Lab-glass-02

November 16, 2021

1 Data for glass (Normalization & prediction)

[15]:
```
# Class of k-Nearest Neigbor Classifier


class kNN():
    maxTrain = []

    def __init__(self, k = 3, exp = 2):
    # constructor for kNN classifier
    # k is the number of neighbor for local class estimation
    # exp is the exponent for the Minkowski distance
        self.k = k
        self.exp = exp

    def fit(self, X_train, Y_train):
    # training k-NN method
    # X_train is the training data given with input attributes. n-th row␣
↪correponds to n-th instance.
    # Y_train is the output data (output vector): n-th element of Y_train is␣
↪the output value for n-th instance in X_train.
        self.X_train = X_train
        self.Y_train = Y_train

    def getDiscreteClassification(self, X_test):
    # predict-class k-NN method
    # X_test is the test data given with input attributes. Rows correpond to␣
↪instances
    # Method outputs prediction vector Y_pred_test:  n-th element of␣
↪Y_pred_test is the prediction for n-th instance in X_test

        Y_pred_test = [] #prediction vector Y_pred_test for all the test␣
↪instances in X_test is initialized to empty list []
```

```
        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances = []  #list of distances of the i-th test_instance for␣
↪all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances␣
↪in X_train
                train_instance = self.X_train.iloc[j] #j-th training␣
↪instance
                distance = self.Minkowski_distance(test_instance,␣
↪train_instance) #distance between i-th test instance and j-th training␣
↪instance
                distances.append(distance) #add the distance to the list of␣
↪distances of the i-th test_instance

            # Store distances in a dataframe. The dataframe has the index of␣
↪Y_train in order to keep the correspondence with the classes of the training␣
↪instances
            df_dists = pd.DataFrame(data=distances, columns=['dist'], index␣
↪= self.Y_train.index)

            # Sort distances, and only consider the k closest points in the␣
↪new dataframe df_knn
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]

            # Note that the index df_knn.index of df_knn contains indices in␣
↪Y_train of the k-closed training instances to
            # the i-th test instance. Thus, the dataframe self.
↪Y_train[df_knn.index] contains the classes of those k-closed
            # training instances. Method value_counts() computes the counts␣
↪(number of occurencies) for each class in
            # self.Y_train[df_knn.index] in dataframe predictions.
            predictions = self.Y_train[df_knn.index].value_counts()

            # the first element of the index predictions.index contains the␣
↪class with the highest count; i.e. the prediction y_pred_test.
            y_pred_test = predictions.index[0]

            # add the prediction y_pred_test to the prediction vector␣
↪Y_pred_test for all the test instances in X_test
            Y_pred_test.append(y_pred_test)

        return Y_pred_test
```

```
    def Minkowski_distance(self, x1, x2):
    # computes the Minkowski distance of x1 and x2 for two labeled instances␣
↪(x1,y1) and (x2,y2)

        # Set initial distance to 0
        distance = 0

        # Calculate Minkowski distance using the exponent exp
        for i in range(len(x1)):
            distance = distance + abs(x1[i] - x2[i])**self.exp

        distance = distance**(1/self.exp)

        return distance


    def normilize_maximum_absolute_scaling(self,df):
        # copy the dataframe
        df_scaled = df.copy()
        # apply maximum absolute scaling
        for column in df_scaled.columns:
                df_scaled[column] = df_scaled[column]  / df_scaled[column].
↪abs().max()

        return df_scaled

    def getClassProbs (self, X_test):

            # getting value type for Y
        Y_type_list =  Y_train.tolist()
        Y_type_no_dublicate = list(dict.fromkeys(Y_type_list))

        #creating new datafaram for prob
        df_probs = pd.DataFrame(index=Y_type_no_dublicate)

        Y_pred_test = [] #prediction vector Y_pred_test for all the test␣
↪instances in X_test is initialized to empty list []
        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances = []  #list of distances of the i-th test_instance for␣
↪all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances␣
↪in X_train
```

```
                    train_instance = self.X_train.iloc[j] #j-th training␣
↪instance
                    distance = self.Minkowski_distance(test_instance,␣
↪train_instance) #distance between i-th test instance and j-th training␣
↪instance
                    distances.append(distance) #add the distance to the list of␣
↪distances of the i-th test_instance

                df_dists = pd.DataFrame(data=distances, columns=['dist'], index␣
↪= self.Y_train.index)
#              print(df_dists)
                df_nn = df_dists.sort_values(by=['dist'], axis=0)
                df_knn =  df_nn[:self.k]
                # calculating probability of having sam one Y_train type
                predictions = self.Y_train[df_knn.index].value_counts()


                df_probs['test'+str(i)] = predictions/self.k


            print(df_probs)


        # Class of k-Nearest Neigbor Classifier
```

Orgiginal Data

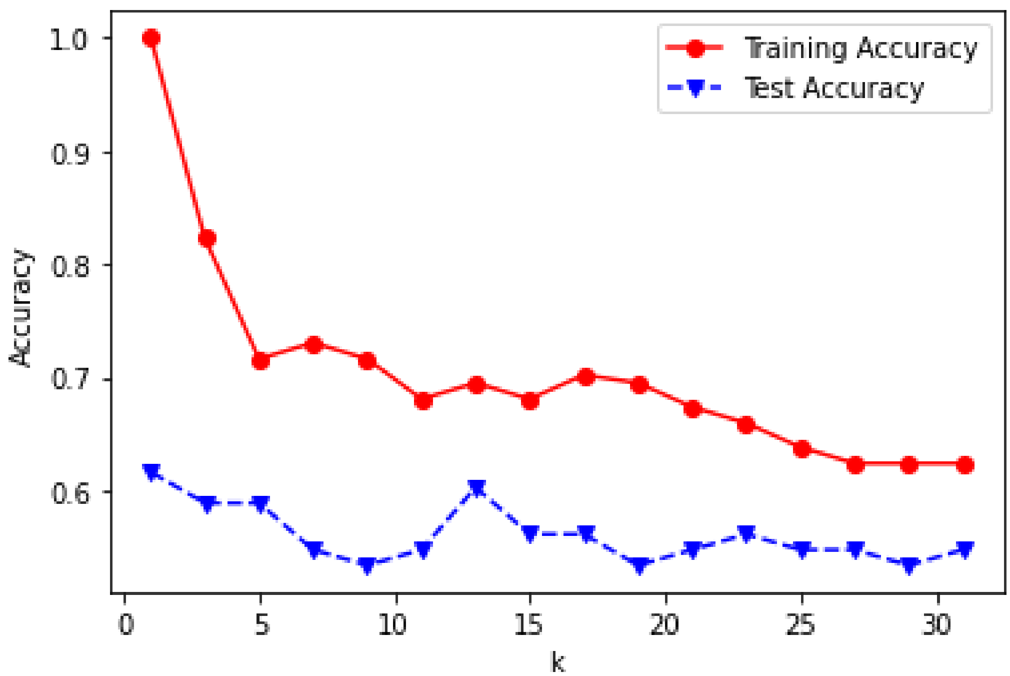
[16]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('glass.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)
```

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
↪random_state=10)

# range for the values of parameter k for kNN

k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]


trainAcc = np.zeros(len(k_range))
testAcc = np.zeros(len(k_range))


index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)

    index += 1


#########################################
# Plot of training and test accuracies
#########################################
print(X_train)
print(X_test)

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
```

```
          RI     Na    Mg    Al     Si     K     Ca    Ba    Fe
166  1.51747  12.84  3.50  1.14  73.27  0.56   8.55  0.00  0.00
21   1.52475  11.45  0.00  1.88  72.19  0.81  13.24  0.00  0.34
136  1.51754  13.48  3.74  1.17  72.99  0.59   8.03  0.00  0.00
206  1.51623  14.20  0.00  2.79  73.46  0.04   9.04  0.40  0.09
75   1.51652  13.56  3.57  1.47  72.45  0.64   7.96  0.00  0.00
..       ...    ...   ...   ...    ...   ...    ...   ...   ...
113  1.51651  14.38  0.00  1.94  73.61  0.00   8.48  1.57  0.00
64   1.52320  13.72  3.72  0.51  71.75  0.09  10.06  0.00  0.16
15   1.51707  13.48  3.48  1.71  72.52  0.62   7.99  0.00  0.00
125  1.51748  12.86  3.56  1.27  73.21  0.54   8.38  0.00  0.17
9    1.51789  13.19  3.90  1.30  72.33  0.55   8.44  0.00  0.28
```

```
[141 rows x 9 columns]
          RI     Na    Mg    Al     Si     K    Ca   Ba    Fe
161  1.52172  13.51  3.86  0.88  71.79  0.23  9.54  0.0  0.11
120  1.51660  12.99  3.18  1.23  72.97  0.58  8.81  0.0  0.24
105  1.51316  13.02  0.00  3.04  70.48  6.21  6.96  0.0  0.00
148  1.51574  14.86  3.67  1.74  71.87  0.16  7.36  0.0  0.12
69   1.52152  13.05  3.65  0.87  72.32  0.19  9.85  0.0  0.17
..       ...    ...   ...   ...    ...   ...   ...  ...   ...
165  1.52213  14.21  3.82  0.47  71.77  0.11  9.57  0.0  0.00
204  1.51860  13.36  3.43  1.43  72.26  0.51  8.60  0.0  0.00
72   1.51888  14.99  0.78  1.74  72.50  0.00  9.95  0.0  0.00
121  1.51589  12.88  3.43  1.40  73.28  0.69  8.05  0.0  0.24
43   1.51590  13.24  3.34  1.47  73.10  0.39  8.22  0.0  0.00

[73 rows x 9 columns]
```

[16]: Text(0, 0.5, 'Accuracy')



in here as you can see all of the values of each culumn ahave very vary rang.

Normalize data:

[113]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

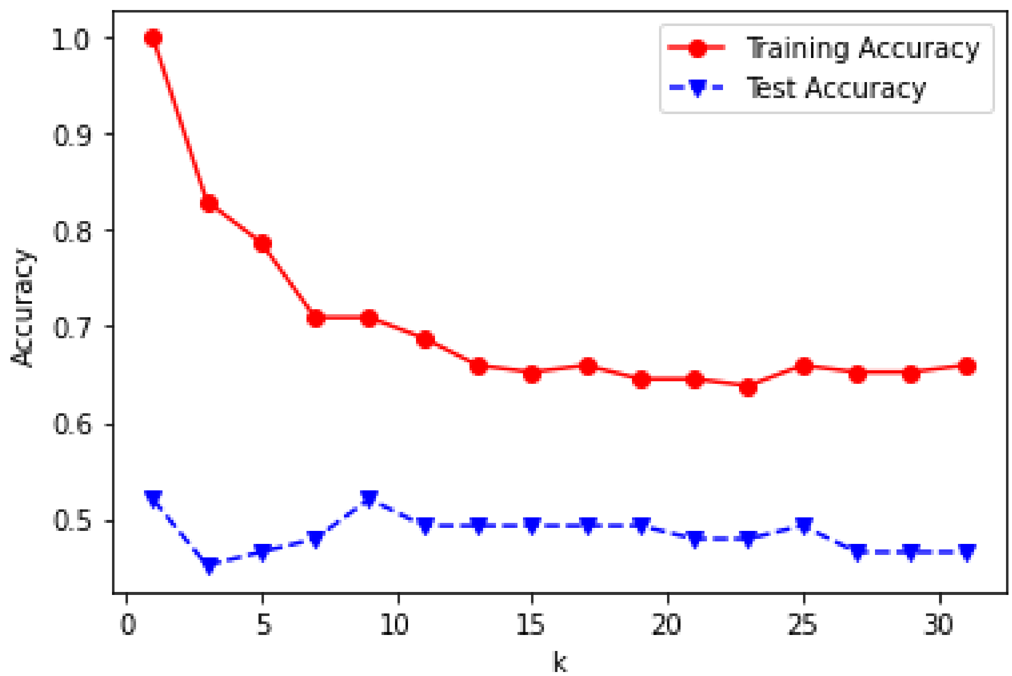
```
from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('glass.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
 ↪random_state=10)


# range for the values of parameter k for kNN

k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]

trainAcc = np.zeros(len(k_range))
testAcc = np.zeros(len(k_range))

index = 0
for k  in  k_range:
    clf = kNN(k)

    X_train = clf.normilize_maximum_absolute_scaling(X_train)
    X_test = clf.normilize_maximum_absolute_scaling(X_test)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1



# #########################################
# # Plot of training and test accuracies
# #########################################
# # trainAcc
print(X_train)
print(X_test)

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
```

```
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
```

```
           RI        Na        Mg        Al        Si         K        Ca  \
166  0.989269  0.738780  0.881612  0.377483  0.971622  0.090177  0.528104
21   0.994015  0.658803  0.000000  0.622517  0.957300  0.130435  0.817789
136  0.989315  0.775604  0.942065  0.387417  0.967909  0.095008  0.495985
206  0.988461  0.817031  0.000000  0.923841  0.974141  0.006441  0.558369
75   0.988650  0.780207  0.899244  0.486755  0.960748  0.103060  0.491662
..        ...       ...       ...       ...       ...       ...       ...
113  0.988644  0.827388  0.000000  0.642384  0.976130  0.000000  0.523780
64   0.993005  0.789413  0.937028  0.168874  0.951465  0.014493  0.621371
15   0.989009  0.775604  0.876574  0.566225  0.961676  0.099839  0.493515
125  0.989276  0.739931  0.896725  0.420530  0.970826  0.086957  0.517603
9    0.989543  0.758918  0.982368  0.430464  0.959157  0.088567  0.521309

           Ba        Fe
166  0.000000  0.000000
21   0.000000  0.918919
136  0.000000  0.000000
206  0.138889  0.243243
75   0.000000  0.000000
..        ...       ...
113  0.545139  0.000000
64   0.000000  0.432432
15   0.000000  0.000000
125  0.000000  0.459459
9    0.000000  0.756757

[141 rows x 9 columns]
           RI        Na        Mg        Al        Si         K        Ca  \
161  0.993776  0.901268  0.859688  0.251429  0.978199  0.037037  0.649864
120  0.990433  0.866578  0.708241  0.351429  0.994277  0.093398  0.600136
105  0.988186  0.868579  0.000000  0.868571  0.960349  1.000000  0.474114
148  0.989871  0.991328  0.817372  0.497143  0.979289  0.025765  0.501362
69   0.993646  0.870580  0.812918  0.248571  0.985420  0.030596  0.670981
..        ...       ...       ...       ...       ...       ...       ...
165  0.994044  0.947965  0.850780  0.134286  0.977926  0.017713  0.651907
204  0.991739  0.891261  0.763920  0.408571  0.984603  0.082126  0.585831
72   0.991922  1.000000  0.173719  0.497143  0.987873  0.000000  0.677793
121  0.989969  0.859239  0.763920  0.400000  0.998501  0.111111  0.548365
43   0.989976  0.883256  0.743875  0.420000  0.996049  0.062802  0.559946

      Ba        Fe
161  0.0  0.215686
120  0.0  0.470588
105  0.0  0.000000
```

```
148  0.0  0.235294
69   0.0  0.333333
..   ...       ...
165  0.0  0.000000
204  0.0  0.000000
72   0.0  0.000000
121  0.0  0.470588
43   0.0  0.000000

[73 rows x 9 columns]
```

[113]: Text(0, 0.5, 'Accuracy')



in here you can see after Normalization all of the numbers are between 0 and 1. I useed absolute mean value to calculate the rate for normalization data and as you can see in the graph the accuracy for both test and train data have been improved in compare to not normalize data.

2 changing exp for glass data

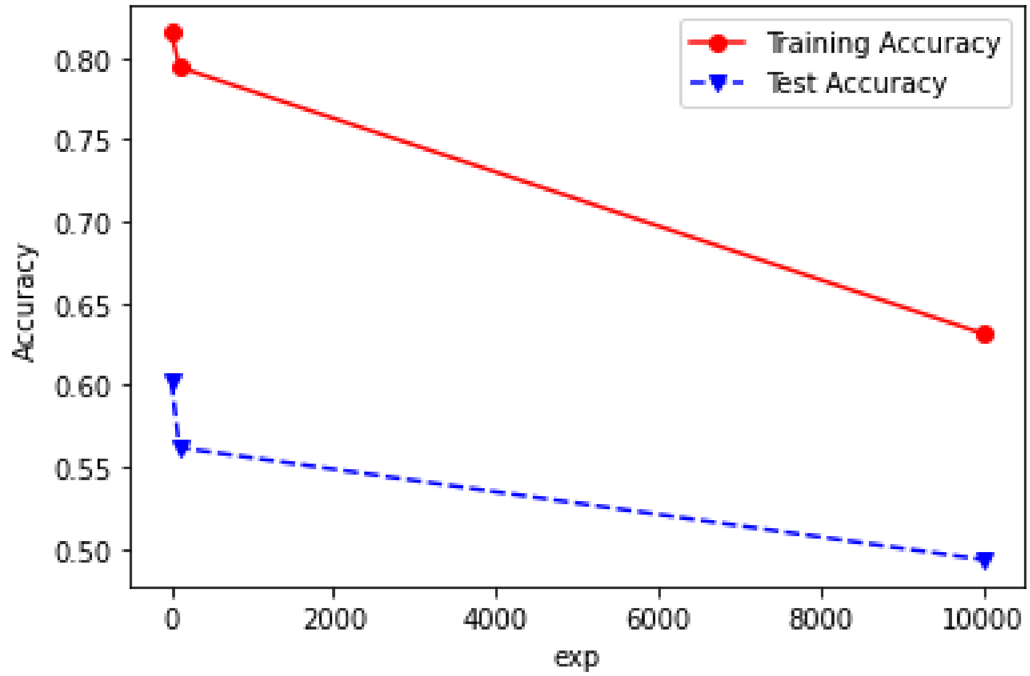
not normalize

[118]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import accuracy_score
from numpy.random import random
```

```
###################################################
# Hold-out testing: Training and Test set creation
###################################################

data = pd.read_csv('glass.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
 ↪random_state=10)


# range for the values of parameter exp for kNN

exp_range = [2,  100, 10000]

trainAcc = np.zeros(len(exp_range))
testAcc = np.zeros(len(exp_range))


index = 0
for exp  in  exp_range:
    clf = kNN(k = 3, exp = exp)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1


#########################################
# Plot of training and test accuracies
#########################################


plt.plot(exp_range,trainAcc,'ro-',exp_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('exp')
plt.ylabel('Accuracy')
```

```
<ipython-input-112-c08c10cd0486>:69: RuntimeWarning: overflow encountered in
double_scalars
  distance = distance + abs(x1[i] - x2[i])**self.exp
```

[118]: Text(0, 0.5, 'Accuracy')



in here as you can see all of the values of each culumn ahave very vary rang.

after normalization

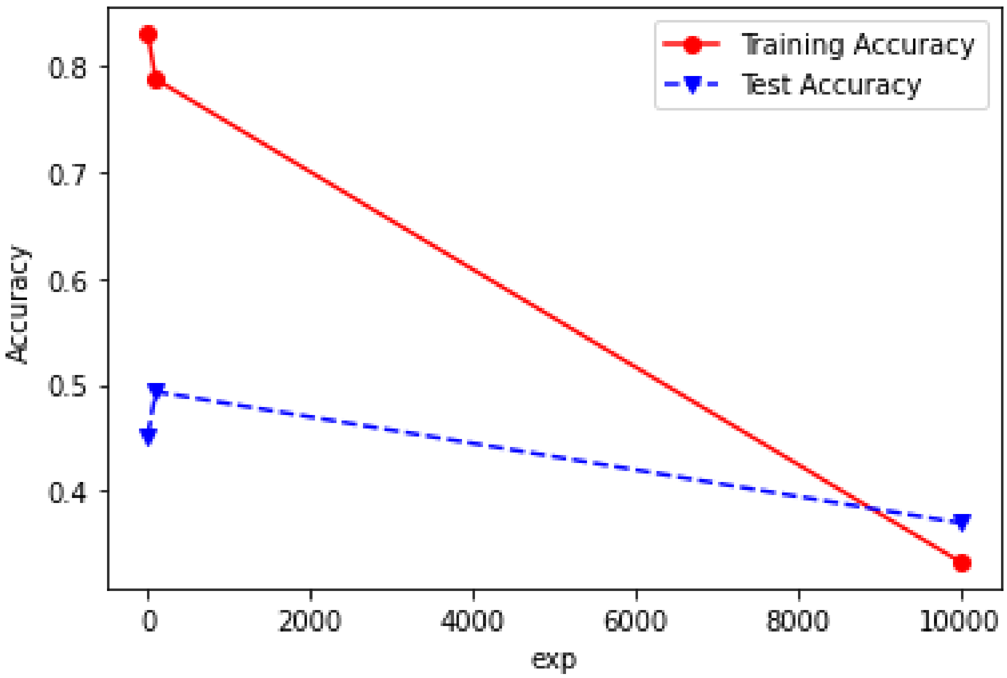
[123]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Normalize testing: Training and Test set creation
##################################################

data = pd.read_csv('glass.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
↪random_state=10)
```

```python
# range for the values of parameter exp for kNN

exp_range = [2,  100, 10000]

trainAcc = np.zeros(len(exp_range))
testAcc = np.zeros(len(exp_range))


index = 0
for exp  in  exp_range:

    clf = kNN(k = 3, exp = exp)
    X_train = clf.normilize_maximum_absolute_scaling(X_train)
    X_test = clf.normilize_maximum_absolute_scaling(X_test)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1


# #########################################
# # Plot of training and test accuracies
# #########################################
# # trainAcc
print(X_train)
print(X_test)

plt.plot(exp_range,trainAcc,'ro-',exp_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('exp')
plt.ylabel('Accuracy')
```

```
           RI        Na        Mg        Al        Si         K        Ca  \
166  0.989269  0.738780  0.881612  0.377483  0.971622  0.090177  0.528104
21   0.994015  0.658803  0.000000  0.622517  0.957300  0.130435  0.817789
136  0.989315  0.775604  0.942065  0.387417  0.967909  0.095008  0.495985
206  0.988461  0.817031  0.000000  0.923841  0.974141  0.006441  0.558369
75   0.988650  0.780207  0.899244  0.486755  0.960748  0.103060  0.491662
..        ...       ...       ...       ...       ...       ...       ...
113  0.988644  0.827388  0.000000  0.642384  0.976130  0.000000  0.523780
64   0.993005  0.789413  0.937028  0.168874  0.951465  0.014493  0.621371
15   0.989009  0.775604  0.876574  0.566225  0.961676  0.099839  0.493515
125  0.989276  0.739931  0.896725  0.420530  0.970826  0.086957  0.517603
9    0.989543  0.758918  0.982368  0.430464  0.959157  0.088567  0.521309
```

```
           Ba        Fe
166  0.000000  0.000000
21   0.000000  0.918919
136  0.000000  0.000000
206  0.138889  0.243243
75   0.000000  0.000000
..        ...       ...
113  0.545139  0.000000
64   0.000000  0.432432
15   0.000000  0.000000
125  0.000000  0.459459
9    0.000000  0.756757

[141 rows x 9 columns]
           RI        Na        Mg        Al        Si         K        Ca  \
161  0.993776  0.901268  0.859688  0.251429  0.978199  0.037037  0.649864
120  0.990433  0.866578  0.708241  0.351429  0.994277  0.093398  0.600136
105  0.988186  0.868579  0.000000  0.868571  0.960349  1.000000  0.474114
148  0.989871  0.991328  0.817372  0.497143  0.979289  0.025765  0.501362
69   0.993646  0.870580  0.812918  0.248571  0.985420  0.030596  0.670981
..        ...       ...       ...       ...       ...       ...       ...
165  0.994044  0.947965  0.850780  0.134286  0.977926  0.017713  0.651907
204  0.991739  0.891261  0.763920  0.408571  0.984603  0.082126  0.585831
72   0.991922  1.000000  0.173719  0.497143  0.987873  0.000000  0.677793
121  0.989969  0.859239  0.763920  0.400000  0.998501  0.111111  0.548365
43   0.989976  0.883256  0.743875  0.420000  0.996049  0.062802  0.559946

      Ba        Fe
161  0.0  0.215686
120  0.0  0.470588
105  0.0  0.000000
148  0.0  0.235294
69   0.0  0.333333
..   ...       ...
165  0.0  0.000000
204  0.0  0.000000
72   0.0  0.000000
121  0.0  0.470588
43   0.0  0.000000

[73 rows x 9 columns]
```

[123]:
```
Text(0, 0.5, 'Accuracy')
```



in here you can see after Normalization all of the numbers are between 0 and 1. I useed absolute mean value to calculate the rate for normalization data and as you can see in the graph the accuracy for both test and train data have been improved in compare to not normalize data.

3 Prediction

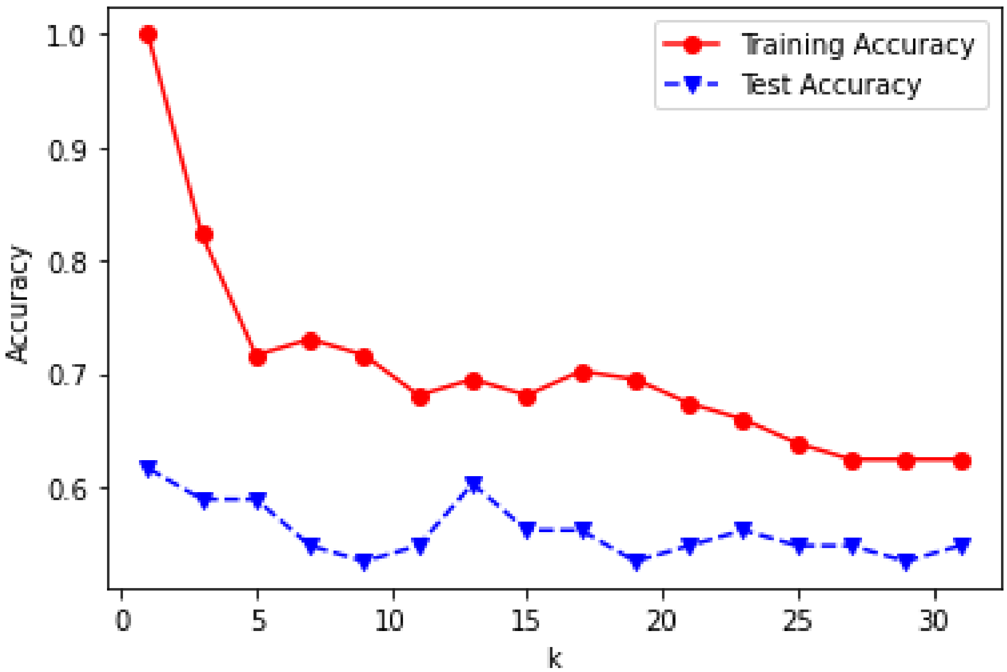
[14]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('glass.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)
```

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
 ↪random_state=10)

# range for the values of parameter k for kNN

k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]


trainAcc = np.zeros(len(k_range))
testAcc = np.zeros(len(k_range))


index = 0
for k  in  k_range:
    clf = kNN(k)
#     X_train = clf.normilize_maximum_absolute_scaling(X_train)
#     X_test = clf.normilize_maximum_absolute_scaling(X_test)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    clf.getClassProbs(X_test)

    index += 1


#########################################
# Plot of training and test accuracies
#########################################
print(X_train)
print(X_test)

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
```

```
                        test0  test1  test2  test3  test4  test5  test6  \
'build wind float'        1.0    1.0    NaN    1.0    1.0    NaN    1.0
'build wind non-float'    NaN    NaN    NaN    NaN    NaN    NaN    NaN
headlamps                 NaN    NaN    NaN    NaN    NaN    1.0    NaN
'vehic wind float'        NaN    NaN    NaN    NaN    NaN    NaN    NaN
containers                NaN    NaN    1.0    NaN    NaN    NaN    NaN
tableware                 NaN    NaN    NaN    NaN    NaN    NaN    NaN

                        test7  test8  test9  ...  test63  test64  test65  \
```

```
'build wind non-float'     3.0    20.0     6.0    16.0     5.0    15.0    15.0
headlamps                 20.0     NaN     1.0     1.0    18.0     NaN     NaN
'vehic wind float'         NaN     4.0     6.0     6.0     NaN     NaN     3.0
containers                 3.0     NaN     NaN     NaN     4.0     NaN     NaN
tableware                  4.0     NaN     1.0     NaN     3.0     NaN     NaN

[6 rows x 73 columns]
          RI     Na    Mg    Al     Si     K     Ca    Ba    Fe
166  1.51747  12.84  3.50  1.14  73.27  0.56   8.55  0.00  0.00
21   1.52475  11.45  0.00  1.88  72.19  0.81  13.24  0.00  0.34
136  1.51754  13.48  3.74  1.17  72.99  0.59   8.03  0.00  0.00
206  1.51623  14.20  0.00  2.79  73.46  0.04   9.04  0.40  0.09
75   1.51652  13.56  3.57  1.47  72.45  0.64   7.96  0.00  0.00
..       ...    ...   ...   ...    ...   ...    ...   ...   ...
113  1.51651  14.38  0.00  1.94  73.61  0.00   8.48  1.57  0.00
64   1.52320  13.72  3.72  0.51  71.75  0.09  10.06  0.00  0.16
15   1.51707  13.48  3.48  1.71  72.52  0.62   7.99  0.00  0.00
125  1.51748  12.86  3.56  1.27  73.21  0.54   8.38  0.00  0.17
9    1.51789  13.19  3.90  1.30  72.33  0.55   8.44  0.00  0.28

[141 rows x 9 columns]
          RI     Na    Mg    Al     Si     K    Ca   Ba    Fe
161  1.52172  13.51  3.86  0.88  71.79  0.23  9.54  0.0  0.11
120  1.51660  12.99  3.18  1.23  72.97  0.58  8.81  0.0  0.24
105  1.51316  13.02  0.00  3.04  70.48  6.21  6.96  0.0  0.00
148  1.51574  14.86  3.67  1.74  71.87  0.16  7.36  0.0  0.12
69   1.52152  13.05  3.65  0.87  72.32  0.19  9.85  0.0  0.17
..       ...    ...   ...   ...    ...   ...   ...  ...   ...
165  1.52213  14.21  3.82  0.47  71.77  0.11  9.57  0.0  0.00
204  1.51860  13.36  3.43  1.43  72.26  0.51  8.60  0.0  0.00
72   1.51888  14.99  0.78  1.74  72.50  0.00  9.95  0.0  0.00
121  1.51589  12.88  3.43  1.40  73.28  0.69  8.05  0.0  0.24
43   1.51590  13.24  3.34  1.47  73.10  0.39  8.22  0.0  0.00

[73 rows x 9 columns]
```

[14]: Text(0, 0.5, 'Accuracy')



in here I caclute the probablity of happening of each Y for each test cases . which are showned in the above table. the data that has been used are normalized

LAB-Diabites-02

November 16, 2021

1 Data For Diabites (Normalization & prediction)

[4]:
```
# Class of k-Nearest Neigbor Classifier



class kNN():
    maxTrain = []

    def __init__(self, k = 3, exp = 2):
    # constructor for kNN classifier
    # k is the number of neighbor for local class estimation
    # exp is the exponent for the Minkowski distance
        self.k = k
        self.exp = exp

    def fit(self, X_train, Y_train):
    # training k-NN method
    # X_train is the training data given with input attributes. n-th row␣
 ↪correponds to n-th instance.
    # Y_train is the output data (output vector): n-th element of Y_train is the␣
 ↪output value for n-th instance in X_train.
        self.X_train = X_train
        self.Y_train = Y_train

    def getDiscreteClassification(self, X_test):
    # predict-class k-NN method
    # X_test is the test data given with input attributes. Rows correpond to␣
 ↪instances
    # Method outputs prediction vector Y_pred_test:  n-th element of Y_pred_test␣
 ↪is the prediction for n-th instance in X_test

        Y_pred_test = [] #prediction vector Y_pred_test for all the test␣
 ↪instances in X_test is initialized to empty list []


        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance
```

```
            distances = []  #list of distances of the i-th test_instance for all␣
↪the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances in␣
↪X_train
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance,␣
↪train_instance) #distance between i-th test instance and j-th training␣
↪instance
                distances.append(distance) #add the distance to the list of␣
↪distances of the i-th test_instance

            # Store distances in a dataframe. The dataframe has the index of␣
↪Y_train in order to keep the correspondence with the classes of the training␣
↪instances
            df_dists = pd.DataFrame(data=distances, columns=['dist'], index =␣
↪self.Y_train.index)

            # Sort distances, and only consider the k closest points in the new␣
↪dataframe df_knn
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]

            # Note that the index df_knn.index of df_knn contains indices in␣
↪Y_train of the k-closed training instances to
            # the i-th test instance. Thus, the dataframe self.Y_train[df_knn.
↪index] contains the classes of those k-closed
            # training instances. Method value_counts() computes the counts␣
↪(number of occurencies) for each class in
            # self.Y_train[df_knn.index] in dataframe predictions.
            predictions = self.Y_train[df_knn.index].value_counts()

            # the first element of the index predictions.index contains the␣
↪class with the highest count; i.e. the prediction y_pred_test.
            y_pred_test = predictions.index[0]

            # add the prediction y_pred_test to the prediction vector␣
↪Y_pred_test for all the test instances in X_test
            Y_pred_test.append(y_pred_test)

        return Y_pred_test


    def Minkowski_distance(self, x1, x2):
```

```
    # computes the Minkowski distance of x1 and x2 for two labeled instances␣
↪(x1,y1) and (x2,y2)

        # Set initial distance to 0
        distance = 0

        # Calculate Minkowski distance using the exponent exp
        for i in range(len(x1)):
            distance = distance + abs(x1[i] - x2[i])**self.exp

        distance = distance**(1/self.exp)

        return distance

    def normilize_maximum_absolute_scaling(self,df):
        # copy the dataframe
        df_scaled = df.copy()
        # apply maximum absolute scaling
        for column in df_scaled.columns:
            df_scaled[column] = df_scaled[column]  / df_scaled[column].abs().
↪max()

        return df_scaled
```

Original Data

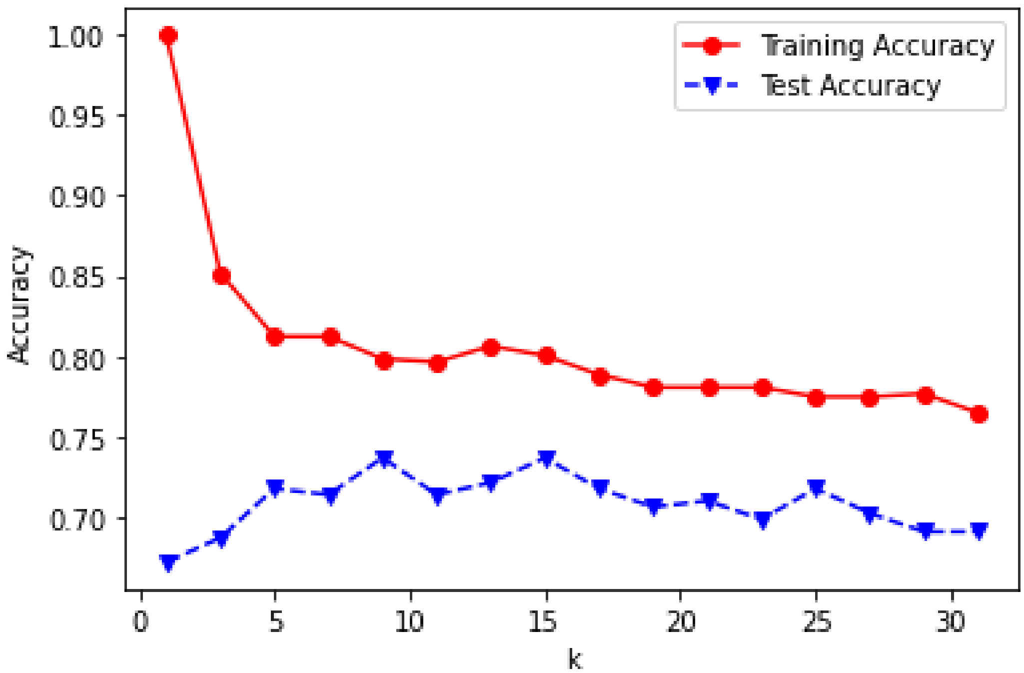
[5]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('diabetes.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
↪random_state=10)

# range for the values of parameter k for kNN
```

```
k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]

trainAcc = np.zeros(len(k_range))
testAcc = np.zeros(len(k_range))


index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1


#########################################
# Plot of training and test accuracies
#########################################
print(X_train)
print(X_test)

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
```

```
     preg  plas  pres  skin  insu  mass   pedi  age
659     3    80    82    31    70  34.2  1.292   27
439     6   107    88     0     0  36.8  0.727   31
72     13   126    90     0     0  43.4  0.583   42
329     6   105    70    32    68  30.8  0.122   37
692     2   121    70    32    95  39.1  0.886   23
..    ...   ...   ...   ...   ...   ...    ...  ...
369     1   133   102    28   140  32.8  0.234   45
320     4   129    60    12   231  27.5  0.527   31
527     3   116    74    15   105  26.3  0.107   24
125     1    88    30    42    99  55.0  0.496   26
265     5    96    74    18    67  33.6  0.997   43

[506 rows x 8 columns]
     preg  plas  pres  skin  insu  mass   pedi  age
568     4   154    72    29   126  31.3  0.338   37
620     2   112    86    42   160  38.4  0.246   28
456     1   135    54     0     0  26.7  0.687   62
197     3   107    62    13    48  22.9  0.678   23
```

```
714     3  102   74    0    0  29.5  0.121   32
..    ...  ...  ...  ...  ...   ...    ...  ...
581     6  109   60   27    0  25.0  0.206   27
300     0  167    0    0    0  32.3  0.839   30
110     3  171   72   33  135  33.3  0.199   24
450     1   82   64   13   95  21.2  0.415   23
21      8   99   84    0    0  35.4  0.388   50

[262 rows x 8 columns]
```

[5]: Text(0, 0.5, 'Accuracy')



in here as you can see all of the values of each culumn have very vary rang.

Normilize Data:

[4]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Hold-out testing: Training and Test set creation
```

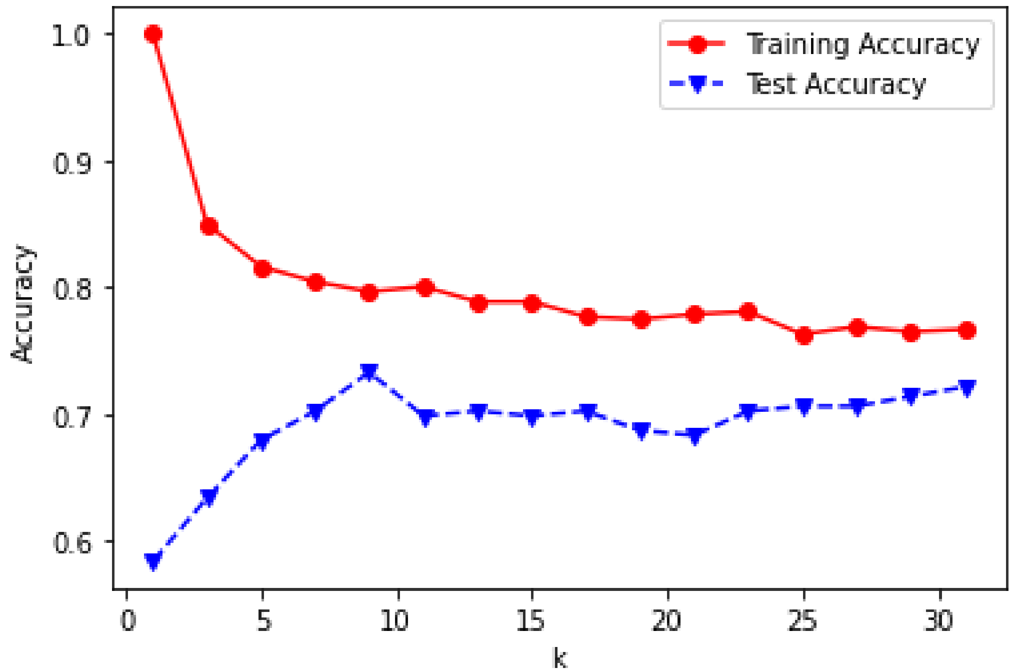
```
##################################################

data = pd.read_csv('diabetes.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
 ↪random_state=10)


# range for the values of parameter k for kNN

k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]

trainAcc = np.zeros(len(k_range))
testAcc = np.zeros(len(k_range))

index = 0
for k  in  k_range:
    clf = kNN(k)

    X_train = clf.normilize_maximum_absolute_scaling(X_train)
    X_test = clf.normilize_maximum_absolute_scaling(X_test)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1



# #########################################
# # Plot of training and test accuracies
# #########################################
# # trainAcc
print(X_train)
print(X_test)

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
```

```
         preg      plas      pres      skin      insu      mass      pedi  \
659  0.176471  0.402010  0.672131  0.313131  0.082742  0.509687  0.533884
439  0.352941  0.537688  0.721311  0.000000  0.000000  0.548435  0.300413
```

```
72   0.764706  0.633166  0.737705  0.000000  0.000000  0.646796  0.240909
329  0.352941  0.527638  0.573770  0.323232  0.080378  0.459016  0.050413
692  0.117647  0.608040  0.573770  0.323232  0.112293  0.582712  0.366116
..        ...       ...       ...       ...       ...       ...       ...
369  0.058824  0.668342  0.836066  0.282828  0.165485  0.488823  0.096694
320  0.235294  0.648241  0.491803  0.121212  0.273050  0.409836  0.217769
527  0.176471  0.582915  0.606557  0.151515  0.124113  0.391952  0.044215
125  0.058824  0.442211  0.245902  0.424242  0.117021  0.819672  0.204959
265  0.294118  0.482412  0.606557  0.181818  0.079196  0.500745  0.411983

          age
659  0.333333
439  0.382716
72   0.518519
329  0.456790
692  0.283951
..        ...
369  0.555556
320  0.382716
527  0.296296
125  0.320988
265  0.530864

[506 rows x 8 columns]
         preg      plas      pres      skin      insu      mass      pedi  \
568  0.285714  0.781726  0.654545  0.483333  0.259794  0.588346  0.178553
620  0.142857  0.568528  0.781818  0.700000  0.329897  0.721805  0.129952
456  0.071429  0.685279  0.490909  0.000000  0.000000  0.501880  0.362916
197  0.214286  0.543147  0.563636  0.216667  0.098969  0.430451  0.358162
714  0.214286  0.517766  0.672727  0.000000  0.000000  0.554511  0.063920
..        ...       ...       ...       ...       ...       ...       ...
581  0.428571  0.553299  0.545455  0.450000  0.000000  0.469925  0.108822
300  0.000000  0.847716  0.000000  0.000000  0.000000  0.607143  0.443212
110  0.214286  0.868020  0.654545  0.550000  0.278351  0.625940  0.105124
450  0.071429  0.416244  0.581818  0.216667  0.195876  0.398496  0.219229
21   0.571429  0.502538  0.763636  0.000000  0.000000  0.665414  0.204966

          age
568  0.536232
620  0.405797
456  0.898551
197  0.333333
714  0.463768
..        ...
581  0.391304
300  0.434783
110  0.347826
450  0.333333
```

```
21    0.724638

[262 rows x 8 columns]
```

[4]: Text(0, 0.5, 'Accuracy')



in here you can see after Normalization all of the numbers are between 0 and 1. I useed absolute mean value to calculate the rate for normalization data and as you can see in the graph the accuracy for both test and train data have been improved in compare to not normalize data.

2 changing exp for Diabites data

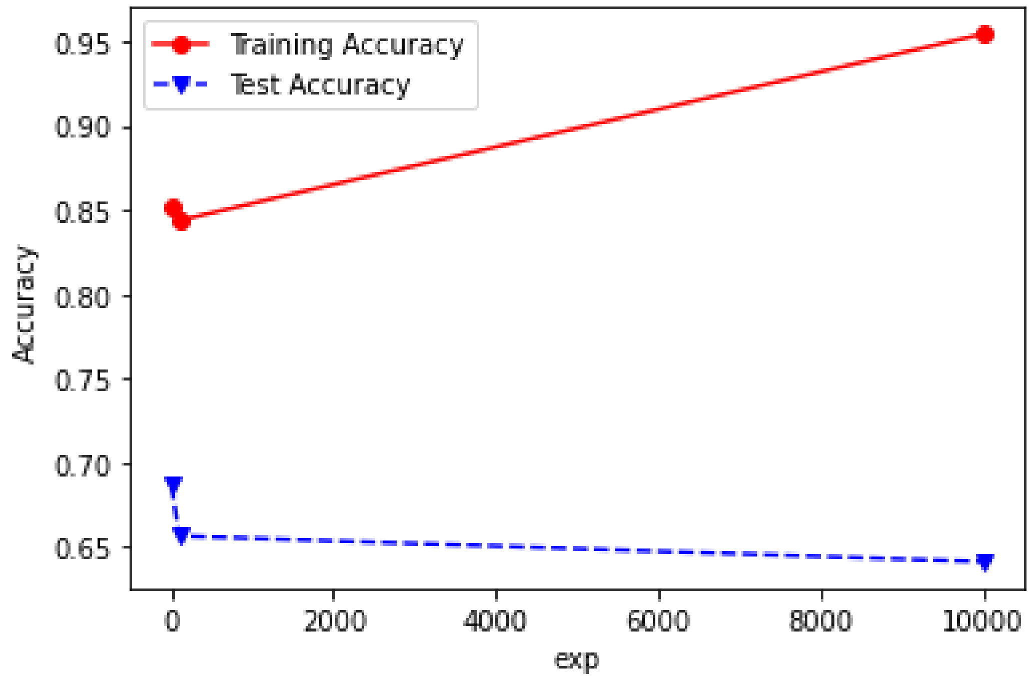
Not Normalize

[5]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import accuracy_score
from numpy.random import random

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('diabetes.csv')
data.head()
```

```
Y = data['class']
X = data.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
 ↪random_state=10)


# range for the values of parameter exp for kNN

exp_range = [2,  100, 10000]

trainAcc = np.zeros(len(exp_range))
testAcc = np.zeros(len(exp_range))


index = 0
for exp  in  exp_range:
    clf = kNN(k = 3, exp = exp)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1


#########################################
# Plot of training and test accuracies
#########################################


plt.plot(exp_range,trainAcc,'ro-',exp_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('exp')
plt.ylabel('Accuracy')
```

```
<ipython-input-1-c08c10cd0486>:69: RuntimeWarning: overflow encountered in
double_scalars
  distance = distance + abs(x1[i] - x2[i])**self.exp
```

[5]:
```
Text(0, 0.5, 'Accuracy')
```



in here as you can see all of the values of each culumn ahave very vary rang.

Normilize

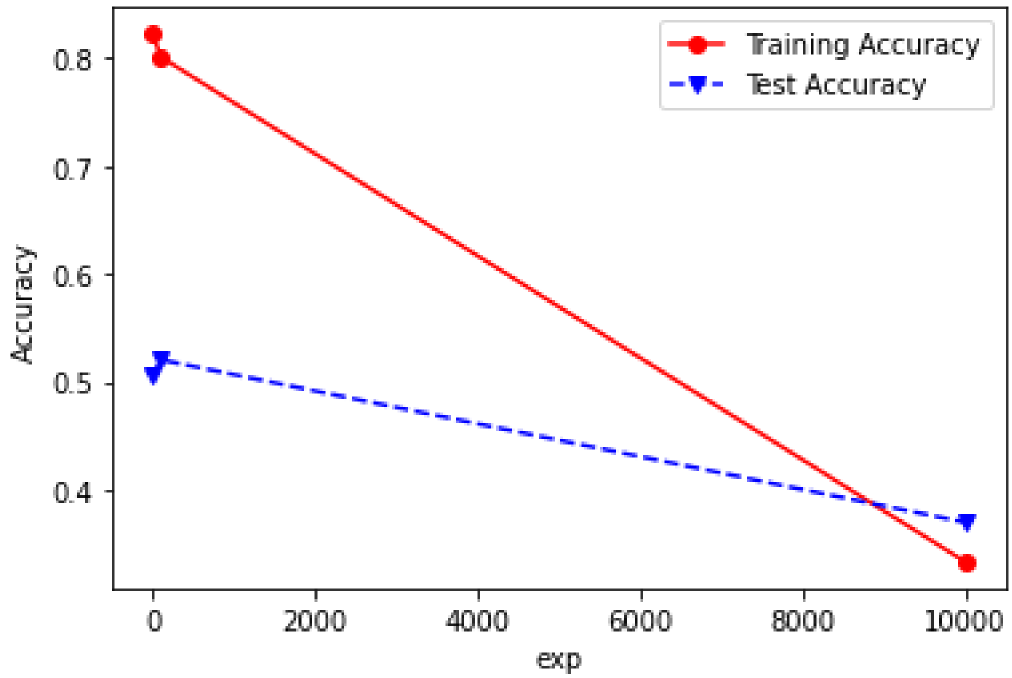
[7]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Normalize testing: Training and Test set creation
##################################################

data = pd.read_csv('glass.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
↪random_state=10)
```

```
# range for the values of parameter exp for kNN

exp_range = [2,  100, 10000]

trainAcc = np.zeros(len(exp_range))
testAcc = np.zeros(len(exp_range))


index = 0
for exp  in  exp_range:

    clf = kNN(k = 3, exp = exp)
    X_train = clf.normilize_maximum_absolute_scaling(X_train)
    X_test = clf.normilize_maximum_absolute_scaling(X_test)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1


# ##########################################
# # Plot of training and test accuracies
# ##########################################
# # trainAcc
print(X_train)
print(X_test)

plt.plot(exp_range,trainAcc,'ro-',exp_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('exp')
plt.ylabel('Accuracy')
```

```
           RI        Na        Mg        Al        Si         K        Ca  \
166  0.989269  0.738780  0.881612  0.377483  0.971622  0.090177  0.528104
21   0.994015  0.658803  0.000000  0.622517  0.957300  0.130435  0.817789
136  0.989315  0.775604  0.942065  0.387417  0.967909  0.095008  0.495985
206  0.988461  0.817031  0.000000  0.923841  0.974141  0.006441  0.558369
75   0.988650  0.780207  0.899244  0.486755  0.960748  0.103060  0.491662
..        ...       ...       ...       ...       ...       ...       ...
113  0.988644  0.827388  0.000000  0.642384  0.976130  0.000000  0.523780
64   0.993005  0.789413  0.937028  0.168874  0.951465  0.014493  0.621371
15   0.989009  0.775604  0.876574  0.566225  0.961676  0.099839  0.493515
125  0.989276  0.739931  0.896725  0.420530  0.970826  0.086957  0.517603
9    0.989543  0.758918  0.982368  0.430464  0.959157  0.088567  0.521309

           Ba        Fe
```

```
166  0.000000  0.000000
21   0.000000  0.918919
136  0.000000  0.000000
206  0.138889  0.243243
75   0.000000  0.000000
..        ...       ...
113  0.545139  0.000000
64   0.000000  0.432432
15   0.000000  0.000000
125  0.000000  0.459459
9    0.000000  0.756757

[141 rows x 9 columns]
           RI        Na        Mg        Al        Si         K        Ca  \
161  0.993776  0.901268  0.859688  0.251429  0.978199  0.037037  0.649864
120  0.990433  0.866578  0.708241  0.351429  0.994277  0.093398  0.600136
105  0.988186  0.868579  0.000000  0.868571  0.960349  1.000000  0.474114
148  0.989871  0.991328  0.817372  0.497143  0.979289  0.025765  0.501362
69   0.993646  0.870580  0.812918  0.248571  0.985420  0.030596  0.670981
..        ...       ...       ...       ...       ...       ...       ...
165  0.994044  0.947965  0.850780  0.134286  0.977926  0.017713  0.651907
204  0.991739  0.891261  0.763920  0.408571  0.984603  0.082126  0.585831
72   0.991922  1.000000  0.173719  0.497143  0.987873  0.000000  0.677793
121  0.989969  0.859239  0.763920  0.400000  0.998501  0.111111  0.548365
43   0.989976  0.883256  0.743875  0.420000  0.996049  0.062802  0.559946

      Ba        Fe
161  0.0  0.215686
120  0.0  0.470588
105  0.0  0.000000
148  0.0  0.235294
69   0.0  0.333333
..   ...       ...
165  0.0  0.000000
204  0.0  0.000000
72   0.0  0.000000
121  0.0  0.470588
43   0.0  0.000000

[73 rows x 9 columns]
```

[7]: Text(0, 0.5, 'Accuracy')



in here you can see after Normalization all of the numbers are between 0 and 1. I useed absolute mean value to calculate the rate for normalization data and as you can see in the graph the accuracy for both test and train data have been improved in compare to not normalize data.

Lab-autoprice-02

November 16, 2021

1 Data for autoprice (Normalization & mean)

[15]:
```
# Class of k-Nearest Neigbor Classifier



class kNN():
    maxTrain = []

    def __init__(self, k = 3, exp = 2):
    # constructor for kNN classifier
    # k is the number of neighbor for local class estimation
    # exp is the exponent for the Minkowski distance
        self.k = k
        self.exp = exp

    def fit(self, X_train, Y_train):
    # training k-NN method
    # X_train is the training data given with input attributes. n-th row␣
 ↪correponds to n-th instance.
    # Y_train is the output data (output vector): n-th element of Y_train is the␣
 ↪output value for n-th instance in X_train.
        self.X_train = X_train
        self.Y_train = Y_train

    def getDiscreteClassification(self, X_test):
    # predict-class k-NN method
    # X_test is the test data given with input attributes. Rows correpond to␣
 ↪instances
    # Method outputs prediction vector Y_pred_test:  n-th element of Y_pred_test␣
 ↪is the prediction for n-th instance in X_test

        Y_pred_test = [] #prediction vector Y_pred_test for all the test␣
 ↪instances in X_test is initialized to empty list []
```

```
        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances = []  #list of distances of the i-th test_instance for all␣
↪the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances in␣
↪X_train
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance,␣
↪train_instance) #distance between i-th test instance and j-th training␣
↪instance
                distances.append(distance) #add the distance to the list of␣
↪distances of the i-th test_instance

            # Store distances in a dataframe. The dataframe has the index of␣
↪Y_train in order to keep the correspondence with the classes of the training␣
↪instances
            df_dists = pd.DataFrame(data=distances, columns=['dist'], index =␣
↪self.Y_train.index)

            # Sort distances, and only consider the k closest points in the new␣
↪dataframe df_knn
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]

            # Note that the index df_knn.index of df_knn contains indices in␣
↪Y_train of the k-closed training instances to
            # the i-th test instance. Thus, the dataframe self.Y_train[df_knn.
↪index] contains the classes of those k-closed
            # training instances. Method value_counts() computes the counts␣
↪(number of occurencies) for each class in
            # self.Y_train[df_knn.index] in dataframe predictions.
            predictions = self.Y_train[df_knn.index].value_counts()

            # the first element of the index predictions.index contains the␣
↪class with the highest count; i.e. the prediction y_pred_test.
            y_pred_test = predictions.index[0]

            # add the prediction y_pred_test to the prediction vector␣
↪Y_pred_test for all the test instances in X_test
            Y_pred_test.append(y_pred_test)

        return Y_pred_test
```

```
    def Minkowski_distance(self, x1, x2):
    # computes the Minkowski distance of x1 and x2 for two labeled instances␣
↪(x1,y1) and (x2,y2)

        # Set initial distance to 0
        distance = 0

        # Calculate Minkowski distance using the exponent exp
        for i in range(len(x1)):
            distance = distance + abs(x1[i] - x2[i])**self.exp

        distance = distance**(1/self.exp)

        return distance


    def normilize_maximum_absolute_scaling(self,df):
        # copy the dataframe
        df_scaled = df.copy()
        # apply maximum absolute scaling
        for column in df_scaled.columns:
            df_scaled[column] = df_scaled[column]  / df_scaled[column].abs().
↪max()

        return df_scaled



    def getPrediction (self, X_test):

            # getting value type for Y
        Y_type_list =  Y_train.tolist()
        Y_type_no_dublicate = list(dict.fromkeys(Y_type_list))

        #creating new datafaram for Mean
        df_mean = pd.DataFrame(index=Y_type_no_dublicate)

        Y_pred_test = [] #prediction vector Y_pred_test for all the test␣
↪instances in X_test is initialized to empty list []
        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances = []  #list of distances of the i-th test_instance for all␣
↪the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances in␣
↪X_train
```

```
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance,␣
↪train_instance) #distance between i-th test instance and j-th training␣
↪instance
                distances.append(distance) #add the distance to the list of␣
↪distances of the i-th test_instance

            df_dists = pd.DataFrame(data=distances, columns=['dist'], index =␣
↪self.Y_train.index)
#             print(df_dists)
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]
            # claculting mean for each test data
            mean = self.Y_train[df_knn.index].mean()


            df_mean['test'+str(i)] = mean


        print(df_mean)
```

Orgiginal Data

[18]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from numpy.random import random
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_absolute_error

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('autoprice.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)
data.head()
```

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34,␣
 ↪random_state=10)

# range for the values of parameter k for kNN

k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]

trainMain = np.zeros(len(k_range))
testMain = np.zeros(len(k_range))


index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainMain[index] = mean_absolute_error(Y_train, Y_predTrain)
    testMain[index] = mean_absolute_error(Y_test, Y_predTest)
    teast = clf.getPrediction(X_test)

    index += 1


#########################################
# Plot of training and test accuracies
#########################################

plt.plot(k_range,trainMain,'ro-',k_range,testMain,'bv--')
plt.legend(['Train Mean Abolute Error','Test Mean Abolute Error'])
plt.xlabel('k')
plt.ylabel('Mean Abolute Error')
```

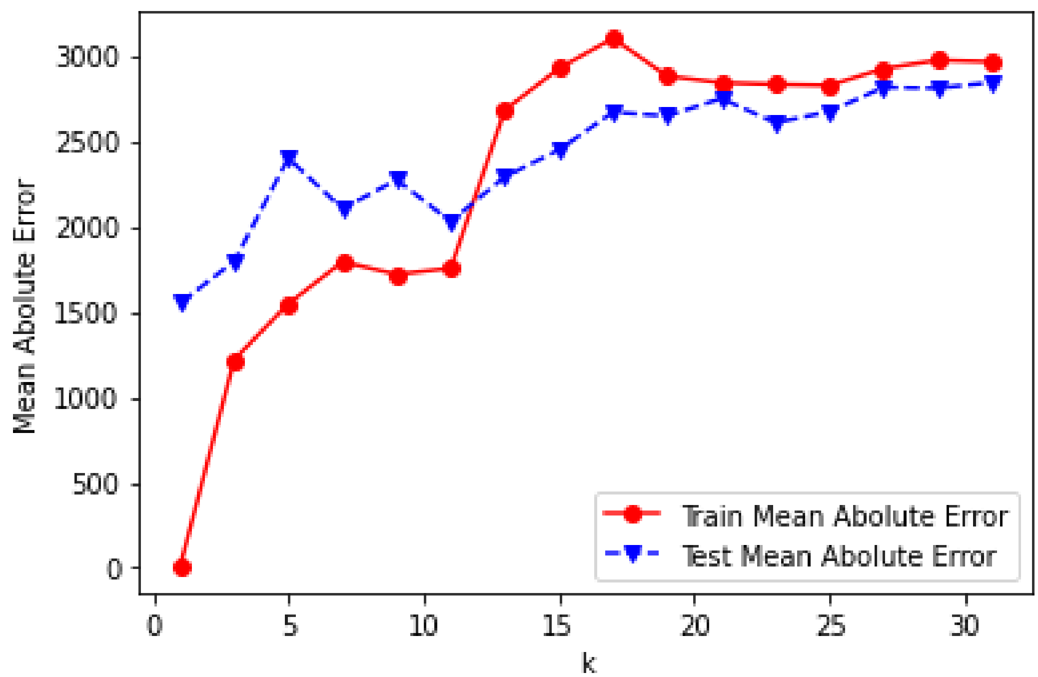
```
       test0   test1    test2   test3    test4    test5   test6   test7  \
13200  6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
35056  6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
7463   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
7295   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
9959   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
...       ...     ...      ...     ...      ...      ...     ...     ...
8238   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
7299   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
6692   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
9538   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0
6295   6095.0  7957.0  28248.0  7295.0  11694.0  18620.0  8449.0  6295.0

       test8   test9  ...  test45   test46   test47  test48   test49  test50  \
13200  9279.0  8449.0  ...  7957.0  11694.0  16900.0  9095.0  15580.0  7299.0
```

13200	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
35056	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
7463	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
7295	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
9959	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
...	...	...	...	...	...
8238	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
7299	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
6692	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
9538	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452
6295	8747.322581	11226.451613	15729.580645	8731.516129	15077.806452

	test50	test51	test52	test53	test54
13200	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
35056	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
7463	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
7295	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
9959	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
...	...	...	...	...	...
8238	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
7299	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
6692	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
9538	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871
6295	7004.967742	8497.258065	15053.967742	18067.741935	6872.483871

[98 rows x 55 columns]

```
[18]: Text(0, 0.5, 'Mean Abolute Error')
```



the above grash shoen the absolute error value of Y test and Y train data. and for the tables you can see the mean of each value for each Y for all of the test cases