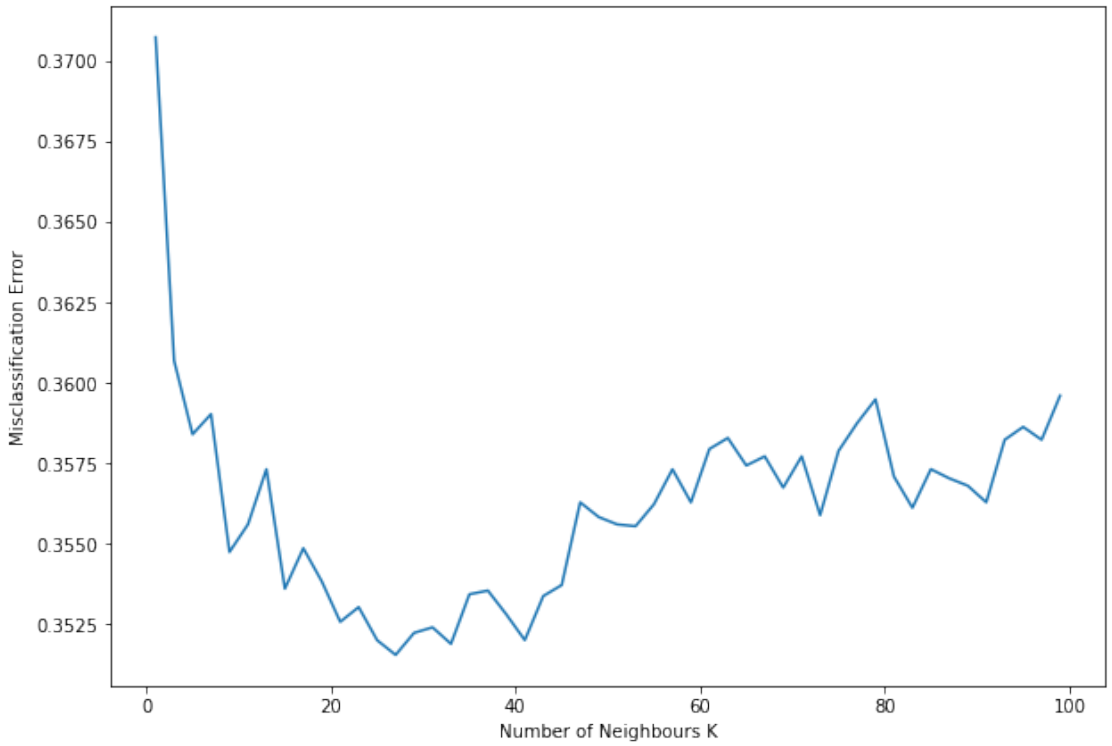
```
The optimal number of neighbours is: 27
```

```
In [37]: # Plot the Misclassification Error v/s k:
         plt.figure(figsize=(10,7))
         plt.plot(neighbours, MSE)

         #for xy in zip(neighbours, np.round(MSE, 2)):
             #plt.annotate('(%s, %s)' % xy, xy=xy, textcoords='data')

         plt.xlabel('Number of Neighbours K')
         plt.ylabel('Misclassification Error')
         plt.show()

         print("The Miscalssification Error for each k value is:\n", np.round(MSE, 3))
```




```
The Miscalssification Error for each k value is:
 [ 0.371  0.361  0.358  0.359  0.355  0.356  0.357  0.354  0.355  0.354
  0.353  0.353  0.352  0.352  0.352  0.352  0.352  0.353  0.354  0.353
  0.352  0.353  0.354  0.356  0.356  0.356  0.356  0.356  0.357  0.356
  0.358  0.358  0.357  0.358  0.357  0.358  0.356  0.358  0.359  0.359
  0.357  0.356  0.357  0.357  0.357  0.356  0.358  0.359  0.358  0.36 ]
```

From the plot above, we can see that the lowest value of Misclassification error is generated in between k=[20, 21, ..., 40]. That's the reason, we got our optimal_k to be 27.

Let us see the accuracy score after querying the k-NN model with the test data.

```
In [40]: # KNN with k = optimal_k
         st = time()
         # Configured parameters are:-
         #
         # 1. algorithm = 'auto':
         #    automatically choose the algorithm (KDTree, BallTree or Brute Force)
         #
         # 2. metric = 'minkowski', p = 2:
         #    Use L2 Minkowski Distance which is nothing but Euclidean Distance.
         #
         # 3. n_jobs = -1:
```

```
        #    Use all the CPU cores to apply KNN Classfication.

        # Instantiate the learning model:
        knn_optimal = KNeighborsClassifier(
            n_neighbors = optimal_k,
            algorithm = 'auto',
            metric = 'minkowski',
            p = 2,
            n_jobs = 3
        )

        # Fitting the model on train:
        knn_optimal.fit(X_train, y_train)

        # Predict the response on test:
        predict_y_test = knn_optimal.predict(X_test)

        # Evaluate the test accuracy:
        acc_test = accuracy_score(predict_y_test, y_test, normalize=True) * float(100)
        print('''\nThe Accuracy of k-NN classifier on Quora Question Pairs Dataset
        for predicting whether two given questions have the same intent or not with
        k={} is {}%'''.format(optimal_k, acc_test))

        time_taken(st)
```

```
The Accuracy of k-NN classifier on Quora Question Pairs Dataset
for predicting whether two given questions have the same intent or not with
k=27 is 65.70666666666666%

Runtime: 111.65 seconds
```