```
# coding: utf-8

# #### CS 6316 : Machine Learning
# ## In-class activity: K-Nearest Neighbor Algorithm

# #####    D.J. Anderson    #####
# #####          dra2zp          #####
# ##### In Class Activity 4 #####
#
# In this activity, you will further explore KNN algorithm through python code.
#
# The data set weâ€™ll be using is the Iris Flower Dataset (IFD) which was first introduced in 1936
by the famous statistician Ronald Fisher and consists of 50 observations from each of three species
of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each
sample: the length and the width of the sepals and petals. Our goal is to train the KNN algorithm to
be able to distinguish the species from one another given the measurements of the 4 features.
#
# Lets start by running an example code using scikit-learn to train a KNN classifier and evaluate its
performance on the data set.
# There is 4 step modeling pattern:
#
# 1. Import the learning algorithm
# 2. Instantiate the model
# 3. Learn the model
# 4. Predict the response
#
#
# The following steps will print out accuracy score for K=3:

# In[41]:

# 1) loading libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from collections import Counter

from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

# define column names
names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class']

# loading training data
df = pd.read_csv('iris.data.txt', header=None, names=names)
df.head()

# making our predictions
predictions = []

# create design matrix X and target vector y
X = np.array(df.ix[:, 0:4])     # end index is exclusive
y = np.array(df['class'])   # another way of indexing a pandas df

# split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

```
# Now try traininging and comparing performances for different k values (from 1 to 50).

# In[42]:

# creating odd list of K for KNN
myList = list(range(1,50))

# subsetting just the odd ones
neighbors = list(filter(lambda x: x % 2 != 0, myList))

# empty list that will hold cross validation scores
cv_scores = []

# perform 10-fold cross validation we are already familiar with
for k in neighbors:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=10, scoring='accuracy')
    cv_scores.append(scores.mean())

# changing to misclassification error
MSE = [1 - x for x in cv_scores]




# In[43]:

# Let's try to discover what is the best value of k
# Run the following cell to show the table


# In[44]:

print ("k          Score          MSE")
for i in range(len(neighbors)):
    print ('%d          %.5f          %.5f' % (neighbors[i], cv_scores[i], MSE[i]))



# What is the optimal value of K?
#
# Write your answer here: 7
# (Double-click cell to edit)
#
#
# Lets make a plot to visually compare performances at each k value.

# In[45]:

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
print("The optimal number of neighbors is %d" % optimal_k)

# plot misclassification error vs k
plt.plot(neighbors, MSE)
plt.xlabel('Number of Neighbors K')
plt.ylabel('Misclassification Error')
plt.show()


# We can confirm that the optimal value of K (see above)
#
# Now, we are going to implement KNN prediction from scratch.
#
```

```
#
# Note that KNN is a instance-based learning, which means that our algorithm doesnâ€™t explicitly
learn a model. Instead, it chooses to memorize the training instances which are subsequently used as
â€œknowledgeâ€ for the prediction phase. Concretely, this means that only when a query to our
database is made (i.e. when we ask it to predict a label given an input), will the algorithm use the
training instances to spit out an answer.
#
# In our implementation, the training block reduces to just memorizing the training data.
# This mean it just returns.

# In[46]:

def train(X_train, y_train):
    # do nothing
    return


#
# The minimal training phase of KNN comes both at a memory cost, since we must store a potentially
huge data set, as well as a computational cost during test time since classifying a given observation
requires a run down of the whole data set. Practically speaking, this is undesirable since we usually
want fast responses.
#
#
# Now you are going to fill in some missing lines in predict phase. Here is general workflow:
# 1. Compute the euclidean distance between the â€œnewâ€ observation and all the data points in the
training set.
# 2. Select the K nearest ones and perform a majority vote.
# 3. Assigns the corresponding label to the observation.

# In[47]:

from collections import Counter

def predict(X_train, y_train, x_test, k):
    # create list for distances and targets
    distances = []
    targets = []

    for i in range(len(X_train)):
        # first we compute the Euclidean distance
        # (use x_test and X_train[i, :]. Also, where appropriate, you can use np.sqrt, np.square, and
np.sum...)
        distance = np.sqrt(np.sum(np.square(x_test - X_train[i, :])))
        # add it to list of distances
        distances.append([distance, i])

    # sort the list
    distances = sorted(distances)

    # make a list of the k neighbors' targets
    for i in range(k):
        # (Hint: index receives particular value in distances[something][something])
        index = distances[i][1]
        # (Hint: use y_train and index below)
        targets.append([y_train, index])

    # return most common target
    return Counter(targets).most_common(1)[0][0]


# Lets test our implementation, using the predict() method.
#
# *** NOTE: Fill in "optimalK" below
```

```
# In[48]:

from sklearn.metrics import accuracy_score

def kNearestNeighbor(X_train, y_train, X_test, predictions, k):
    # train on the input data
    train(X_train, y_train)

    # loop over all observations
    for i in range(len(X_test)):
        predictions.append(predict(X_train, y_train, X_test[i, :], k))

# making our predictions
# Using the optimal value of K discovered above
predictions = []
try:
    optimalK = 7 # Add your answer here (and delete line!)
    kNearestNeighbor(X_train, y_train, X_test, predictions, optimalK)
    predictions = np.asarray(predictions)

    # evaluating accuracy
    accuracy = accuracy_score(y_test, predictions) * 100
    print('\nThe accuracy of OUR classifier is %d%%' % accuracy)

except ValueError:
    print('Can\'t have more neighbors than training samples!!') # Need to be careful about value of k


# #### Please turn in your work in pdf format by clicking "File/download as/pdf".
```