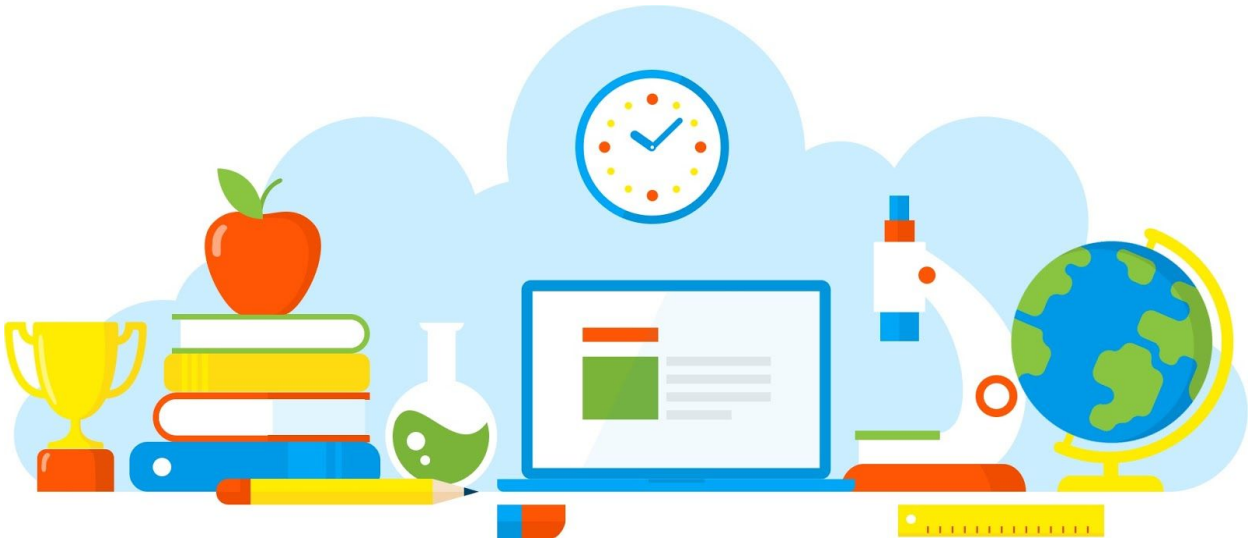

Big Data HW2 Report

Teacher: Dr. Haratizadeh

Mahdi Sadeghi
830598035
2019/11/2

Finding Similar Documents using Minhash and LSHF

Algorithm

For finding similar documents in a document set we can compute Jaccard distance between all possible pairs but this is unfeasible in large dataset. One way to overcome this problem is using Minhash Signatures and LSH functions to lower count of candidate document pairs. In this approach we first transform each document to set of shingles with the length of k. After that we generate matrix with shingles as rows and documents as columns. If a shingle is present i a document, the corresponding element of the matrix is set to 1, otherwise 0. Then we should compute minhash signatures on each document by first generating different permutations of the shingle matrix and then finding row number of the first element with non-zero value. It can be shown that the similarity of two minhash signatures is equal to the Jaccard similarity of original documents. After obtaining the minhash signatures we can select candidate pairs by hashing the signature part by part using LSH functions. Any pair with the same LSH hash in at least one part will be a candidate. At last we can compute the similarity between candidates and select the ones with similarity over the threshold.

Implementation

1. Loading Documents

first we load all the documents to an array of strings:

```
def load_documents():
    """
    loads documents and returns them as a list
    :return: list of documents
    """
    documents = []
    for i in range(1, DOCUMENTS_COUNT + 1):
        file_path = os.path.join(
            os.path.abspath(os.path.dirname(__file__)), '../data/{0}'.format(i)
        )
        with open(file_path, 'r') as fp:
            document = fp.read()
            documents.append(document)

    return documents
```

2. Preprocess

each document is sent to be preprocessed. in this stage we remove all the spaces, new lines and tab characters. we also convert all the texts to upper case. this is due to the fact that digits and uppercase alphabet ascii codes come back to back, so the calculations will be easier for us in the next stages:

```
def preprocess_document(document):
    """
    this function will remove redundant blanks, and some other characters from
document
    :param document:
    :return: a processed document as string
    """
    processed_document = ''

    # string concatenation is an expensive operation so instead of
concatenating character by character, we concatenate
    # bulk of document from the last known invalid character to the current
known invalid character
    window_start = 0
    window_end = 0
    for c in document:
        if c in [' ', ',', ':', ';', '.', '\n', '\t']:
            # an invalid character found, we add the content before it and
shift the window
            processed_document += document[window_start:window_end].upper()
            window_start = window_end = window_end + 1
            continue

        window_end += 1

    # add the remaining document to processed document
    processed_document += document[window_start:window_end].upper()
    return processed_document
```

3. Convert Document to Shingles

We have used the concept of window in generating shingles for each document. Easily we move a window with length of k from the beginning of the document to the end and save the corresponding text as a shingle. K is 5 in this implementation as it is not too small so that most of the documents be similar and it's also not too big so that the results suffer from overfitting:

```
def generate_shingles(k, processed_document):
    """
    generates k-shingles from a processed document
    :param processed_document:
    :param k:
    :return:
    """
    shingles = []

    # we will use a window with length of k, so that the content between window
start and window end will be a shingle
    # we will shift this window to the right 1 by 1
    for window_end in range(k, len(processed_document) + 1):
        window_start = window_end - k
        # the shingle is generated by cutting document from window_start to
window_end
        shingles.append(processed_document[window_start:window_end])

    return shingles
```

4. Hashing Shingles

By hashing shingles we mean transforming the shingle string to a unique integer that allows us to position it in the document/shingle matrix. The formula is very similar (actually it's exactly the same) to converting numbers of an arbitrary base to 10 base number:

```
def hash_shingle(k, shingle):
    """
    in this function we transform input k-shingle to a unique integer.
    for example with k=3, "aba" = 0 + 30 + 0.
    """
    hashed_value = 0

    i = k - 1
    for c in shingle:
        # as 'A' ascii code is 49 we subtract it from each characters ascii
code to get zero based values
        c_code = ord(c) - 49
        hashed_value += (ALPHABET_SIZE ** i) * c_code
        i -= 1

    return hashed_value
```

5. Shingle/Document Matrix

In saving the document/shingle matrix we save the shingles present in a document in one row with their hash values. We do this because saving the full matrix requires large amounts of ram and is not required due to the fact that the matrix is largely sparse:

```
def generate_shingle_matrix(k, all_shingles):
    """
    generates a matrix that indicates which document has which shingles
    :param all_shingles: list of list of shingles
    :return: matrix
    """

    # we will be saving the matrix as sparse, so that each row represents one
documents shingles
    # the shingles in the matrix are hashed before being saved to take less
memory
    matrix = []
    for shingles in all_shingles:
        # we create a row for each document in the matrix
        row = []
        for shingle in shingles:
            row.append(hash_shingle(k, shingle))
        matrix.append(row)
    return matrix
```

6. Matrix Permutation

Because permuting the matrix by moving rows is heavy, instead we define a function which maps the position of old matrix to position in the new matrix. I have implemented a function which generates different versions of these functions:

```
def generate_permutation_function(k, seed):
    """
    this function returns a hash function which virtually creates a permutation
of a matrix
    :param seed: it should be higher than zero
    :return: hash function
    """
    mod_argument = (ALPHABET_SIZE ** k) + 1

    def permutation_function(position):
        # this function works with following formula: new_position = s*old
position mod number of rows
        return (seed * position) % mod_argument

    return permutation_function
```

6. Minhash Signature and Matrix

The algorithm used to compute minhash signature here is that first we obtain the permutation function, then we set the minhash to ∞ and find the minimum value of permuted index by passing the hashed shingles in document to permutation function. We do this N number of times to generate a signature of size N. by increasing the size of signature, we can increase accuracy. In here we chose 50 as the size:

```
def minhash(document_column, permutation_function):
    # at first we set minhash_value as infinity, because we want to find the
minimum value in the list
    minhash_value = math.inf
    # as we hashed shingles, each value in document_column shows both shingle
value and position with one int
    for position in document_column:
        permuted_position = permutation_function(position)
        if permuted_position < minhash_value:
            minhash_value = permuted_position

    return minhash_value


def generate_minhash_signature(k, document_column, rounds):
    """
    generates and returns minhash signature for a specific document, the length
of the signature is
    equal to the input rounds argument
    """
    signature = []
    # we generate a permutation function for each round and compute the minhash
with that specific permutation
    for i in range(rounds):
        permutation_function = generate_permutation_function(k, i + 1)
        minhash_value = minhash(document_column, permutation_function)
        signature.append(minhash_value)

    return signature
```

7. Finding Similar Candidates using LSH Functions

Here we divide minhash of all the documents to x bands. Then we start hashing these bands and if two documents have a band with similar hash, they will be saved as candidates(Leskovec, Rajaraman, and Ullman 2014):

```
def generate_lsh_function(seed):
    band_start = seed * BAND_SIZE
    band_end = band_start + BAND_SIZE

    def lsh_function(signature):
        val = ''
        for i in range(band_start, band_end):
            val += ',' + str(signature[i])

        return hash(val)

    return lsh_function


def generate_candidate_pairs(minhash_matrix):
    candidate_pairs = set()

    bands_count = int(SIGNATURE_SIZE / BAND_SIZE)
    for i in range(bands_count):
        # documents with similar lsh value will go into the same key of this
dictionary
        locality_array = {}
        lsh_function = generate_lsh_function(i)
        # call lsh_function for all signatures of minhash_matrix
        for dn, signature in enumerate(minhash_matrix):
            h = lsh_function(signature)
            if h in locality_array:
                locality_array[h].append(dn)
            else:
                locality_array[h] = [dn]
        # find keys with more than 1 item in it to generate candidate pair
        for key in locality_array:
            for j in range(len(locality_array[key])):
                for k in range(j + 1, len(locality_array[key])):
                    candidate_pairs.add(
                        (locality_array[key][j], locality_array[key][k]))

    return candidate_pairs
```

8. Compute Similarity between Two Candidates

Similarity is computed by comparing the minhash signatures of candidate pairs. If the similarity is higher than a certain threshold, we print them to the output :

```
def get_candidate_pairs_similarity(minhash_matrix, candidate_pair):
    """
    computes the similarity of two documents using their minhash signatures
    """
    found = 0
    for minhash in minhash_matrix[candidate_pair[0]]:
        if minhash in minhash_matrix[candidate_pair[1]]:
            found += 1

    return found / SIGNATURE_SIZE

for pair in candidate_pairs:
    similarity = get_candidate_pairs_similarity(minhash_matrix, pair)
    # only show items with similarity higher than the threshold
    if similarity >= SIMILARITY_THRESHOLD:
        print('{0}: {1}'.format(pair,
                    get_candidate_pairs_similarity(minhash_matrix, pair)))
```

Bibliography

Leskovec, Jure, Anand Rajaraman, and Jeffrey David Ullman. 2014. “Mining of Massive Datasets.” https://doi.org/10.1017/cbo9781139924801.