Assignment #2

In This assignment you are asked to read a data which include 48505 articles (Documents). Then fint the most similar documents using Locality Sensitive Hashing. Follow the lecture covering this topic step by step.

1. Data is available in Json format and you need to read it. 'https://www.ux.uis.no/~vsetty/data/assignment2_aricles.json' (5 points)

2. Shingle the documents (10 points)

Tips:

- Use string package to cleanup the articles e.g, str.maketrans('', '', string. punctuation)
- It is better to convert text to lower case that way you get fewer n-grams
- apply ngrams(x.split(), n) using ngrams from nltk on the content + title for computing n-grams, for this data n = 2 is suffcient
    - You can use n-gram at word level for this task
    - try with different n-gram values
    - You can use ngrams from nltk for this

3. Convert n-grams into binary vector representation for each document. You can do some optimzations if the matrix is too big. (10 points)

- For example,
    - Select top 10000 most frequent n-grams.
    - You may also try smaller values of n (like 2 or 3) which result in fewer n-grams.
    - Finally, you can also try sparse matrix representation. Like csr_matrix from scipy.sparse. It works even with full vocabulary.
        - Given a list of n-grams for each document, see how to builid a sparse matrix here https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

4. We need hash function that maps integers 0, 1, . . . , k − 1 to bucket numbers 0 through k − 1. It might be impossible to avoid collisions but as long as the collions are too many it won't matter much. (10 points)

- The simplest would be using the builtin hash() function, it can be for example, hash(rownumber) % Numberofbuckets
- You can generate several of these hash functions by xoring a random integer (hash(rownumber)^randint) % Numberofbuckets
- It can also be a as simple as (rownumber * randint) % Numberofbuckets

5. Compute minhash following the faster algorithm from the lecture (10 points)

6. Hash signature bands into buckets. Find a way to combine all the signature values in a band and hash them into a number of buckets ususally very high. (10 points)

- Easiest way is to add all the signature values in the bucket and use a similar hash function like before
- You should use the same hash function for all bands. And all documents ending up in same bucket for at least one band are considered as candidate pairs.

7. Tune parameters to make sure the threshold is appropriate. (10 points)

- plot the probability of two similar items falling in same bucket for different threshold values

8. Choose the best parameters and get nearest neighbors of each articles (20 points)

- Jaccard Similarity
- convert hash table into dictionary of article ids and its other articles that hashed in at least 1 same bucket

9. Write the nearest neibhors of each document to submissions.csv (comma separated, first column is the current document followed by a list of nearest neighbors) file and get the score (10 points)

10. Write a report + notebook + submission file in a zip file (5 points)

Imports used for the assignment

In [104…

```
import pandas as pd
import numpy as np
from nltk import ngrams
import string
import requests
from scipy.sparse import csr_matrix
import random
from collections import Counter
import math
import matplotlib.pyplot as plt
import copy
```

1. Read the data from the jsonfile, using requests worked fine for me

NOTE; Had to use less data to test it properly. Used the first 5000 articles.

In [51]:

```
json_url = "https://www.ux.uis.no/~vsetty/data/assignment2_aricles.json"
data = requests.get(json_url).json()
```

In [52]:

```
data = data[0:5000] # Only using 5000 artiles for performance
```

2. Shingle the documents, using all of the data

Using 2-grams here, meaning we get pairs of words.

In [53]:

```
def getFrequentNgrams(articles):
      shingled_articles = []
      for article in articles:
            article = article["Content"]

            n = 2 # K-shingling, using 2 for now as it says in the description

            # To lower case
            article = article.lower()

            # Remove punctuation and parentheses
            translator=str.maketrans('','',string.punctuation)
            article = article.translate(translator)

            # Replace newline with a space
            article = article.replace("\n", " ")

            # Split on the spaces, so that we can apply n-gram at word level
            word_list = article.split(" ")

            # Remove empty entries
            word_list = [word for word in word_list if word]

            # Create k-shingles for the list of words
            shingles = list(ngrams(word_list, n))

            # Add the SET(shingles)(no duplicates) to our list of all shingled articles
                  # Since we only check if it's there or not
            shingled_articles.append(set(shingles))

      return shingled_articles
```

In [54]:

```
# Shingles
shingles = getFrequentNgrams(data)
```

3. Turn the shingles into binary matrixes. For this I only use the 1000 most common shingles.

If we were not using the 1000 most common, we could just gather a set of all possible shingles and loop over these when we check if the documents contain the shingle or not. Had to reduce for performance

In [55]:

```
def getBinaryMatrix(docs):
    # Take each set of shingles, make it into a list and add it to a list that we can use to COUNT which are
    all_shingles = []
    for shingles in docs:
        all_shingles += list(shingles)

    # Use counter to find the most only the most common shingles across articles
    counter = Counter(all_shingles)
    most_common_shingles = counter.most_common(1000) # 1000 most common shingles

    unique_shingles = []
    for shingle in most_common_shingles:
        unique_shingles.append(shingle[0])

    # Now, unique shingles holds the 1000 MOST COMMON shingles across articles
        # Loop over these shingles, and check if they are in each article
            # Add 1 if they exist, 0 if they dont
    rows = len(unique_shingles)
    cols = len(docs)
    binary_matrix = csr_matrix((rows, cols), dtype=np.int8).toarray()

    for row, shingle in enumerate(unique_shingles):
        for col, article in enumerate(docs):
            if shingle in article:
                binary_matrix[row][col] = 1
            else:
                binary_matrix[row][col] = 0

    return binary_matrix
```

In [56]:

```
# Binary matrix
binmatrix = getBinaryMatrix(shingles)
```

4. Create hashfunctions using a simple random number to create a big hash

These are simply just permutations of a list. The value of their number does not matter as much, similar documents will get the same random value, which is the point.

In [57]:

```
def getHashFunctionValues(numrows, numhashfunctions):
    # Each hashfunction is
    hashFunctions = []
    for i in range(numhashfunctions):
        hashFunction = [] # Store the hash function permutations
        random_number = random.random() # Creates a random number between 0 and 1

        for j in range(numrows):
            hashFunction.append( hash( (j+1)*random_number ) % numrows ) # Since random number is a decimal,

        hashFunctions.append(hashFunction)

    return np.asarray(hashFunctions) # Convert to np.ndarray such it matches other matrixes (consistency)
```

In [58]:

```
# Hashfunction matrix
numHashFunctions = 100 # 100 is a clean number
hashmatrix = getHashFunctionValues(len(binmatrix), numHashFunctions)
```

5. Get Signature Matrixes using the optimal algorithm from the slides, using the hashfunctions I previously created.

Simply just looping over and replacing the current value, if the hash value is smaller than what is already there. There has been no mentions of the case if we do NOT find a match though (given that we use common shingles and not ALL shingles), but in this case, the signature will be left with a number in the columns that is +1 the total rows, meaning it should not make sense really.

In [59]:

```
# Implemented based on the optimized algorithm
def getMinHashSignatureMatrix(binary_matrix, hash_val_matrix):

    signature_matrix = csr_matrix((len(hash_val_matrix), len(binary_matrix[0])), dtype=np.int64).toarray()
    signature_matrix[:] = len(binary_matrix)+1 # Did not see any measures to account for this, slide says INF
    # print("Cap: " + str(len(binary_matrix)+1)) # Perhaps not compare CAPS in the future?

    for row, docs in enumerate(binary_matrix):
        for col, num in enumerate(docs):
            if num == 1:
                for idx, hashfunction in enumerate(hash_val_matrix):
                    if hashfunction[row] < signature_matrix[idx][col]:
                        signature_matrix[idx][col] = hashfunction[row]

    return signature_matrix
```

In [ ]:

```
# Signature matrix
signmatrix = getMinHashSignatureMatrix(binmatrix, hashmatrix)
```

6. Hash the signatures into LSH buckets, where we store their document id, such that similar documents are placed together.

Bands and Rows plays a role here by; If we use fewer rows, like 1, we will probably have more matches, but they might not be similar, as we are comparing documents to each other based on a SINGLE shingle. If we set rows to be 2, we now need a PAIR of shingle in the bands to match, which means fewer matches, but documents are probably more similar. etc.

In [92]:

```
def getLSH(signature_matrix, num_bands, num_buckets):

    # Get rows from num_bands
    num_rows = math.floor(len(signature_matrix) / num_bands)

    # First, seperate the signature matrix in to bands
    bands = []
    for i in range(0, len(signature_matrix), num_rows):
        bands.append(signature_matrix[i:i+num_rows])

    # Secondly, take each band, hash the columns towards a bucket(unique per band), store the DOCUMENT index
        # This way, we can see which documents we should compare, as we found similarities
    buckets = [] # Store all bucket dictionaries for all bands
    for band in bands:
        bucket = {} # Using dictionary instead of list, since not all indexes are used perhaps

        columnValues = band.transpose()
        for docIndex, column in enumerate(columnValues):

            # Using the same hash function as before, but since str() gives it a BIG number, the random number
            hashValue = ( hash(str(column)) % num_buckets) # Using string() convertion to be able to hash the

            if hashValue in bucket.keys():
                bucket[hashValue].append(docIndex)
            else:
                bucket[hashValue] = []
                bucket[hashValue].append(docIndex)

        buckets.append(bucket)

    return buckets # list of buckets
```

In [82]:

```
# We try with 10 bands
num_bands = 10
num_buckets = 1000
lsh_buckets = getLSH(signmatrix, num_bands, num_buckets)
```

```
10
```

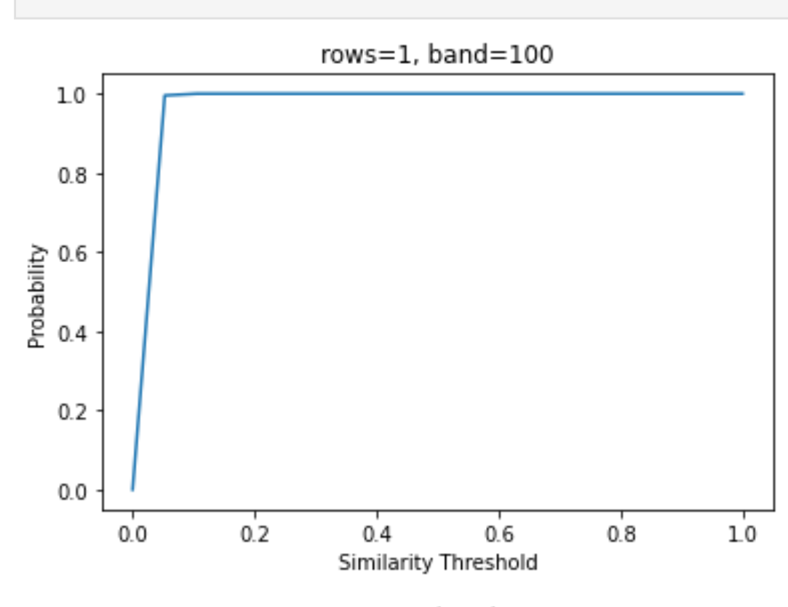
7. Plot the similarity x probability for 100 hashfunctions, like in the assignment. Later, I set the threshold to 0.6, and for that threshold rows=5 and bands=20 is preferred, as we can see the peak around 0.6 for that plot.

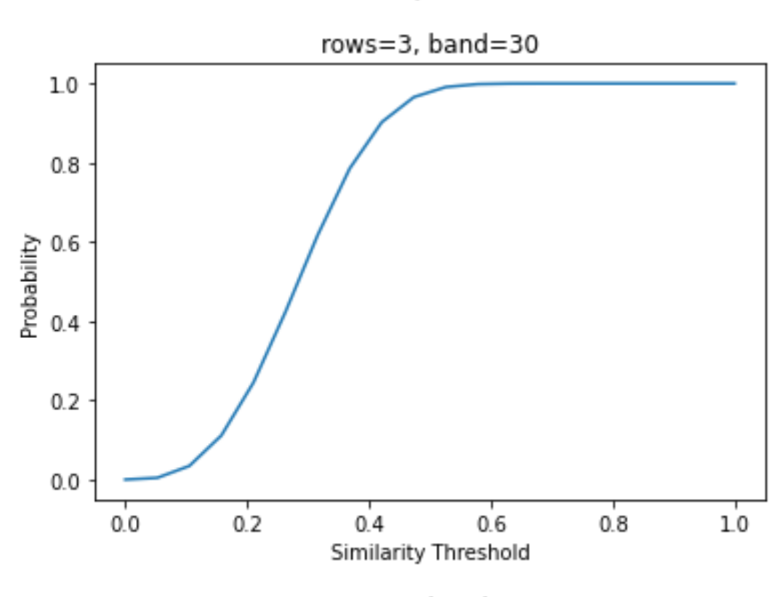
In [87]:

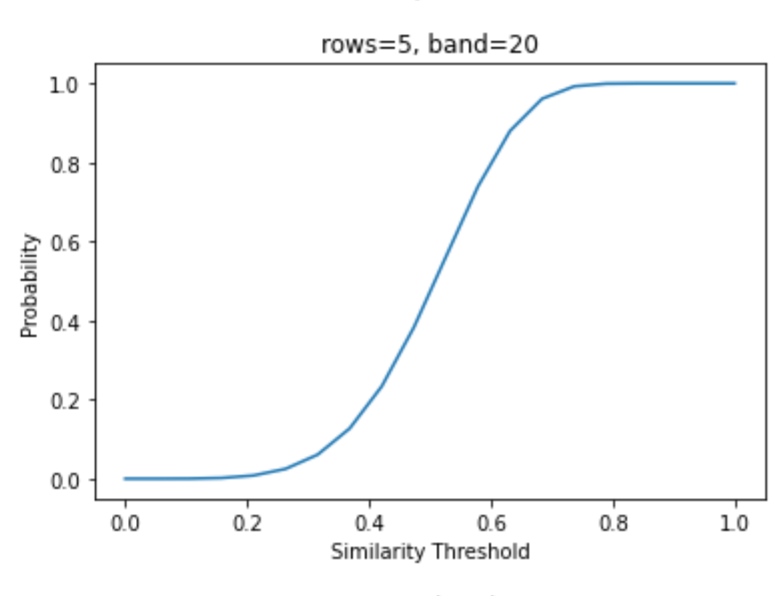
```
def plotProbability(s, b, r):
    x_values = s
    y_values = []
    for threshold in s:
        y_values.append( 1  - (1-threshold**r)**b )

    plt.plot(x_values, y_values)
    plt.xlabel("Similarity Threshold")
    plt.ylabel("Probability")
    plt.title("rows="+str(r)+", band="+str(b))
```

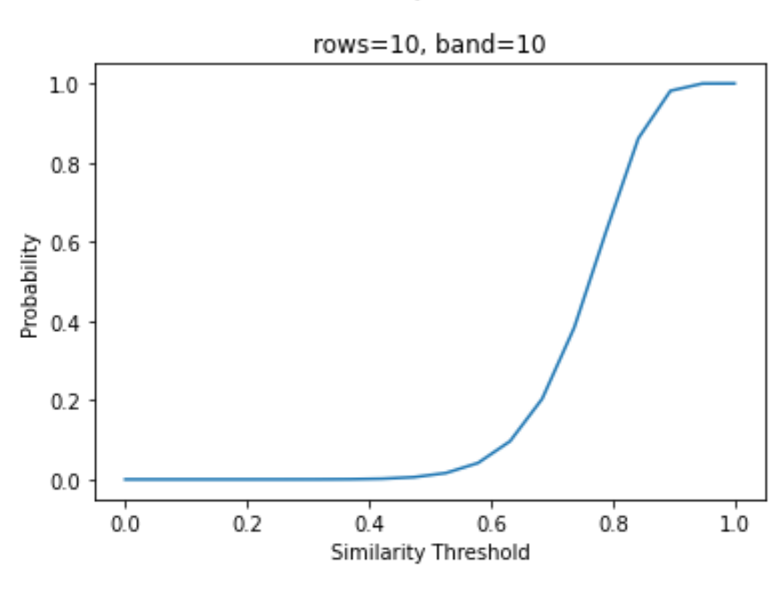
In [88]:

```
# Example given 100 hash functions
s = np.linspace(0, 1, 20)
r = [1, 3, 5, 10]
b = [100, 30, 20, 10]

for i in range(len(b)):
    plt.figure()
    plotProbability(s, b[i], r[i])

plt.show()
```









8. Tuning based on the plot, then finding neighbors

For Jaccard, the similarity between signature is the fraction of times they agree. Nearest neighbors consists of a key which corresponds to the article id, and the value becomes a list of all documents that is similar to that article.

In [89]:

```
def getJaccardSimilarityScore(C1, C2):
    # When using it for C1 and C2 for SIGNATURE MATRIXES
        # The Jaccard Similarity score is the fraction of times the hash functions agree
    rows = len(C1)
    similarities = 0
    for i in range(rows):
        if C1[i] == C2[i]:
            similarities += 1

    return (similarities / rows)
```

In [98]:

```
def getNearestNeighbors(all_buckets):
    # convert hash table into dictionary of article ids and its other articles that hashed in at least 1 same
    nearest_neighbors = {}

    # Loop over all buckets to check the articles inside
    for i, bucket in enumerate(all_buckets):
        for key in bucket.keys():
            if len(bucket[key]) > 1: # Only care about buckets that have several articles in them
                for article_id in bucket[key]:

                    # Similar to the lsh buckets
                    if article_id in nearest_neighbors.keys():
                        for doc_id in bucket[key]: # Looping over the same list again to add everything but my
                            if doc_id != article_id:
                                nearest_neighbors[article_id].add(doc_id)
                    else:
                        nearest_neighbors[article_id] = set() # Using a set here, as the same article might be
                        for doc_id in bucket[key]:
                            if doc_id != article_id:
                                nearest_neighbors[article_id].add(doc_id)

    return nearest_neighbors
```

In [99]:

```
# Change up the bands to correspond to the optimal 20 bands, increase buckets aswell
num_bands = 20
num_buckets = 10000 # Not sure if high enough, but higher = less collisions
lsh_buckets = getLSH(signmatrix, num_bands, num_buckets)

# Find the lsh neighbors
lsh_neighbors = getNearestNeighbors(lsh_buckets)
```

In [102…

```
# Example of a neighbor for an article
article_id_test = 0
lsh_neighbors[article_id_test]

# Can see that there are some similarity between 0 - 39, 0 - 77 etc
```

Out[102]:

```
{39, 77, 499, 779, 1360, 2193, 3421, 4207, 4558}
```

9. Write the nearest neighbors into a dataframe

First checking if the similarity is above the set threshold that we have. Then placed in lists, where the indexes(matches) should correspond across lists. When returned, I simply returned a list of those lists.

In [110…

```
def removeUnsimilarNeighbors(threshold, neighbors, signatures):

    n_copy = copy.deepcopy(neighbors)

    submission_id = []
    submission_nid = []

    for article_id, neighbor_ids in n_copy.items():
        for nid in neighbor_ids:
            C1 = signatures.transpose()[article_id]
            C2 = signatures.transpose()[nid]
            score = getJaccardSimilarityScore(C1, C2)

            if score >= threshold:
                submission_id.append(article_id)
                submission_nid.append(nid)

    return [submission_id, submission_nid]
```

In [111…

```
# Prepare them to be added to a dataframe
similarity_threshold = 0.6 # Chosen by me

# Remove neighbors below similarity score
lists = removeUnsimilarNeighbors(similarity_threshold, lsh_neighbors, signmatrix)
```

In [112…

```
data = pd.DataFrame()
data['article_id'] = lists[0]
data['neighbor_id'] = lists[1]
data.sort_values(by=['article_id', 'neighbor_id'], inplace=True)
data.head(100)

# Can see that article 9 and article 825 have similarity above 60% threshold
```

Out[112]:

	article_id	neighbor_id
20559	9	825
20560	9	3359
11	11	51
13	11	437
17	11	714
...	...	...
32	51	3856
46	51	4175
34	51	4241
42	51	4284
31	51	4367

100 rows × 2 columns

In [116…

```
data.to_csv("submissions.csv", index=False)
```

Smaller sample(unrelated to the articles), if you want to run the ipynb yourself

In [114…

```
# Easy similarity text example ( C3 and C4 are very similar )
demo_similar_matrix =    [[0, 0, 1, 1],
                         [0, 1, 0, 0],
                         [0, 1, 1, 1],
                         [0, 0, 1, 1],
                         [1, 0, 0, 0],
                         [0, 1, 0, 1],
                         [1, 0, 0, 0],
                         [0, 1, 0, 0],
                         [1, 0, 1, 1],
                         [0, 1, 0, 0]]

# All possible unique shingles accross all documents
demo_binmatrix = np.asarray(demo_similar_matrix)
print("Binary Matrix")
print(demo_binmatrix)
print("\n")

# Hash matrix
demo_hashmatrix = getHashFunctionValues(len(demo_binmatrix), 4) # (numrows, numhashfunctions)
print("Hash function Matrix")
print(demo_hashmatrix.transpose()) # Transposed to represent it like in the lecture
print("\n")

# Signature matrix
demo_signmatrix = getMinHashSignatureMatrix(demo_binmatrix, demo_hashmatrix)
print("Signature Matrix")
print(demo_signmatrix)
print("\n")

# LSH buckets
demo_lsh_buckets = getLSH(demo_signmatrix, 2, 1000)
print("LSH Buckets")
print(demo_lsh_buckets)
print("\n")

# Find the lsh neighbors
demo_lsh_neighbors = getNearestNeighbors(demo_lsh_buckets)
print("Article Neighbors")
print(demo_lsh_neighbors)
print("\n")

# Prepare them to be added to a dataframe
demo_similarity_threshold = 0.6 # Chosen by me
# Remove neighbors below similarity score
demo_lists = removeUnsimilarNeighbors(demo_similarity_threshold, demo_lsh_neighbors, demo_signmatrix)
print("Similar Article Neighbors")
print(demo_lists)
print("\n")

demo_data = pd.DataFrame()
demo_data['article_id'] = demo_lists[0]
demo_data['neighbor_id'] = demo_lists[1]
demo_data.sort_values(by=['article_id', 'neighbor_id'], inplace=True)
demo_data.head(100)
```

```
Binary Matrix
[[0 0 1 1]
 [0 1 0 0]
 [0 1 1 1]
 [0 0 1 1]
 [1 0 0 0]
 [0 1 0 1]
 [1 0 0 0]
 [0 1 0 0]
 [1 0 1 1]
 [0 1 0 0]]


Hash function Matrix
[[0 2 8 0]
 [0 4 5 9]
 [9 5 8 4]
 [9 7 9 8]
 [9 6 0 1]
 [8 0 5 8]
 [8 4 2 8]
 [8 3 8 5]
 [7 3 3 2]
 [7 2 0 2]]


[[11 11 11 11]
 [11 11 11 11]
 [11 11 11 11]
 [11 11 11 11]]
Signature Matrix
[[7 0 0 0]
 [3 0 2 0]
 [0 0 3 3]
 [1 2 0 0]]


LSH Buckets
[{856: [0], 795: [1, 3], 460: [2]}, {885: [0], 460: [1], 554: [2, 3]}]


Article Neighbors
{1: {3}, 3: {1, 2}, 2: {3}}


Similar Article Neighbors
[[3, 2], [2, 3]]
```

Out[114]:

	article_id	neighbor_id
1	2	3
0	3	2