DEPAUL UNIVERSITY

DSC478: Programming Machine Learning Applications

Roselyne Tchoua

rtchoua@depaul.edu

School of Computing, CDM, DePaul University

General Notes

- Submit one notebook per question for clarity
- Zip all files and submit a single file

General Notes

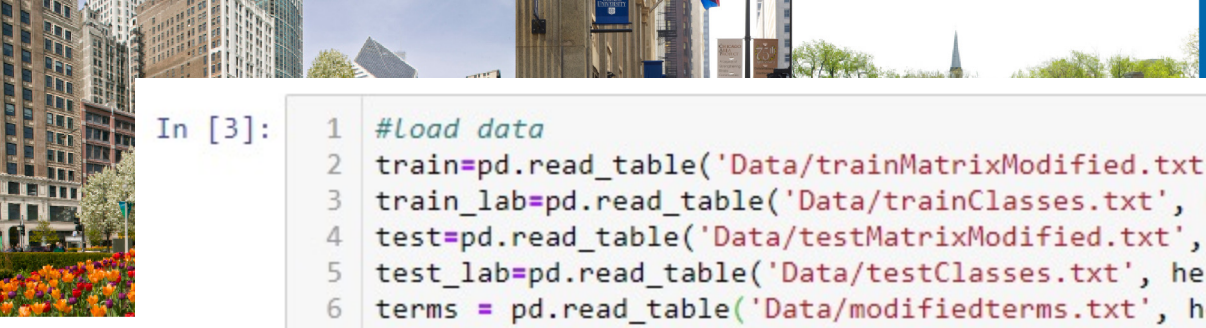
- Remember to use markdown section for any observations. Comments can be used for short code comments but make sure to annotate any plot/table and reply separately to any question asking for your comments.
  - For example, you are asked to compare results with Euclidean and Cosine similarities, or results with and without TFIDF weights, use markdown cells to make observations

Compare/contrast/discuss

- You have definition for bias and variance, you should be able to not only define these concepts but observe them in your results. Especially variance = difference in accuracy with different training sets.
- Your overall observations can be a combination of accuracy observations and what you answered above (bias-variance) with potential suggestions of what you would do next.

Data Format

Even though you would expect a full frame of training and testing, the data for the news dataset is already separated in training vs. testing data and training vs. testing labels. It is also not in the right format for our typical matrix, so you need to transpose it.



In [3]:
```
#load data
train=pd.read_table('Data/trainMatrixModified.txt', header= None)
train_lab=pd.read_table('Data/trainClasses.txt', header=None, index_col=0)
test=pd.read_table('Data/testMatrixModified.txt', header=None)
test_lab=pd.read_table('Data/testClasses.txt', header=None, index_col=0)
terms = pd.read_table('Data/modifiedterms.txt', header=None)
```

In [4]:
```
train.head()
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	...	790	791	792	793	794	795	796	797	798	799
0	2.0	0.0	0.0	2.0	2.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4	8.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

5 rows × 800 columns

In [5]:
```
train.shape
```

Out[5]: (5500, 800)

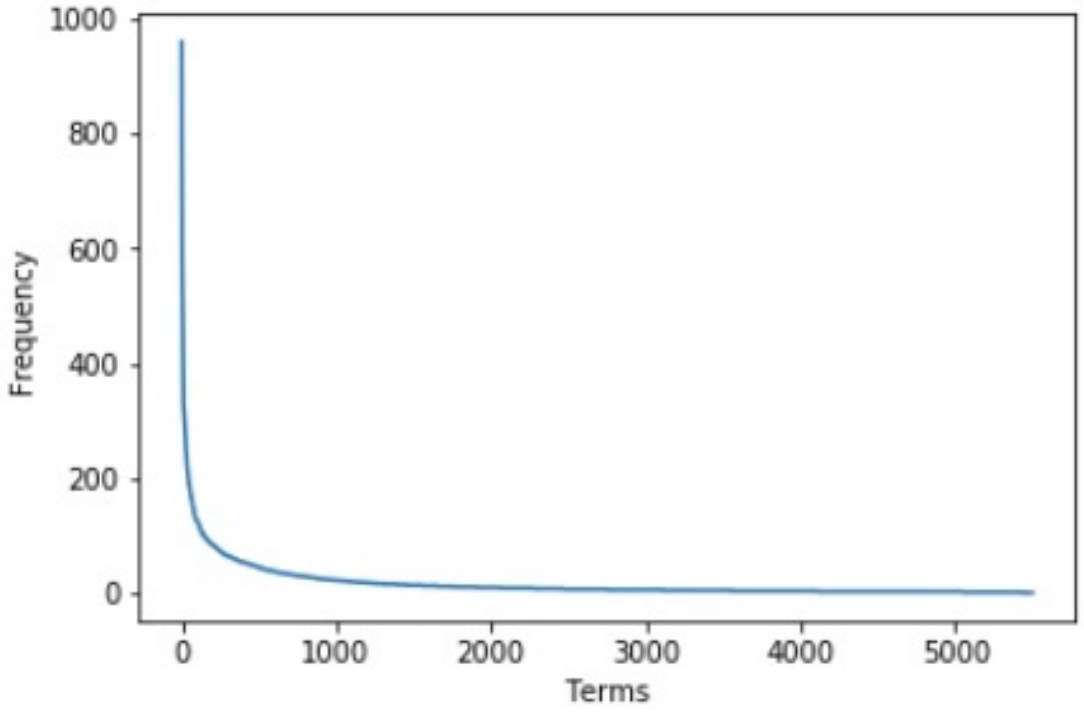
In [6]:
```
train_lab.shape
```

Out[6]: (800, 1)

In [7]:
```
#converting from TD to DT matrix
DT_train = train.T
DT_test = test.T
```

Zipf Distribution in Training Data

```
In [7]: plt.plot(sorted(termFreqs, reverse=True));
        plt.xlabel('Terms')
        plt.ylabel('Frequency')
        plt.show()
```



train_labels=pd.read_table('Data/trainClasses.txt', header=None, index_col=0)

```
In [97]: np.array(train_labels)

Out[97]: array([[0],
                [1],
                [0],
                [1],
                [0],
                [1],
                [1],
                [1],
                [1],
                [1],
                [1],
                [0],
                [0],
                [1],
                [1],
                [1],
                [0],
                [0],
                [1],
                [0],
```

train_labels=pd.read_table('Data/trainClasses.txt', header=None, index_col=0)

```
In [98]: np.array(train_labels).flatten()

Out[98]: array([0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1,
                0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0,
                0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1,
                1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0,
                1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1,
                1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1,
                0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0,
                1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0,
                1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,
                0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1,
                1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
                1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0,
                1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1,
                0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0,
                1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
                1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1,
                0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1,
                1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0,
                0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0,
                1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0,
                1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0,
                0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1,
                1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
                0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1,
                1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1,
                0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1,
                1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0,
                0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0,
                1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,
                0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
                0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1,
                0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,
                0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0,
                0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
                0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1,
                1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1,
                1, 0, 1, 0, 1, 1, 1, 1], dtype=int64)
```

See the Class Notebook: TF*IDF and Document Categorization

```python
def knn_search(x, D, K, measure):
    """ find K nearest neighbors of an instance x among the instances in D """
    if measure == 0:
        # euclidean distances from the other points
        dists = np.sqrt(((D - x)**2).sum(axis=1))
    elif measure == 1:
        # first find the vector norm for each instance in D, as well as for vector x
        D_norm = np.array([np.linalg.norm(D[i]) for i in range(len(D))])
        x_norm = np.linalg.norm(x)
        # Compute Cosine: dot product of x and each instance in D divided
        #                                              by the product of the two norms
        sims = np.dot(D,x)/(D_norm * x_norm)
        # The distance measure will be the inverse of Cosine similarity
        dists = 1 - sims
    idx = np.argsort(dists) # sorting
    # return the indexes of K nearest neighbors
    return idx[:K], dists

def knn_classify(x, D, K, labels, measure):
    from collections import Counter
    neigh_idx, distances = knn_search(x, D, K, measure)
    neigh_labels = labels[neigh_idx]
    count = Counter(neigh_labels)
    # print("Labels for top", K, "neighbors: ", count)
    predicted_label = count.most_common(1)[0][0]
    return neigh_idx, predicted_label
```

```
#preparation: transform data to matrix
DTM_train = np.array(DT_train)
DTM_test = np.array(DT_test)
train_lab_array = np.array(train_lab).flatten()
test_lab_array = np.array(test_lab).flatten()
```

```
# knn_function test of measure 1
top_K_neighbors, predicted_class = knn_classify(DTM_test[12], DTM_train, 10, train_lab_array, 1)
print ("The indices of K Nearest Neighbors are: {} \nThe predicted class is: {}".format(top_K_neighbors, predicted_class))
```

```
The indices of K Nearest Neighbors are: [334 177 233 715 732 510 221 178  94 254]
The predicted class is: 1
```

```
# knn_function test of measure 0
top_K_neighbors, predicted_class = knn_classify(DTM_test[12], DTM_train, 10, train_lab_array, 0)
print ("The indices of K Nearest Neighbors are: {} \nThe predicted class is: {}".format(top_K_neighbors, predicted_class))
```

```
The indices of K Nearest Neighbors are: [798 757 119 398  38 224 711 615 751  69]
The predicted class is: 0
```

Knn Classification & Evaluation

```
knn_search(x, DT_train, K, measure):

knn_classify(x, DT_train, K, train_labels, measure):

knn_evaluate(DT_test, test_labels, DT_train, train_labels, K, measure)
```

Note: **`knn_evaluate`** iterates through rows in DT_test matrix of test instances, and for each Instance, calls **`knn_classify`** to predict the label. It compares the predicted label for each test instance to the actual labels and then returns classification accuracy.

Knn Classification & Evaluation

```
Euclid=[]
Cosine=[]
```

```
%%time
for K in range(5, 100, 5):
    Euclid.append(knn_evaluate(DTM_test, test_lab_array, DTM_train, train_lab_array, K, 0))
```

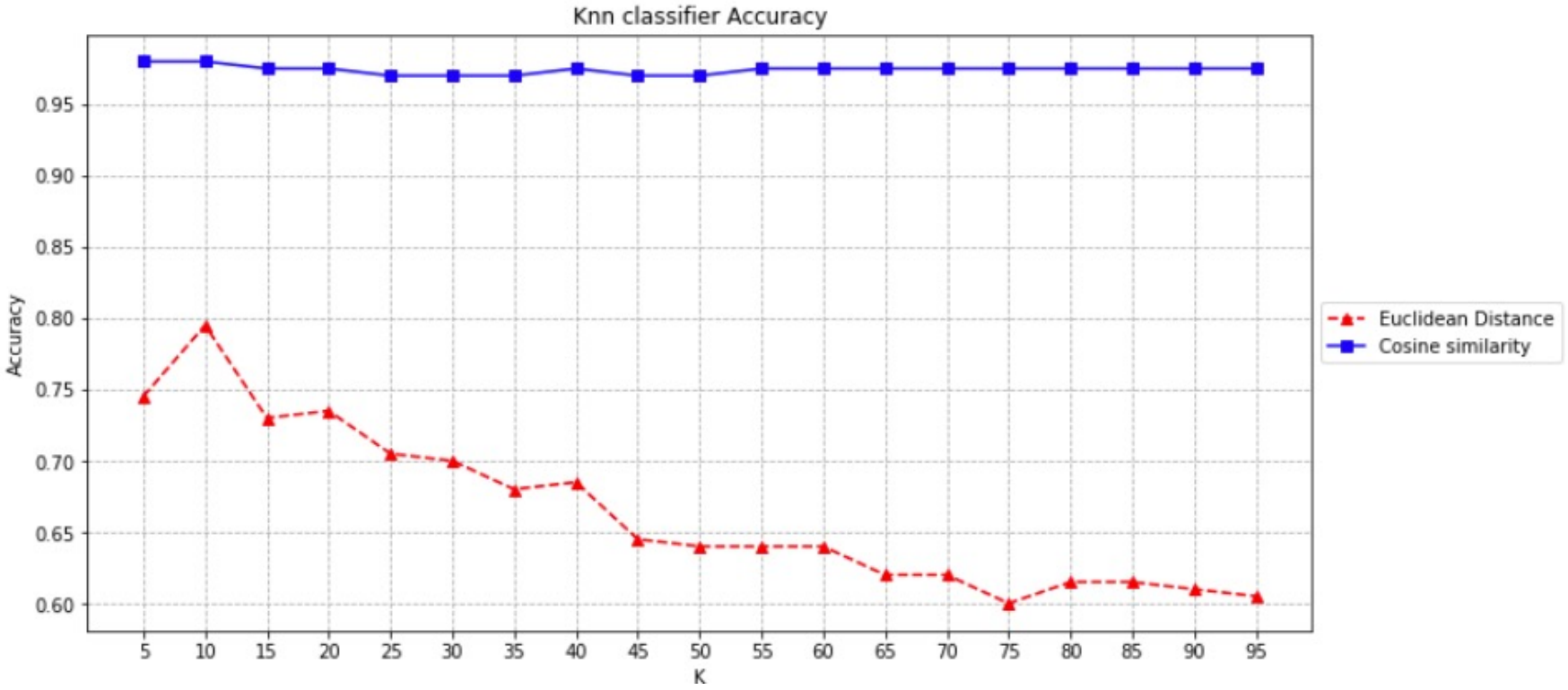
```
Wall time: 2min 36s
```

```
%%time
for K in range(5, 100, 5):
    Cosine.append(knn_evaluate(DTM_test, test_lab_array, DTM_train, train_lab_array, K, 1))
```

```
Wall time: 52 s
```

Knn Classification & Evaluation

```
knn_evaluate(DT_test, test_labels, DT_train, train_labels, K, measure)
```



If you get stuck…

- Other questions depend on your KNN function
- If you get stuck, compare Euclidean and Cosine with scikit-learn's function

Rocchio Text Categorization Algorithm (Training)

Assume the set of categories is $\{c_1, c_2, \ldots c_n\}$

For i from 1 to n let $\mathbf{p}_i = <0, 0, \ldots, 0>$ (*init. prototype vectors*)

For each training example $<x, c(x)> \in D$

Let $\mathbf{d}$ be the TF/IDF term vector for doc x

Let $i = j$ where $c_j = c(x)$

(*sum all the document vectors in c_i to get $\mathbf{p}_i$*)

Let $\mathbf{p}_i = \mathbf{p}_i + \mathbf{d}$

Rocchio Text Categorization Algorithm (Test)

Given test document x

Let $\mathbf{d}$ be the TF/IDF term vector for x
Let $m = -2$ (*init. maximum cosSim*)
For i from 1 to n:
    (*compute similarity to prototype vector*)
    Let $s = \text{cosSim}(\mathbf{d}, \mathbf{p}_i)$
    if $s > m$
        let $m = s$
        let $r = c_i$ (*update most similar class prototype*)
Return class r

Rocchio Classification

```
def Rocchio_Train(train, labels):
    . . .
    return prototype

def Rocchio_classifier(prototype, instance):
    . . .
    return predicted_label, sims

def rocchio_evaluate(test, test_lab, prototype):
    . . .
    return accuracy
```

Note: **`prototype`** could be a dictionary with unique class labels as keys and one dimensional arrays representing the prototype vectors for the class as values

tf x idf Transformation

The initial Term x Doc matrix (Inverted Index)

	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5	Doc 6
T1	0	2	4	0	1	0
T2	1	3	0	0	0	2
T3	0	1	0	2	0	0
T4	3	0	1	5	4	0
T5	0	4	0	0	0	1
T6	2	7	2	1	3	0
T7	1	0	0	5	5	1
T8	0	1	1	0	0	3

df	idf = log2(N/df)
3	1.00
3	1.00
2	1.58
4	0.58
2	1.58
5	0.26
4	0.58
3	1.00

Documents represented as vectors of words

tf x idf Term x Doc matrix

	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5	Doc 6
T1	0.00	2.00	4.00	0.00	1.00	0.00
T2	1.00	3.00	0.00	0.00	0.00	2.00
T3	0.00	1.58	0.00	3.17	0.00	0.00
T4	1.74	0.00	0.58	2.90	2.32	0.00
T5	0.00	6.34	0.00	0.00	0.00	1.58
T6	0.53	1.84	0.53	0.26	0.79	0.00
T7	0.58	0.00	0.00	2.92	2.92	0.58
T8	0.00	1.00	1.00	0.00	0.00	3.00