HW 5

CS 216, Everything Data, Spring 2020

DUE: Monday Feb. 24 by 4:40 pm (class time)

In this assignment, you will use Numpy to build three different basic classifiers for prediction. You will include all of your answers for this assignment within this notebook. You will then convert your notebook to a .pdf and a .py file to submit to gradescope (submission instructions are included at the bottom).

Please take note of the course collaboration policy (https://sites.duke.edu/compsci216s2020/policies/). You may work alone or with a single partner. If you work with a partner, you may not split up the assignment; you should work together in-person or complete parts independently and come together to discuss your solutions. In either case, you are individually responsible for your work, and should understand everything in your submission.

Part 1: Getting Started with Numpy

Numpy is the standard library for scientific computing with Python, and Numpy arrays are the standard data structure for working with prediction tasks in Python. If you are unfamiliar with Numpy, we reccomend that you start this assignment by reviewing the brief tutorial on Numpy at http://cs231n.github.io/python-numpy-tutorial/#numpy (http://cs231n.github.io/python-numpy-tutorial/#numpy) (just the Numpy section). The Numpy documentation itself also contains an expanded tutorial https://numpy.org/doc/1.17/user/quickstart.html (https://numpy.org/doc/1.17/user/quickstart.html). We mention a few particularly important notes here.

- You can easily turn any Python list into a Numpy array (for example, ar = numpy.array([0,1,2]) creates a Numpy array named ar containing 1, 2, and 3).
- Indexing and slicing 1-d Numpy arrays is similar to indexing and slicing Python lists. For example, ar[1:] returns [1,2].
- Indexing and slicing 2-d Numpy arrays is similar to using the `.iloc` method on a Pandas dataframe. For example, if you have a 2-d Numpy array Mat and you want the entry for row 0, column 5, you would just index Mat[0,5]. If you want column 5, you just write Mat[:,5].
- Operations on Numpy arrays are element-wise *by default*. For example, ar+5 would return [5, 6, 7]. ar+ar would return [0, 2, 4].
- Building on the element-wise theme, writing a boolean condition will return a True/False array. For example, ar==1 would return [False, True, False].
- Similar to Pandas filtering, you can use a boolean mask of this sort to filter a Numpy array. For example, ar[ar >= 1] would return [1, 2].
- Numpy comes with full support for nearly all mathematical computing needs. You can find the reference documentation at https://docs.scipy.org/doc/numpy/reference (https://docs.scipy.org/doc/numpy/reference). Of particular interest are the many mathematical functions implemented in Numpy https://docs.scipy.org/doc/numpy/reference/routines.math (https://docs.scipy.org/doc/numpy/reference/routines.math) and the many random sampling functions implemented in Numpy https://docs.scipy.org/doc/numpy/reference/random/index (https://docs.scipy.org/doc/numpy/reference/random/index).

Once you have familiarized yourself with the basics of Numpy, it's time to start building classifiers for prediction. Run the below codeblocks to import libraries and define a function to calculate the uniform error of a prediction. Recall that the uniform error is just the percentage of predictions that are not the same as the true class.

In [267]:
```
import pandas as pd
import numpy as np
```

In [268]:
```
def uniform_error(prediction, target):
    n = len(prediction)
    if(n != len(target)):
        print('Error: prediction and target should have same length')
        return(1)
    else:
        return((n-np.sum(prediction==target))/n)
```

Part 2: Naive Bayes Classifier

In this part of the assignment, you will implement a simple Naive Bayes classifier for predicting political party membership of members of the house of represenatives based on votes.

Each row corresponds to a member of congress. The first column has a 1 if that member is a republican, and a 0 if that member is a democrat. The next 16 columns contain information about sixteen different votes that were taken in that year; there is a 1 if the member voted positively on that issue, and a 0 if that member voted negatively on that issue. We want to predict the first column based on the next sixteen.

```
In [269]: df_v = pd.read_csv('votes.csv')
          df_v.head()
```

Out[269]:

	republican	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1	1	1	1

The first thing we do when learning a predictive model from data is split the dataset into training and test data. For each, we separate the target variable we want to predict (`y` and `y_test` below), and the variables we want to use to predict (`x` and `x_test` below). We will use our training data to learn our predictive model, and then use the test data to verify its accuracy (it is, of course, no great accomplishment to do well at prediction for the data points the model has already seen in training; you can always just memorize the training data).

Below, we randomly split 1/3 of the data into the test set (`x_test` and `y_test`) and the remaining 2/3 of the data into the training set (`x` and `y`). We do this randomly to ensure that the training and test sets look very similar. You will also note that we convert all of the different datasets into Numpy arrays; `.values` in Pandas converts a Pandas dataframe into a Numpy array. Note that `x` and `x_test` are 2-d arrays, whereas `y` and `y_test` are 1-d arrays. In both cases, `y[i]` corresponds to the row `x[i,:]` . We print the first five rows of `x` and the first five values of `y` .

```
In [270]: # Do not change this code, including the seed
np.random.seed(137486213)
test_indices = np.random.binomial(1, 0.33, df_v.shape[0])
df_v['test'] = test_indices

df_v_test = df_v[df_v['test']==1]
df_v_train = df_v[df_v['test']==0]

x = (df_v_train[['v_1', 'v_2', 'v_3', 'v_4', 'v_5', 'v_6', 'v_7', 'v_8',
'v_9', 'v_10']]).values
x_test = (df_v_test[['v_1', 'v_2', 'v_3', 'v_4', 'v_5', 'v_6', 'v_7', 'v
_8', 'v_9', 'v_10']]).values

y = df_v_train['republican'].values
y_test = df_v_test['republican'].values

print(x[0:5,:])
print(y[0:5])
```

```
[[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0]
 [0 1 0 1 1 0 0 0 0 0]]
[0 0 1 0 1]
```

For a baseline, let's see what kind of error we get if we just guess randomly. Below, we generate a "prediction" for the test data by ignoring the test data altogether (we don't use `x_test` at all below) and just flipping a coin to predict 0 or 1 for each data point. Note that the prediction is also just a 1-d array where the entry at index `i` corresponds to a prediction for `y_test[i]`. As expected, this yields fairly high error (0.5 in expectation). In what follows, you will train a Naive Bayes classifier to improve on this performance.

```
In [271]: random_guess = np.random.randint(2, size=len(y_test))
print(uniform_error(random_guess, y_test))
```

```
0.44680851063829785
```

Problem A

Recall how Naive Bayes classifiers work: for a data point `x_test[i,:]` we want to predict the class `j` that maximizes the liklihood: the conditional probability of seeing `x[i,:]` given `j`. We make the assumption that the different features are independent given the class, that is, that we can break this probability up into the product over columns `c` of the probability of `x[i,c]` given `j`, all times the probability that `y=j`.

First, compute all of the conditional probabilities on the training data. That is, for every column `c`, compute the conditional probability for a random representative `i` that `x[i,c]=1` given `y=1`, and the same conditional probability given `y=0` (note that the conditional probability that `x[i,c]=0` given `y=1` is just one minus the probability that `x[i,c]=1` given `y=1`, and the same for `y=0`). You will also need to compute the probability that `y=1` and the probability that `y=0`. Print the resulting probabilities.

```
In [272]: total = len(x) #294
          demCount = 0 # 142
          repCount = 0 # 152
          for i in y:
              if i == 1:
                  repCount += 1
              else:
                  demCount += 1
          print(demCount/total)
          print(repCount/total)
          for c in range(10):
              repYes = 0 # RepNo = 152 - RepYes
              demYes = 0 # DemNo = 122 - DemYes
              for i in (range(len(y))):
                  if y[i] == 1 and x[i,c] == 1:
                      repYes += 1
                  if y[i] == 0 and x[i,c] == 1:
                      demYes += 1
              yesRepProb = round((repYes/repCount), 3)
              yesDemProb = round((demYes/demCount), 3)
              print("v_",c+1,"P(Yes Rep)", yesRepProb,"P(No Rep)",round((1-yesRepP
          rob),2),
                    "P(Yes Dem)", yesDemProb, "P(No Dem)", round((1-yesDemProb),3
          ))
```

```
0.48299319727891155
0.5170068027210885
v_ 1 P(Yes Rep) 0.632 P(No Rep) 0.37 P(Yes Dem) 0.317 P(No Dem) 0.683
v_ 2 P(Yes Rep) 0.737 P(No Rep) 0.26 P(Yes Dem) 0.268 P(No Dem) 0.732
v_ 3 P(Yes Rep) 0.592 P(No Rep) 0.41 P(Yes Dem) 0.472 P(No Dem) 0.528
v_ 4 P(Yes Rep) 0.987 P(No Rep) 0.01 P(Yes Dem) 0.113 P(No Dem) 0.887
v_ 5 P(Yes Rep) 0.934 P(No Rep) 0.07 P(Yes Dem) 0.275 P(No Dem) 0.725
v_ 6 P(Yes Rep) 0.658 P(No Rep) 0.34 P(Yes Dem) 0.423 P(No Dem) 0.577
v_ 7 P(Yes Rep) 0.579 P(No Rep) 0.42 P(Yes Dem) 0.465 P(No Dem) 0.535
v_ 8 P(Yes Rep) 0.579 P(No Rep) 0.42 P(Yes Dem) 0.444 P(No Dem) 0.556
v_ 9 P(Yes Rep) 0.77 P(No Rep) 0.23 P(Yes Dem) 0.345 P(No Dem) 0.655
v_ 10 P(Yes Rep) 0.586 P(No Rep) 0.41 P(Yes Dem) 0.31 P(No Dem) 0.69
```

Problem B

Now that you have computed the marginals, use them to predict, for every test data point (that is, for each row in `x_test`), whether the given member is a republican (1) or a a democrat (0). More concretely, you should compute an array of the same length as `y_test` where for each entry, your prediction is a `1` or `0`. Do so by selecting, for each data point, the class (1 or 0) that maximizes the above likelihood under the independence assumption of the Naive Bayes classifier. You do not need to use Laplace smoothing for this example. Once you have your prediction, compute and print the uniform error of your prediction by comparing to `y_test`. For full credit, your classifier should have uniform error less than 0.15 on this particular data.

In [273]:
```
total = len(x)
demCount = 0
repCount = 0
abcd=[]
efgh=[]
for i in y:
    if i == 1:
        repCount += 1
    else:
        demCount += 1
print(demCount/total)
print(repCount/total)
for c in range(10):
    repYes = 0
    demYes = 0
    for i in (range(len(y))):
        if y[i] == 1 and x[i,c] == 1:
            repYes += 1
        if y[i] == 0 and x[i,c] == 1:
            demYes += 1
    abcd.append(round((repYes/repCount), 3))
    efgh.append(round((demYes/demCount), 3))
print(abcd, efgh)
ab=[]
for i in (range(len(x_test))):
    a=0
    b=0
    for c in range(10):
        if x_test[i,c] == 1:
            a+=np.log(abcd[c])
            b+=np.log(efgh[c])
        else:
            a+=np.log(1-abcd[c])
            b+=np.log(1-efgh[c])
    if(a>b):
        ab.append(1)
    else:
        ab.append(0)
print(ab)
print(uniform_error(ab, y_test))
```

```
0.48299319727891155
0.5170068027210885
[0.632, 0.737, 0.592, 0.987, 0.934, 0.658, 0.579, 0.579, 0.77, 0.586]
[0.317, 0.268, 0.472, 0.113, 0.275, 0.423, 0.465, 0.444, 0.345, 0.31]
[0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0,
0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0,
1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0,
1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0]
0.11347517730496454
```

Part 3: Decision Tree Classification

In this part, we will tackle the same prediction task as in Part 2 with the same data. However, instead of a Naive Bayes classifier, we will use a very simple decision tree that simply chooses a single feature on which to split the data.

Problem C

Recall that decision trees split the training data into different sets based on their values for a given feature. Concretely, we will choose a single column `c`, and split the data into two sets, one for the data points with `x[i,c]=1` and another for the data points with `x[i,c]=0`. Then, to make a prediction for a given test point, we simply check which of the sets it would go into, and predict the most common class (`y=1` or `y=0`) among the training data split into that set.

To decide on the best feature to use for splitting, we calculate the information gain of a split as the entropy of the original distribution (over classes `y=1` and `y=0`) minus the weighted average of the entropy of the two distributions represented by the split. To begin, compute the information gain for every feature (that is, every column of `x`) in the training data. Which feature (that is, which column) has the greatest information gain?

```
In [277]: demoftotal = 142/294
          repoftotal = 152/294
          import math
          def entropy(probA, probB):
              e = -(probA)*(np.log2(probA))-(probB)*(np.log2(probB))
              return e

          for c in range(10):
              group1 = []
              group0 = []
              for i in range(len(x)):
                  if x[i,c] == 1:
                      group1.append(i)
                  else:
                      group0.append(i)

              group1rep = 0 #96, 141 - 96 = 45 dem
              group1dem = len(group1) - group1rep
              group0rep = 0 #56, 153 - 56 = 97 dem
              group0dem = len(group0) - group0rep
              for k in group1:
                  if y[k] == 1:
                      group1rep += 1
              for k in group0:
                  if y[k] == 0:
                      group0rep += 1
              group1repoftotal = group1rep/(len(group1))
              group1demoftotal = group1dem/(len(group1))
              group0repoftotal = group0rep/(len(group0))
              group0demoftotal = group0dem/(len(group0))

              group1oftotal = (len(group1))/len(y)
              group0oftotal = (len(group0))/len(y)

              A = entropy(group1repoftotal,group1demoftotal)
              B = entropy(group0repoftotal,group0demoftotal)

              weightedAvg= group1oftotal*A + group0oftotal*B
              infoGain = entropy(demoftotal,repoftotal) - weightedAvg
              print("column",c + 1,"Info Gain is", infoGain)
```

```
column 1 Info Gain is 0.6011530325126268
column 2 Info Gain is 0.6725312409536987
column 3 Info Gain is 0.5316819836849563
column 4 Info Gain is 0.9148256993269871
column 5 Info Gain is 0.7832370494685652
column 6 Info Gain is 0.570909376254024
column 7 Info Gain is 0.5296746427266231
column 8 Info Gain is 0.5359653239754248
column 9 Info Gain is 0.6525490630119604
column 10 Info Gain is 0.5849879103172203
```

v_4 is the feature(column) that has the greatest information gain.

Problem D

Now that you have identified the feature with the greatest information gain, split the training data into two sets based on that feature. For each of the two sets, find the most common class (`y=1` or `y=0`) among that set. Use that information make predictions for the test data as in Part 1. Also as in Part 1, once you have your prediction, compute and print the uniform error of your prediction by comparing to `y_test` . For full credit, your classifier should have uniform error less than 0.15 on this particular data.

In [278]:
```
v4group1 = []
v4group0 = []
v4group1rep = 0
v4group0rep = 0
for i in range(len(x)):
    if x[i,3] == 1:
        v4group1.append(i)
    else:
        v4group0.append(i)
for k in v4group1:
    if y[k] == 1:
        v4group1rep += 1
for k in v4group0:
    if y[k] == 1:
        v4group0rep += 1
print(v4group1rep/(len(v4group1)))
print(v4group0rep/(len(v4group0)))
v4group0dem = len(v4group0) - v4group0rep
print(v4group0dem)
print("For column 4, the group that voted NO for were 98% dem")
#This means that people that have a 0 in column 4 are most likely a demo
crat.
predictarray = []
for z in range(len(y_test)):
    if x_test[z,3] == 0:
        predictarray.append(0)
    else:
        predictarray.append(1)
print(predictarray)
print(uniform_error(predictarray,y_test))
```

```
0.9036144578313253
0.015625
126
For column 4, the group that voted NO for were 98% dem
[0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0,
0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0,
1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1,
1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0]
0.0851063829787234
```

Part 4: k-Nearest Neighbor Classification

In this part, we will look at a new dataset, `iris.csv` , and explore a different technique for classification, the k-nearest neighbor classifier. This dataset contains measurements of 150 flowers of three different types. The measurements (the first four columns) are all physical measurements in centimeters, and the flower types are designated by 0, 1, and 2 in the rightmost column. Our goal is to predict the flower type from measurements of the physical dimensions of the flower. Note that unlike in Parts 1 and 2, our features are no longer categorial, but instead are numerical. That would require some adapting of our previous methods, but k-nearest neighbor actually works very well with many numerical features all of which are on the same scale.

In [279]:
```
df_iris = pd.read_csv('iris.csv')
df_iris.head()
```

Out[279]:
```
   sepal_length  sepal_width  petal_length  petal_width  flower_type
0           5.1          3.5           1.4          0.2            0
1           4.9          3.0           1.4          0.2            0
2           4.7          3.2           1.3          0.2            0
3           4.6          3.1           1.5          0.2            0
4           5.0          3.6           1.4          0.2            0
```

As before, we split the dataset into training and test datasets, and separate the target `y` form the data we want to use to predict the target `x` . Review the discussion in Part 2 for more details.

In [280]:
```
# Do not change this code, including the seed
np.random.seed(137486213)
test_indices = np.random.binomial(1, 0.33, df_iris.shape[0])
df_iris['test'] = test_indices

df_train = df_iris[df_iris['test']==0]
df_test = df_iris[df_iris['test']==1]

x = df_train[['sepal_length', 'sepal_width', 'petal_length', 'petal_widt
h']].values
x_test = df_test[['sepal_length', 'sepal_width', 'petal_length', 'petal_
width']].values

y = df_train['flower_type'].values
y_test = df_test['flower_type'].values

print(x[0:5,:])
print(y[0:5])
```

```
[[4.9 3.  1.4 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]]
[0 0 0 0 0]
```

Note that the uniform error of a random guess on this dataset is higher than in Parts 2 and 3, because here we are trying to classify into three possible classes (`y` can be 0, 1, or 2) instead of just two classes.

In [281]:
```
random_guess = np.random.randint(3, size=len(y_test))
print(uniform_error(random_guess, y_test))
```

```
0.6875
```

Problem E

Recall that the k-nearest neighbor algorithm works by searching for points from the training data that are similar to the point for which we want to make a prediction. We quantiy this notion by employing a distance function. The simplest example is the Euclidean distance function (that is, the distance function from your high school geometry and physics) which, given two points u and v in d dimensions, is given by

$$dist(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{d} (u_i - v_i)^2}.$$

Write a function that computes the Euclidean distance between two points represented as Numpy arrays.

In [282]:
```
def euclidean_distance(u, v):
    return np.sqrt(np.sum((u - v) ** 2))
```

Problem F

Now, to make predictions for a given point `p` (for example, `x_test[0,:]` would be one such point), we first find the `k` points in the training data set `x` (note that each row in `x` is a data point) with minimum distance to the point `p`. `k` is a parameter that can be set to different values, but for the purpose of this assignment, let's use `k=5`. We then predict the class (`y=2` , `y=1` , or `y=0`) that is most common among these `k` points in the training data set.

Use the k-nearest neighbor algorithm to make predictions of the flower type for all of the test data. More concretely, you should compute an array of the same length as `y_test` where for each entry, your prediction is a `2` , `1` , or `0` . Once you have your prediction, compute and print the uniform error of your prediction by comparing to `y_test` . For full credit, your classifier should have uniform error less than 0.1 on this particular data.

In [283]:
```
f=[1,2,7,3]
print(np.amax(f))
```

```
7
```

In [284]:
```python
def mode(list):
    count0=0
    count1=0
    count2=0
    for l in list:
        if(l==0):
            count0+=1
        elif(l==1):
            count1+=1
        else:
            count2+=1
    d=[count0,count1,count2]
    g=np.amax(d)
    if(g==count0):
        return 0
    elif(g==count1):
        return 1
    else:
        return 2
r=[]
for i in range(len(x_test)):
    t=[]
    for c in range(len(x)):
        e=0
        for h in range(4):
            e+=(euclidean_distance(x[c,h], x_test[i,h]))
        t.append([e,c])
    r.append(sorted(t)[0:5])
q=[]
for k in range(len(r)):
    z=[]
    for p in range(5):
        z.append(y[r[k][p][1]])
    q.append(z)
v=[]
for j in range(len(q)):
    v.append(mode(q[j]))
print(v)
print(uniform_error(v, y_test))
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2]
0.0
```

Submitting HW 5

1. Double check that you have written all of your answers along with your supporting work in this notebook. Make sure you save the complete notebook.
2. Double check that your entire notebook runs correctly and generates the expected output. To do so, you can simply select Kernel -> Restart and Run All.
3. You will download two versions of your notebook to submit, a .pdf and a .py. To create a PDF, we reccomend that you select File --> Download as --> HTML (.html). Open the downloaded .html file; it should open in your web broser. Double check that it looks like your notebook, then print a .pdf using your web browser (you should be able to select to print to a pdf on most major web browsers and operating systems). Check your .pdf for readability: If some long cells are being cut off, go back to your notebook and split them into multiple smaller cells. To get the .py file from your notebook, simply select File -> Download as -> Python (.py) (note, we recognize that you may not have written any Python code for this assignment, but will continue the usual workflow for consistency).
4. Upload the .pdf to gradescope under hw 5 report and the .py to gradescope under hw 5 code. If you work with a partner, only submit one document for both of you, but be sure to add your partner using the group feature on gradescope (https://www.gradescope.com/help#help-center-item-student-group-members).