Machine Learning Lab 2

Martin Michaux i6220118

November 16, 2020

[9]:
```python
# Class of k-Nearest Neigbor Classifier
import pandas as pd
from sklearn import preprocessing

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␣
↪corrends 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:
    #I created a static method that takes X_train and X_test as paremeters and␣
↪returns the same values normalized the the max of X_train
    @staticmethod
    def normalize(X_train,X_test):
        max = X_train.max()
        X_train = X_train/max
        X_test = X_test/max
        return X_train, X_test


    #EXERCISE C:
    def getClassProbs(self, X_test):
    # X_test is the test data given with input attributes. Rows correpond to␣
↪instances
    # Method outputs posterior class probabilities of vector Y_pred_test:  n-th␣
↪element of Y_pred_test is the set of class probas. for n-th instance in X_test

        #compute all possible class as columns (the row will represent each test␣
↪instance and their probas)
        possibleClass = list(pd.unique(self.Y_train))

        #table that will contain each test instance as row with their probas.␣
↪and colum as all possible class
        Y_postClassProba_test = pd.DataFrame(columns = possibleClass)

        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()

            #all points around the test instance that are in the k-range
            innerPoints = self.Y_train[df_knn.index]
            #take the nbr of points in that k-range to calculate later the␣
↪probas.
            totalNbrOfInstances = len(innerPoints)

            probs = []
            #calculate for all possible class, the probas. of each test instance
            for j in range(len(possibleClass)):
                #if the occurence of the class is different than 0
                try:
                    probs.append((predictions.iloc[j]/totalNbrOfInstances).
↪item())
                #if the occurence is equal to 0, its proba is also 0
                except:
                    probs.append(0)


            # add the probas. of y_pred_test for all the test instances in X_test
```

```
            Y_postClassProba_test.loc[i] = probs
        return Y_postClassProba_test

    #EXERCISE D:
    def getPrediction(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()

            #calculates the average prediction of the classes for  each test␣
↪instance depending on the regression function
            average = sum(predictions.index)/self.k

            # 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(average)

        return Y_pred_test
```

1 QUESTION B

2 DIABETE DATA

[3]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from numpy.random import random
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
##################################################
# Hold-out testing: Training and Test set creation
##################################################
# TEST DIABETE DATA WITHOUT NORMALIZATION

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))

#data testing with data not normalized depending on the k value
```

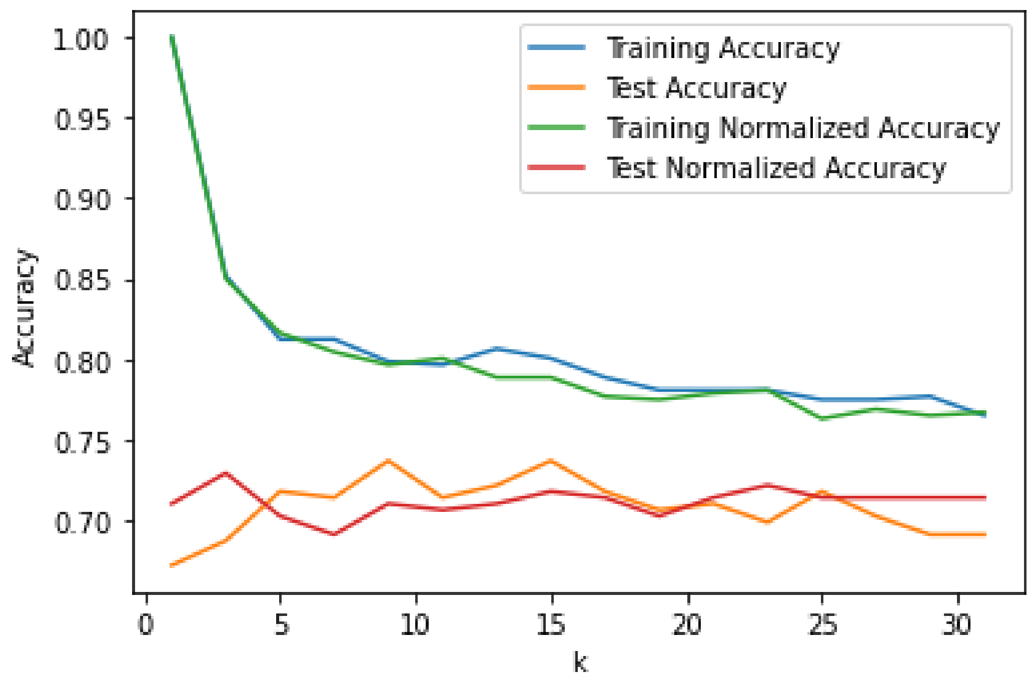
```python
index = 0
for k  in  k_range:
    clf = kNN(k)
    #for not normalized data
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1
```

[4]:

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from numpy.random import random
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
##################################################
# Hold-out testing: Training and Test set creation
##################################################
# TEST DIABETE DATA WITH NORMALIZATION

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)
#normalize the data
X_train, X_test = kNN.normalize(X_train,X_test)

# 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]

trainAccNormalized = np.zeros(len(k_range))
testAccNormalized = np.zeros(len(k_range))

#data testing with data normalized depending on the k value
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)
    trainAccNormalized[index] = accuracy_score(Y_train, Y_predTrain)
```

```
    testAccNormalized[index] = accuracy_score(Y_test, Y_predTest)
    index += 1

plt.
 ↪plot(k_range,trainAcc,k_range,testAcc,k_range,trainAccNormalized,k_range,testAccNormalized)
plt.legend(['Training Accuracy','Test Accuracy','Training Normalized␣
 ↪Accuracy','Test Normalized Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
```

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



In the accuracy rate graph above, we can clearly see than there is not a big difference of accuracy between the normalized and original data. Nevertheless, the normalization of the data does modify the calculation of the prediction. Indeed, after being normalized, the distance between each attribute instance point has changed (smaller), so very high original data is rescaled to a smaller one, so the k-range may contain that data while the k-range of the original data may not contain it because of its high distance difference. So a higher accuracy in the normalized data is due to the fact that the standard deviation between the normalized data is much smaller than the original data.

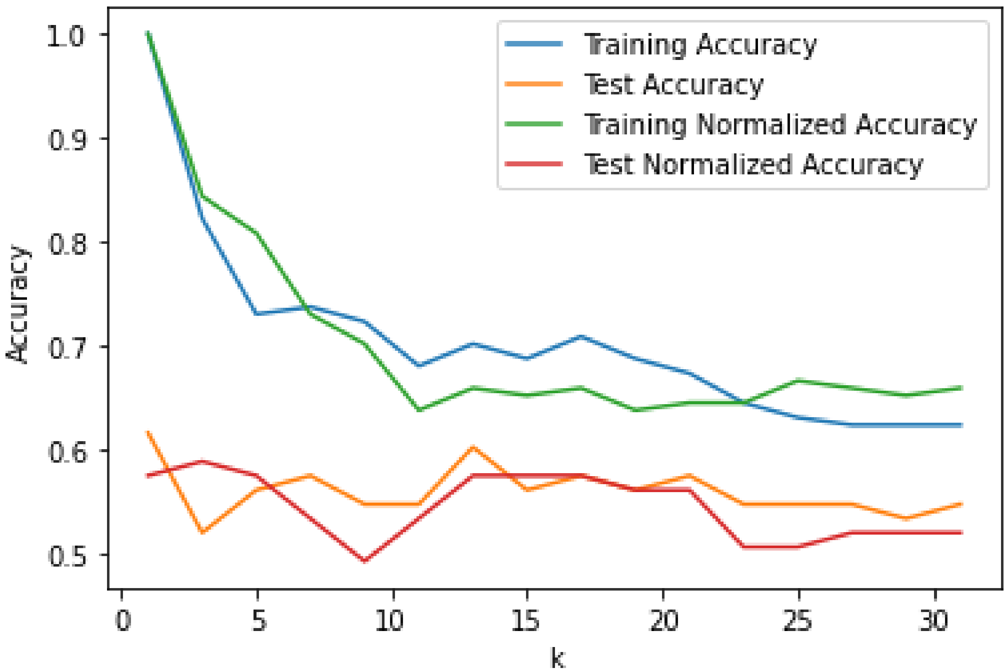
3 GLASS DATA

[5]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

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

##################################################
# Hold-out testing: Training and Test set creation
##################################################
# TESTING GLASS DATA WITHOUT NORMALIZATION ON K VALUE
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))

#data testing with data not normalized depending on the k value
index = 0
for k  in  k_range:
    clf = kNN(k)
    #for not normalized data
    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
```

[6]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

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

```
from sklearn.model_selection import train_test_split

################################################
# Hold-out testing: Training and Test set creation
################################################
# TESTING GLASS DATA WITH NORMALIZATION ON K VALUE
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)
#normalize the data
X_train, X_test = kNN.normalize(X_train,X_test)

# 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]

trainAccNormalized = np.zeros(len(k_range))
testAccNormalized = np.zeros(len(k_range))

#data testing with data normalized depending on the k value
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)
    trainAccNormalized[index] = accuracy_score(Y_train, Y_predTrain)
    testAccNormalized[index] = accuracy_score(Y_test, Y_predTest)
    index += 1

plt.
↪plot(k_range,trainAcc,k_range,testAcc,k_range,trainAccNormalized,k_range,testAccNormalized)
plt.legend(['Training Accuracy','Test Accuracy','Training Normalized␣
↪Accuracy','Test Normalized Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
```

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



In the accuracy rate graph above, we can clearly see than there is not a big difference of accuracy between the normalized and original data. Nevertheless, the normalization of the data does modify the calculation of the prediction. Indeed, after being normalized, the distance between each attribute instance point has changed (smaller), so very high original data is rescaled to a smaller one, so the k-range may contain that data while the k-range of the original data may not contain it because of its high distance difference. So a higher accuracy in the normalized data is due to the fact that the standard deviation between the normalized data is much smaller than the original data.

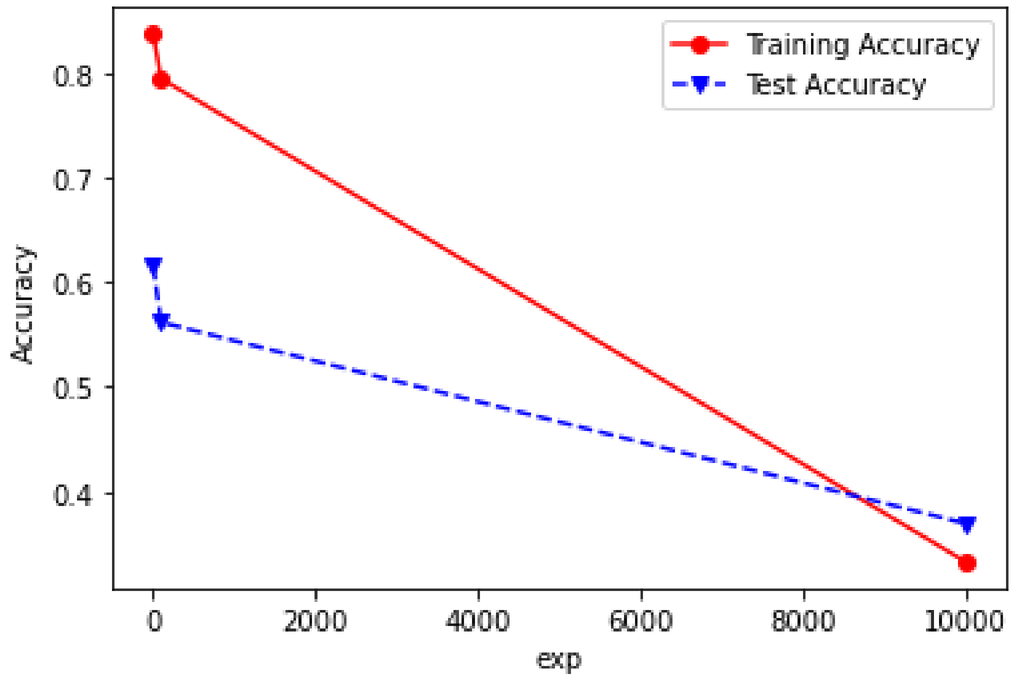
4 QUESTION B WITH EXP RANGE ON GLASS DATA

[7]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

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

##################################################
# Hold-out testing: Training and Test set creation
##################################################
# TESTING GLASS DATA WITH NORMALIZATION ON EXP OF MINK. DISTANCE
data = pd.read_csv('glass.csv')
```

```python
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)
#normalize the data
X_train, X_test = kNN.normalize(X_train,X_test)

exp_range = [2,  100, 10000]
trainAccExpNorm = np.zeros(len(exp_range))
testAccExpNorm = np.zeros(len(exp_range))
#data testing with data normalized depending on the exp of the Minkowski distance
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)
    trainAccExpNorm[index] = accuracy_score(Y_train, Y_predTrain)
    testAccExpNorm[index] = accuracy_score(Y_test, Y_predTest)
    index += 1

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

```
<ipython-input-2-e63bfd26ae6a>:70: RuntimeWarning: overflow encountered in
double_scalars
  distance = distance + abs(x1[i] - x2[i])**self.exp
```

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



In the accuracy rate graph above, we can clearly see that the higher the exp of the Minkowski distance is, the accuracy of both data decreases drastically But, after a specifif exp value, wen see that the accuracy of the test data is better than the instance data This is due to the fact that, because the data is normalized, the distance are smaller, therefore, if those distances are high exponenital, the distances will be even drastically smaller. So, it makes the accuracy of both smaller.

5 QUESTION C

6 DIABETE DATA

[8]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

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

##################################################
# 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)
#normalize the data to have better results
X_train, X_test = kNN.normalize(X_train,X_test)

#I chose to calculates the probas. with a k=10
k = 10
clf = kNN(k)
clf.fit(X_train, Y_train)
y_pred_probs = clf.getClassProbs(X_test)
print(y_pred_probs)
```

```
     tested_positive  tested_negative
0                0.5              0.5
1                0.9              0.1
2                0.7              0.3
3                1.0              0.0
4                0.9              0.1
..               ...              ...
257              1.0              0.0
258              0.5              0.5
259              0.5              0.5
260              1.0              0.0
261              0.6              0.4

[262 rows x 2 columns]
```

7 GLASS DATA

[9]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

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

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

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

```python
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)
#normalize the data to have better results
X_train, X_test = kNN.normalize(X_train,X_test)

#I chose to calculates the probas. with a k=10
k = 10
clf = kNN(k)
clf.fit(X_train, Y_train)
y_pred_probs = clf.getClassProbs(X_test)
#print the table probas. table
print(y_pred_probs)
```

```
    'build wind float'  'build wind non-float'  headlamps  'vehic wind float'  \
0                  0.6                     0.4        0.0                 0.0
1                  0.5                     0.4        0.1                 0.0
2                  0.8                     0.2        0.0                 0.0
3                  0.7                     0.2        0.1                 0.0
4                  0.5                     0.5        0.0                 0.0
..                 ...                     ...        ...                 ...
68                 0.7                     0.2        0.1                 0.0
69                 0.4                     0.3        0.3                 0.0
70                 0.6                     0.2        0.1                 0.1
71                 0.5                     0.5        0.0                 0.0
72                 0.5                     0.3        0.2                 0.0

    containers  tableware
0          0.0        0.0
1          0.0        0.0
2          0.0        0.0
3          0.0        0.0
4          0.0        0.0
..         ...        ...
68         0.0        0.0
69         0.0        0.0
70         0.0        0.0
71         0.0        0.0
72         0.0        0.0

[73 rows x 6 columns]
```

8 QUESTION D ON AUTORPICE DATA

[8]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error

data = pd.read_csv('autoprice.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)
X_train, X_test = kNN.normalize(X_train,X_test)

#I chose to calculates the regression prediction with a k=10
k = 10

clf = kNN(k)
clf.fit(X_train, Y_train)
Y_predTest = clf.getPrediction(X_test)
#calculates the mean absolute error
testMeanAbsError = mean_absolute_error(Y_test, Y_predTest)

print(testMeanAbsError)
```

```
1957.612727272727
```

[ ]:
```

```