Lab-02

November 16, 2021

[1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from statistics import mean
from numpy.random import random
from sklearn.metrics import mean_absolute_error
```

1 Lena Feiler - i6246119: Machine Learning Assignment 2

[17]:
```
# Class of k-Nearest Neigbor Classifier
class kNN():
    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

    # Exercise B
    # not allowed to use anything from X_test
    def normalize(self, X_test, X_train):
        # if data has negative values need to take lower bound into account
        train_copy = X_train.copy()
        test_copy = X_test.copy()
        max_value = X_train.max()
        min_value = X_train.min()
        # Normalization: (value-min)/ (max-min)
        for col in X_train.columns:
            # extract the columns
            rows = X_train.loc[:, col]
            new_rows = (rows - min_value[col]).divide(max_value[col] -␣
↪min_value[col])
            train_copy.loc[:, col] = new_rows

        for col in X_test.columns:
            # extract the columns
            # clip values to 1 or 0
            rows = X_test.loc[:, col]
            new_rows = (rows - min_value[col]).divide(max_value[col] -␣
↪min_value[col])
            new_rows = np.clip(new_rows, 0, 1)
            test_copy.loc[:, col] = new_rows
        # return copies to preserve the original
        return test_copy, train_copy
```

```
   # exercise C
   def getClassProbs(self, X_test, X_train, Y_train): # computes for all test␣
↪instances in X_test the posterior class probabilities
       posterior_prob = {} #dictionary
       for i in range(len(X_test)):   #iterate over all instances in X_test
           test_instance = X_test.iloc[i]
           distances = []

           for j in range(len(X_train)):  #iterate over all instances in␣
↪X_train
               train_instance = X_train.iloc[j]
               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.
           df_dists = pd.DataFrame(data=distances, columns=['dist'], index =␣
↪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]

           nmbr_occurences = Y_train[df_knn.index].value_counts() #counts what␣
↪the how many neighbours have the same classification
           interior_dic = dict()

           for instance, occurence in nmbr_occurences.items():
               prob = occurence/self.k
               interior_dic[instance] = prob
           posterior_prob[i] = interior_dic
       return pd.DataFrame(posterior_prob).fillna(0)

   #Exercise D
   def getPrediction(self, X_test):
       prob_df = self.getClassProbs(X_test, self.X_train, self.Y_train)
       interior_dic = dict()

       # knn regression (1/k) sum(yi)
       for index, price_prob in prob_df.iteritems():
           sum_price = 0
           for price, prob in price_prob.items():
               if (prob != 0):
```

```
                    sum_price += price * prob
                #dictionary = dict()

            interior_dic[index] = sum_price

        # regression value equal to the average of y values in Y_train of the␣
↪k-nearest neighbors of the instance in X_train
        return pd.DataFrame.from_dict(interior_dic, orient = 'index')
```

2 Task B

2.1 B-1: Testing the knn classifier on the datasets

Test the knn classifier on glass and diabetis data set for the case when the data is not normalized and the case when the data is normalized. Indicate whether the training and hold-out accuracy rates improve with normalization.

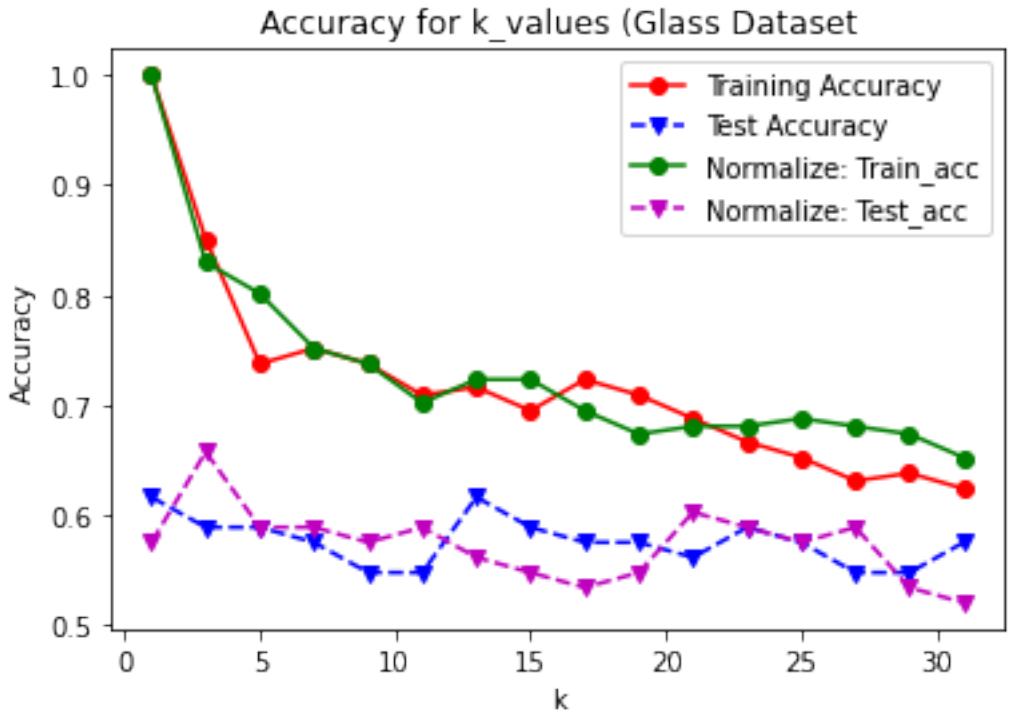
[3]:
```
def test_classifier(dataset_name, normalize, k_range):
    ######################################################
    # Hold-out testing: Training and Test set creation
    ######################################################

    data = pd.read_csv(dataset_name + '.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)
    if normalize:
        knn = kNN()
        X_test, X_train = knn.normalize(X_test, X_train) # normalization using␣
↪max values

    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) # saves data that is being passed in self

        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

    return trainAcc, testAcc
```

2.1.1 Glass dataset

[4]:
```
# 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, testAcc = test_classifier('glass', False, k_range) # Glass data set␣
↪without Normalization
trainAcc_norm, testAcc_norm = test_classifier('glass', True, k_range) # Glass␣
↪data set with Normalization

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--',
            k_range, trainAcc_norm,'go-',k_range, testAcc_norm,'mv--')
plt.legend(['Training Accuracy','Test Accuracy', 'Normalize: Train_acc',␣
↪'Normalize: Test_acc'])
plt.title('Accuracy for k_values (Glass Dataset)')
plt.xlabel('k')
plt.ylabel('Accuracy')
```

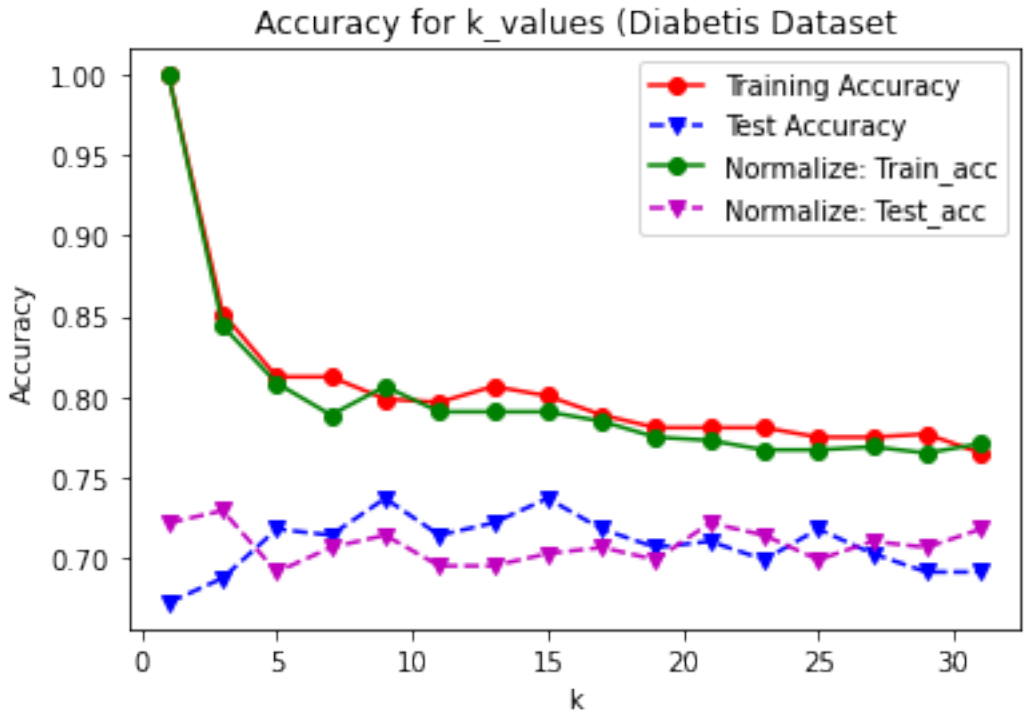
[4]:
```
Text(0, 0.5, 'Accuracy')
```



2.1.2 Diabetis dataset

[5]:
```
# 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, testAcc = test_classifier('diabetes', False, k_range) # Diabetis data␣
 ↪set without Normalization
trainAcc_norm, testAcc_norm = test_classifier('diabetes', True, k_range) #␣
 ↪Diabetis data set with Normalization

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--',
             k_range, trainAcc_norm,'go-',k_range, testAcc_norm,'mv--')
plt.legend(['Training Accuracy','Test Accuracy', 'Normalize: Train_acc',␣
 ↪'Normalize: Test_acc'])
plt.title('Accuracy for k_values (Diabetis Dataset)')
plt.xlabel('k')
plt.ylabel('Accuracy')
```

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



2.1.3 Answer B-1

For the Glass dataset, we can see a peak in the normalized data for k_values in both the Training (at k = 5) and Testing (at k = 3) accuracies. This shows an improvement compared to the non-normalized data. This improvement only holds for few k_values. with k<2 or k>6, we cannot see any improvement anymore and both the normalized and non-normalized data show very similar accuracies.

The plot for the diabetis data set does not demonstrate any clear peaks in the normalized data. Therfore we cannot see any improvement in normalizing the data in the accuracy measures. So, normalizing the datasets X_train and X_test with respect to the training and testing accuracies does not give us a better accuracy.

This shows that normalization does not have to always be good, and that it depends on the data you have. Gnerally, normalisation helpsus give the same level of importance to each instance. However, it can happen, that it causes the importance of originally important features to decrease, resulting in a misclassification of certian instances. This will lead the accuracy to decrease. Additionally to what was already said above, on both plots, even though the accuracies of normalized and non-normalized data are very similar, for different values of k, either the non-normalized or normalized data is slightly better. This is caused by effects of normalization just described.

2.2 B-2: Testing the knn classifier on the datasets

Test the kNN classifier on the glass classification data sets the data is normalized for different values of the exp parameter of the Minkowski distance. Indicate whether the training and hold-out accuracy rates changes due to exp.

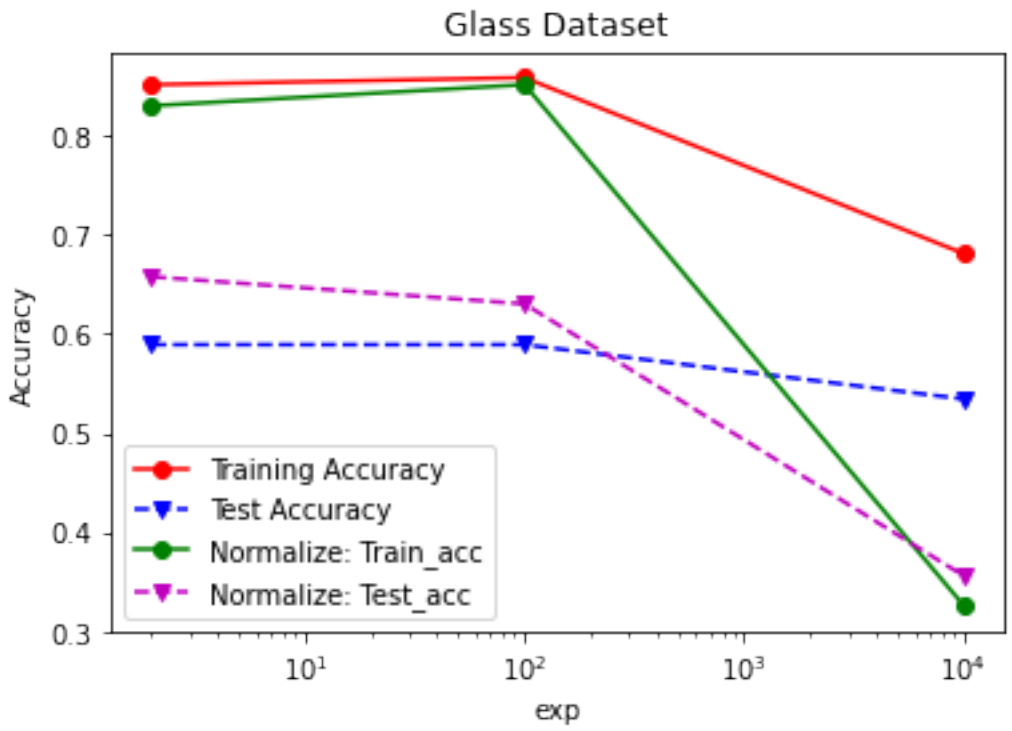
[6]:
```
def test_classifier(dataset_name, normalize, exp_range):
    data = pd.read_csv(dataset_name + '.csv')
    data.head()
    Y = data['class']
    X = data.drop(['class'],axis=1)

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

    if normalize:
        knn = kNN()
        X_test, X_train = knn.normalize(X_test, X_train) # normalization using␣
↪max values

    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

    return trainAcc, testAcc
```

[7]:
```
# range for the values of parameter exp for kNN
exp_range = [2,  100, 10000]

trainAcc, testAcc = test_classifier('glass', False, exp_range) # Glass data set␣
 ↪without Normalization
trainAcc_norm, testAcc_norm = test_classifier('glass', True, exp_range) #Glass␣
 ↪data set with Normalization

plt.plot(exp_range,trainAcc,'ro-',exp_range,testAcc,'bv--',
            exp_range, trainAcc_norm,'go-',exp_range, testAcc_norm,'mv--')
plt.legend(['Training Accuracy','Test Accuracy', 'Normalize: Train_acc',␣
 ↪'Normalize: Test_acc'])
plt.title('Glass Dataset')
plt.xscale("log") #use a logarithmic scale to make changes more visible
plt.xlabel('exp')
plt.ylabel('Accuracy')
```

```
<ipython-input-2-03c6cfded83b>:65: RuntimeWarning: overflow encountered in
double_scalars
  distance = distance + abs(x1[i] - x2[i])**self.exp
```

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



2.2.1 Answer B-2

For the training accuracy we can clearly see that the normalized data performs worse than the non-normalized one, and therefore does not provide any improvement. For the Test data, the normalized data performs better for a small amount of exp, however, since we have adjusted the scale on the x-axis to be logarithmic, we can see that for the great majority of exp values, the non-normalized data performs better.

2.3 Task C

[18]:
```
data = pd.read_csv('glass.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)
x = kNN()
pred = x.getClassProbs(X_test, X_train, Y_train)

pred
```

[18]:
```
                            0    1         2         3    4    5    6   \
'build wind float'      0.666667  1.0  0.000000  0.333333  1.0  0.0  1.0
'vehic wind float'      0.333333  0.0  0.000000  0.000000  0.0  0.0  0.0
headlamps               0.000000  0.0  0.666667  0.000000  0.0  1.0  0.0
containers              0.000000  0.0  0.333333  0.000000  0.0  0.0  0.0
'build wind non-float'  0.000000  0.0  0.000000  0.666667  0.0  0.0  0.0
tableware               0.000000  0.0  0.000000  0.000000  0.0  0.0  0.0

                            7         8    9   …   63   64   65   66   67  \
'build wind float'      0.333333  0.000000  0.0  …  0.0  1.0  1.0  0.0  0.0
'vehic wind float'      0.000000  0.666667  0.0  …  0.0  0.0  0.0  0.0  0.0
headlamps               0.000000  0.000000  0.0  …  0.0  0.0  0.0  1.0  0.0
containers              0.000000  0.000000  0.0  …  1.0  0.0  0.0  0.0  0.0
'build wind non-float'  0.333333  0.333333  1.0  …  0.0  0.0  0.0  0.0  1.0
tableware               0.333333  0.000000  0.0  …  0.0  0.0  0.0  0.0  0.0

                           68        69        70   71        72
'build wind float'      0.333333  0.000000  0.000000  0.0  0.333333
'vehic wind float'      0.666667  0.333333  0.000000  0.0  0.000000
headlamps               0.000000  0.333333  0.333333  0.0  0.000000
containers              0.000000  0.000000  0.000000  0.0  0.000000
'build wind non-float'  0.000000  0.333333  0.333333  1.0  0.666667
tableware               0.000000  0.000000  0.333333  0.0  0.000000

[6 rows x 73 columns]
```

3 Task D

[19]:
```
from sklearn.metrics import mean_absolute_error
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
mean_absolute_error(y_true, y_pred)
```

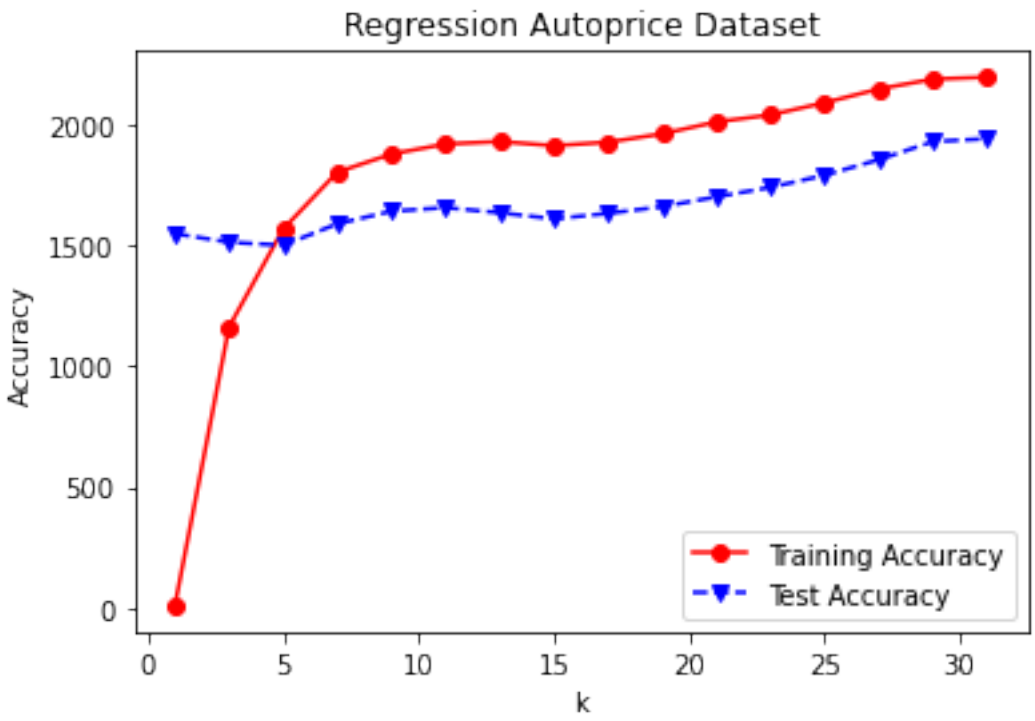
[19]: 0.5

[20]:
```
def regression(dataset_name, k_range):

    data = pd.read_csv(dataset_name + '.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)

    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) # saves data that is being passed in self

        Y_predTrain = clf.getPrediction(X_train)
        Y_predTest = clf.getPrediction(X_test)
        trainAcc[index] = mean_absolute_error(Y_train, Y_predTrain)
        testAc[index] = mean_absolute_error(Y_test, Y_predTest)
        index += 1

    return trainAcc, testAcc
```

[21]:
```
# 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, testAcc = regression('autoprice', k_range)

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

[21]:
```
Text(0, 0.5, 'Accuracy')
```


Regression Autoprice Dataset
Accuracy
2000
1500
1000
500
0
0
5
10
15
20
25
30
k
Training Accuracy
Test Accuracy

[ ]: