Introduction.

We want to find movies that are similar, we can find this out with pairs of people that have seen a lot of the same movies. This is a nearest neighbours problem that is very common among daily data science tasks. There are numerous applications like finding duplicates or similar documents, audio/video search etc . A naive approach that uses brute force to check for all the possible combinations will give an exact nearest neighbour solution but it's not scalable. Approximation algorithms can accomplish this task. Although these algorithms don't guarantee to give you the exact answer they'll be provide a good approximation. These algorithms are faster and scalable. Min Hashing and Locality sensitive hashing (LSH) are two of them.

Data preprocessing.

We define the relation $watched(X, Y)$ as "User X has watched the movie Y". We will create an adjacency matrix according to this. Our data input are comma separated files with the following fields: $userId, movieId, rating, timestamp$. Inside the $load.py$ file the function $load\_movies\_to\_dicts(path)$ scans the file once and returns three dictionaries.

```
userList = {}
movieMap = {}
movieList = {}
movie_counter = 1

with open(path) as infile:
    csv_reader = csv.reader(infile, delimiter=',')
    count = 0

    for row in csv_reader:
        if count == 0:
            count += 1
            continue

        userId_tmp = int(row[0])
        movieId_tmp = int(row[1])
```

Declaring variables, iterating through the csv files, skipping headers and getting the information we want.

- The $userList$, that every row contains a user and their value contains all the movies the user has seen.

```
if userId_tmp not in userList.keys():
    userList[userId_tmp] = []

userList[userId_tmp].append(movieId_tmp)
```

- The $movieMap$, that assigns a unique id to every movie.

- The $movieList$, that every row contains a movie and their value contains all the users that have seen this movie.

```
if movieId_tmp not in movieList.keys():
    movieList[movieId_tmp] = []
    movieMap[movieId_tmp] = movie_counter
    movie_counter += 1

movieList[movieId_tmp].append(userId_tmp)
```

Minimum hashing.

The idea of hashing is to convert each document to a small signature using a hashing function. If two documents have high similarity then the probability that their hash functions are the same is high. The same applies for the low similarity of two documents. The minHash algorithm is implemented in the $minhash.py$ file with the $minHash(sig\_count, userList, movieMap, movieList)$ function.

At first we create a random permutation of the shingle matrix with the given working functions.

```
user_count = len(userList.keys())
movie_count = len(movieList.keys())
permutations = create_random_permutations(user_count, sig_count)
SIG = {}
```

Then we initialize the matrix of signatures.

```
for i in range(1, movie_count + 1):
    SIG[i] = []
    for j in range(sig_count):
        SIG[i].append(user_count + 10)
```

The hash function is the index of the first row in which the column has value 1. After doing this several times with different permutations we can create the signature matrix.

```
    # Scan users
    for user in userList.keys():
        # Scan movies
        for movie in movieList.keys():
            # Check if the user has seen this movie
            if user in movieList[movie]:
                # Scan permutations
                for i in range(sig_count):
                    # Map movieIndex in permutationIndex
                    permutation_index = permutations[i].index(user)

                    if permutation_index < SIG[movieMap[movie]][i]:
                        SIG[movieMap[movie]][i] = permutation_index

    return SIG
```

We know that the similarity of the signatures is the fraction of the min-hash functions in which they agree. This is implemented in the $minhash.py$ file with the $signatureSimilarity(movieId1, movieId2, n, movieMap, SIG)$ function.

```
index1 = movieMap[movieId1]
index2 = movieMap[movieId2]

count = 0
for i in range(n):
    if SIG[index1][i] == SIG[index2][i]:
        count += 1

return count / n
```

Evaluation

We test the algorithm inside the $test\_minhash.py$ function. Bellow we can see the imports and declarations we need.

```
 precision = true_positives / (true_positives + false_positives)
 recall = true_positives / (true_positives + false_negatives)
 f1 = 2 * recall * precision / (recall + precision)
```

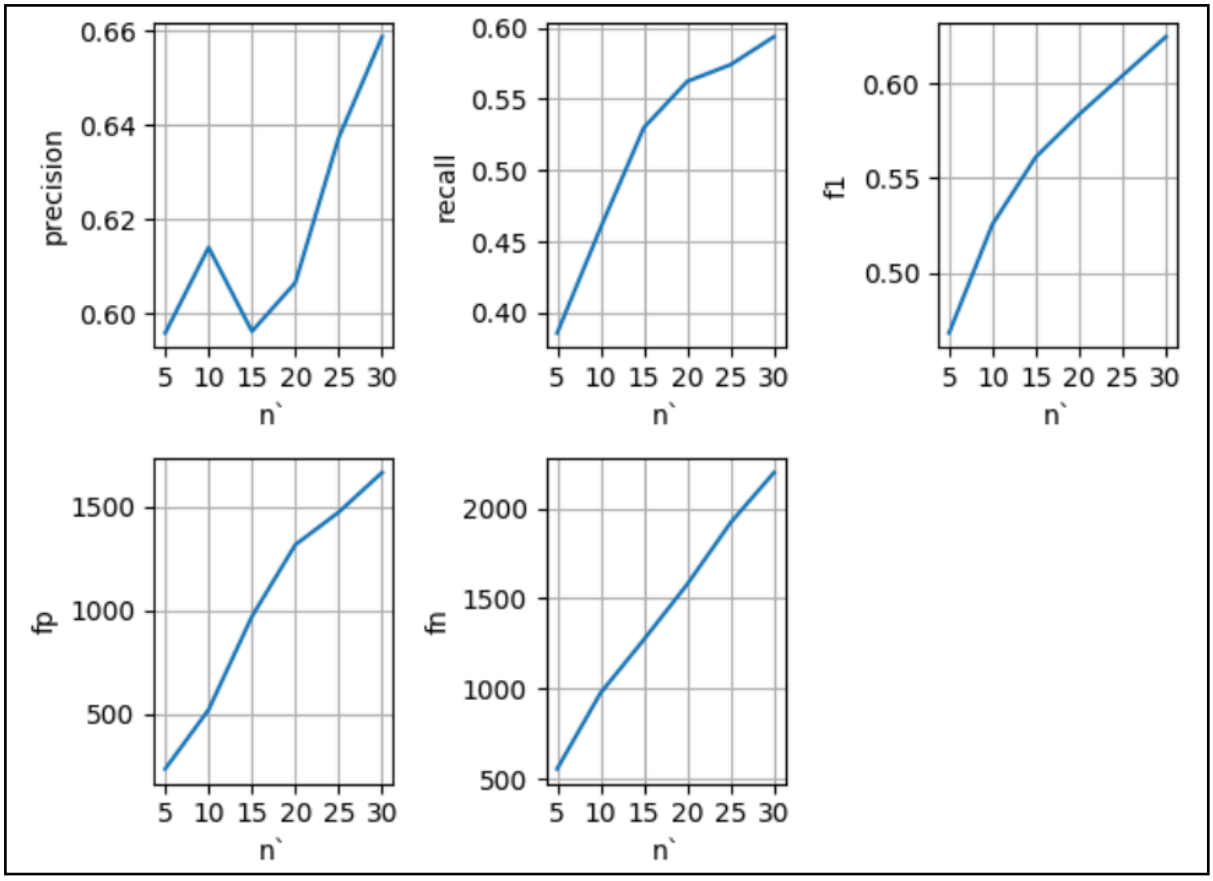
```
from load import load_movies_to_dicts
import matplotlib.pyplot as plt


path = r"/Users/leonidas/Desktop/Algorithms for big data/assignment1/ratings.csv"
userList, movieMap, movieList = load_movies_to_dicts(path)
movie_count = len(movieList.keys())
user_count = len(userList.keys())
```

The function $get\_labels(s, movie\_limit)$ finds the pairs that have a Jaccard similarity of at least s.

```
def get_labels(s, movie_limit):
    labels = []
    for i in range(movie_limit):
        for j in range(movie_limit):
            if j > i:
                m1 = list(movieList)[i]
                m2 = list(movieList)[j]
                js = get_J_Similarity(movieList[m1], movieList[m2])

                if js >= s:
                    labels.append([m1, m2, True])
                else:
                    labels.append([m1, m2, False])
    return labels
```

Now we will count the signature similarity given by the signatureSimilarity function for the first 20 pairs in the series of films taking into account the $n' = 5,10,15,20,25,30$ first lines of SIG. For every value of n' we will compute the precision according to the number of false positive results.

```
 SIG = minHash(sig_num, userList, movieMap, movieList)

 scores = {}
 for n in range(5,31,5): # Testing values of n first signatures
     for i in labels:
         sim = signatureSimilarity(i[0], i[1], n, movieMap, SIG)

         if sim >= 0.25:
             if i[2]:
                 true_positives += 1
             else:
                 false_positives += 1
         else:
             if i[2]:
                 false_negatives += 1
             else:
                 true_negatives += 1
```



Locality sensitive hashing

LSH refers to a family of functions to hash data points into buckets so that data points near each other are located in the same buckets with high probability, while data points far from each other are likely to be in different buckets. This makes it easier to identify observations with various degrees of similarity. The general idea of LSH is to find an algorithm such that if we input signatures of 2 documents, it tells us that those 2 documents form a candidate pair or not. Thats why we need to hash the columns of the signature matrix using several hash functions. If 2 documents hash into same bucket for at least one of the hash functions we can consider the 2 documents as a candidate pair. We will divide the signature matrix into b bands, each band having r rows. For each band, hash its portion of each column to a hash table with k buckets. Pairs are those that hash to the same bucket for at least 1 band. The LSH algorithm is implement in the $lsh.py$ file.

```
def local_sensitive_hashing(signatures,sig_num, b, movieList, movie_limit):
    r = sig_num // b
    hf = create_random_hash_function()
    lsh_sig = prepare_signatures(signatures, r, movieList)
    pairs = []

    for band in range(b):
        buckets = {}
        for i, movie in enumerate(lsh_sig):
            if i == movie_limit - 1:  # STOPS AT 20 FIRST MOVIES
                break
            tmp = hf(int(lsh_sig[movie][band]))
            if tmp not in buckets.keys():
                buckets[tmp] = []
            buckets[tmp].append(movie)

        for key in buckets:
            if len(buckets[key]) > 1:
                pairs = updatePairs(buckets[key], pairs)

    return pairs
```

Evaluation

We test the algorithm inside the $test\_lsh.py$ file. Bellow we can see the imports and declarations we need.

```
from jaccardSimilarity import get_J_Similarity
from minhash import minHash, signatureSimilarity
from lsh import local_sensitive_hashing
from load import load_movies_to_dicts
import matplotlib.pyplot as plt

path = r"/Users/leonidas/Desktop/Algorithms for big data/assignment1/ratings.csv"
userList, movieMap, movieList = load_movies_to_dicts(path)
movie_count = len(movieList.keys())
user_count = len(userList.keys())
```

The $get\_labels(s, movie\_limit)$ is exactly the same with the min hashing implementation.

Then the $get\_lsh\_scores(labels, sig\_num, movie\_limit)$ computes the scores we need. At first the function returns the signature matrix with the minimum hashing algorithm.

```
def get_lsh_scores(labels, sig_num, movie_limit):
    true_positives = 0
    true_negatives = 0
    false_positives = 0
    false_negatives = 0

    SIG = minHash(sig_num, userList, movieMap, movieList)
```

```
scores = {}
for band_num in [3,5,6,10]:# Testing values of band numbers
    pairs = local_sensitive_hashing(SIG, sig_num, band_num, movieList, movie_limit)
    for i in labels:
        tmp = (i[0], i[1])
        if tmp in pairs:
            if i[2]:
                true_positives += 1
            else:
                false_positives += 1
        else:
            if i[2]:
                false_negatives += 1
            else:
                true_negatives += 1

    precision = true_positives / (true_positives + false_positives)
    recall = true_positives / (true_positives + false_negatives)
    f1 = 2 * recall * precision / (recall + precision)
```

Bellow we can see the results. If the band number is large that means we have to deal with more hash functions. This means that we are increasing the probability of finding a candidate pair and the is similar to taking a small threshold. That's why the precision falls as the number of bands gets bigger.