homework1

November 14, 2023

1 Homework 1: Locally Sensitive Hashing

Locality Sensitive Hashing (LSH) is a technique used in computer science to solve the approximate or exact Near Neighbor Search in high-dimensional spaces. It is used to find similar items in a large dataset by hashing input items so that similar items map to the same "buckets" with high probability. LSH is commonly used in recommendation systems, image and audio recognition, and data mining.

In this particular notebook we will implement a simplified version of the LSH algorithm for to compare texts and find how similar are. We will implement 4 classes which will help us to compute how similar 2 texts are. Those classes are: Shingling, CompareSets, MinHasing and CompareSignatures.

1.1 Dataset

As a dataset, we have used the following texts: - Lorem Ipsum with 5 paragraphs (https://www.lipsum.com/)[1.txt] - Lorem Ipsum with 7 paragraphs (https://www.lipsum.com/)[2.txt] - Quijote de la Mancha by Miguel de Cervantes (https://www.gutenberg.org/cache/epub/60884/pg60884.txt)[3.txt] - The Adventures of Sherlock Holmes by Arthur Conan Doyle (https://www.gutenberg.org/cache/epub/1661/pg1661.txt)[4.txt] - The picture of Dorian Gray by Oscar Wilde (https://www.gutenberg.org/cache/epub/174/pg174.txt)[5.txt] - Beyond good and evil by Friedrich Wilheim Nietzsche (https://www.gutenberg.org/cache/epub/4363/pg4363.txt)[6.txt]

This dataset has been selected like this, so it has two text which should be fairly similar (1 & 2), a text which should be fairly different (3) and three texts which should have something in common even though they may be different (4, 5 & 6).

In this notebook we have implemented 4 classes which we will help us to compute the similarity of 2 texts. Those classes are: Shingling, CompareSets, MinHashing and CompareSignatures.

[ ]:
```
import os

dataset_path: str = "dataset"

texts: list[str] = []

for filename in os.listdir(dataset_path):
    if filename.endswith(".txt") and filename != "output.txt":
        with open(os.path.join(dataset_path, filename), "r") as f:
```

```
            text = f.read()
            texts.append(text)

print(texts[5])
```

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean venenatis, dui
eget ullamcorper fermentum, odio nulla malesuada nulla, nec interdum lacus nisi
a justo. Mauris ornare massa non nunc porttitor tristique. Duis tempor risus
eget fermentum malesuada. Mauris sed neque quis risus venenatis sodales.
Maecenas id elit posuere, lobortis mi nec, mollis mauris. Nam nulla ligula,
ornare iaculis nulla non, pretium malesuada urna. Sed eros felis, porttitor
vitae mollis vestibulum, consequat id mi. Donec at auctor dui, dignissim porta
lorem.

Quisque feugiat erat at ligula tincidunt, a pharetra lacus iaculis. Ut ac
lobortis massa, vehicula hendrerit lectus. Cras molestie odio ac felis tincidunt
mattis. Integer nec consequat odio, sit amet pellentesque neque. Praesent
bibendum risus sollicitudin, consequat orci in, scelerisque erat. Quisque
vestibulum arcu eget nulla porttitor, et elementum arcu venenatis. Ut vehicula
nunc ac magna bibendum condimentum. Donec id sagittis tortor, hendrerit tempus
felis. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed ultricies,
ipsum vel efficitur venenatis, tortor felis dignissim arcu, a porta ex ipsum non
orci. Sed et nisl aliquet, elementum felis a, vestibulum lorem. Etiam tempus
neque arcu. Nunc est purus, maximus eget molestie at, tristique ut libero. Donec
faucibus nisi urna, posuere imperdiet turpis hendrerit vitae. Nullam sodales
lacus nec metus tincidunt, vel luctus enim laoreet.

Duis facilisis hendrerit justo, quis interdum neque fermentum a. Maecenas
ullamcorper magna lacus, in sagittis felis fringilla ut. Sed aliquam urna
dictum, pretium tellus eget, aliquet tellus. Phasellus gravida tellus a erat
mollis, sit amet pharetra erat consectetur. Phasellus commodo quam sit amet
vulputate sodales. Quisque semper mauris vitae gravida iaculis. Suspendisse
aliquam magna sit amet vestibulum congue. Ut a ex fringilla, vestibulum odio eu,
finibus turpis. Quisque pretium viverra velit in vestibulum. Fusce tincidunt
diam sit amet feugiat euismod. Integer laoreet lectus id neque ultrices, quis
tristique felis condimentum. Sed tincidunt consectetur lorem, id iaculis lectus
ultrices in. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices
posuere cubilia curae; Nulla felis eros, gravida sed quam a, pretium tincidunt
sapien. Pellentesque molestie dui non consectetur vulputate.

Sed vitae nisi fermentum, placerat ligula nec, pulvinar enim. Nam ullamcorper
lacus eget nibh varius, id vehicula enim fermentum. Proin vehicula sit amet
lectus a laoreet. Fusce viverra lectus eu tincidunt mattis. Pellentesque
pharetra augue sit amet arcu varius accumsan. Donec egestas, tellus at sodales
ornare, massa purus luctus odio, nec rhoncus mauris est eu tortor. Pellentesque
volutpat ullamcorper ullamcorper. Donec at nunc pulvinar, bibendum magna non,
ultrices arcu.
```

Nulla sollicitudin ac sapien at dapibus. Sed at aliquam dolor, a iaculis sem. Donec non tincidunt urna. Aenean vitae porttitor orci. Maecenas semper tristique auctor. Vestibulum auctor ac nunc a vestibulum. Nullam nec egestas mi. Morbi dui nisl, elementum et pharetra nec, venenatis nec libero. Pellentesque eleifend ante ac augue rhoncus facilisis. Phasellus molestie, lorem vel molestie mollis, dolor mauris molestie risus, in gravida justo neque a dolor.

[ ]:
```
import os

dataset_path: str = "dataset"

texts: list[str] = []

for filename in os.listdir(dataset_path):
    if filename.endswith(".txt") and filename != "output.txt":
        with open(os.path.join(dataset_path, filename), "r") as f:
            text = f.read()
            texts.append(text)

print(texts[5])
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean venenatis, dui eget ullamcorper fermentum, odio nulla malesuada nulla, nec interdum lacus nisi a justo. Mauris ornare massa non nunc porttitor tristique. Duis tempor risus eget fermentum malesuada. Mauris sed neque quis risus venenatis sodales. Maecenas id elit posuere, lobortis mi nec, mollis mauris. Nam nulla ligula, ornare iaculis nulla non, pretium malesuada urna. Sed eros felis, porttitor vitae mollis vestibulum, consequat id mi. Donec at auctor dui, dignissim porta lorem.

Quisque feugiat erat at ligula tincidunt, a pharetra lacus iaculis. Ut ac lobortis massa, vehicula hendrerit lectus. Cras molestie odio ac felis tincidunt mattis. Integer nec consequat odio, sit amet pellentesque neque. Praesent bibendum risus sollicitudin, consequat orci in, scelerisque erat. Quisque vestibulum arcu eget nulla porttitor, et elementum arcu venenatis. Ut vehicula nunc ac magna bibendum condimentum. Donec id sagittis tortor, hendrerit tempus felis. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed ultricies, ipsum vel efficitur venenatis, tortor felis dignissim arcu, a porta ex ipsum non orci. Sed et nisl aliquet, elementum felis a, vestibulum lorem. Etiam tempus neque arcu. Nunc est purus, maximus eget molestie at, tristique ut libero. Donec faucibus nisi urna, posuere imperdiet turpis hendrerit vitae. Nullam sodales lacus nec metus tincidunt, vel luctus enim laoreet.

Duis facilisis hendrerit justo, quis interdum neque fermentum a. Maecenas ullamcorper magna lacus, in sagittis felis fringilla ut. Sed aliquam urna dictum, pretium tellus eget, aliquet tellus. Phasellus gravida tellus a erat mollis, sit amet pharetra erat consectetur. Phasellus commodo quam sit amet vulputate sodales. Quisque semper mauris vitae gravida iaculis. Suspendisse

aliquam magna sit amet vestibulum congue. Ut a ex fringilla, vestibulum odio eu, finibus turpis. Quisque pretium viverra velit in vestibulum. Fusce tincidunt diam sit amet feugiat euismod. Integer laoreet lectus id neque ultrices, quis tristique felis condimentum. Sed tincidunt consectetur lorem, id iaculis lectus ultrices in. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Nulla felis eros, gravida sed quam a, pretium tincidunt sapien. Pellentesque molestie dui non consectetur vulputate.

Sed vitae nisi fermentum, placerat ligula nec, pulvinar enim. Nam ullamcorper lacus eget nibh varius, id vehicula enim fermentum. Proin vehicula sit amet lectus a laoreet. Fusce viverra lectus eu tincidunt mattis. Pellentesque pharetra augue sit amet arcu varius accumsan. Donec egestas, tellus at sodales ornare, massa purus luctus odio, nec rhoncus mauris est eu tortor. Pellentesque volutpat ullamcorper ullamcorper. Donec at nunc pulvinar, bibendum magna non, ultrices arcu.

Nulla sollicitudin ac sapien at dapibus. Sed at aliquam dolor, a iaculis sem. Donec non tincidunt urna. Aenean vitae porttitor orci. Maecenas semper tristique auctor. Vestibulum auctor ac nunc a vestibulum. Nullam nec egestas mi. Morbi dui nisl, elementum et pharetra nec, venenatis nec libero. Pellentesque eleifend ante ac augue rhoncus facilisis. Phasellus molestie, lorem vel molestie mollis, dolor mauris molestie risus, in gravida justo neque a dolor.

1.2 Shingling

Shingling is a technique used in text analysis to represent a document as a set of overlapping subsequences of fixed length k, called k-shingles. To compute shingles, we slide a window of size k over the document and extract the k-length substrings that fall within the window. We then store these substrings as a set, which represents the shingles of the document.

[ ]:
```python
class Shingling():

    def __init__(self, k: int):
        self.k: int = k

    def get_shingles(self, document: str) -> set:
        """
        This method constructs k-shingles of a given length k from a given␣
↪document.
        """
        shingles: set = set()
        for i in range(len(document) - self.k + 1):
            shingle: str = document[i:i+self.k]
            shingles.add(shingle)
        return shingles

    def get_hashed_shingles(self, document: str):
        """
```

```
        This method computes a hash value for each unique shingle and␣
↪represents the document in the form of an ordered set of its hashed␣
↪k-shingles.
        """
        shingles: set = self.get_shingles(document)
        hashed_shingles: list[int] = [hash(shingle) for shingle in shingles]
        hashed_shingles.sort()
        return hashed_shingles
```

As an example to show what is shingling, we will use the following text: > "The quick brown fox jumps over the lazy dog"

We can see that the text is divided on chunks of 3 characters like "fox", "the" or "dog". If there is a space in the middle it is also considered as a character. So, if we want to get the shingles of this text with a k=3, we will get the following shingles:

[ ]:
```
example_text_1: str = "The quick brown fox jumps over the lazy dog"
example_text_2: str = "The agile black cat leaps over the active dog"

shingling: Shingling = Shingling(k=3)

example_shingles_1: list[str] = shingling.get_shingles(example_text_1)
print(example_shingles_1)

example_hashed_shingles_texts: list[list[int]] = []
for text in [example_text_1, example_text_2]:
    example_hashed_shingles: int = shingling.get_hashed_shingles(text)
    example_hashed_shingles_texts.append(example_hashed_shingles)
```

```
{'ps ', ' do', 'fox', 'qui', 'row', ' th', 'uic', 'y d', 'wn ', 'own', 'e l',
'ck ', 'The', 'zy ', 'e q', 'ver', 'x j', 'dog', 'ox ', 'er ', ' fo', 'bro', '
qu', 'ick', 's o', ' br', 'the', ' la', ' ov', 'n f', 'azy', ' ju', 'laz', 'he
', 'k b', 'ove', 'r t', 'ump', 'mps', 'jum'}
```

Now we do it with our text. We use a k=9 as we are analyzing text from books instead of emails. In the following lines we are doing exactly the same as we showed in the example above. In this case we will compute the hash of each shingle and we will store it in a set. We will do this for each text.

[ ]:
```
shingling: Shingling = Shingling(k=9)

hashed_shingles_texts: list[list[int]] = []

for text in texts:
    hashed_shingles: int = shingling.get_hashed_shingles(text)
    hashed_shingles_texts.append(hashed_shingles)
```

1.3 MinHashing

1.3.1 One-hot encoding

Before computing the MinHashing we will use the **One-hot encoding** to represent the shingles of each text. This will help us to compute the Jaccard similarity. To create the one-hot encoding for each text we create a set with all the shingles of all the texts. Then, for each text we create a vector with the size of the set of all shingles. If the shingle is in the text, we will put a 1 in the vector, otherwise we will put a 0. This way we will have a vector for each text with the size of the set of all shingles. We will do this for each text.

[ ]:
```
example_vocab_hashed: set = set()
for hashed_shingles in example_hashed_shingles_texts:
    for hashed_shingle in hashed_shingles:
        example_vocab_hashed.add(hashed_shingle)

example_matrix_one_hot_encoding: list[list[int]] = []
for hashed_shingles in example_hashed_shingles_texts:
    example_one_hot_encoding: list[int] = [1 if x in hashed_shingles else 0 for␣
 ↪x in example_vocab_hashed]
    example_matrix_one_hot_encoding.append(example_one_hot_encoding)
print(example_matrix_one_hot_encoding)
```

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

In the following code we do the same, but with the larger texts we originally had.

[ ]:
```
vocab_hashed: set = set()
for hashed_shingles in hashed_shingles_texts:
    for hashed_shingle in hashed_shingles:
        vocab_hashed.add(hashed_shingle)

matrix_one_hot_encoding: list[list[int]] = []
for hashed_shingles in hashed_shingles_texts:
    one_hot_encoding: list[int] = [1 if x in hashed_shingles else 0 for x in␣
 ↪vocab_hashed]
    matrix_one_hot_encoding.append(one_hot_encoding)
```

1.3.2 MinHashing

Now that we have the one-hot encoding for each text, we need to shuffle the texts multiple in order to create the signatures, which will be used to compute the similarity between the texts. We will do this by using the **MinHashing** technique.

To compute the signature of each text we look at each col of the one-hot encoding. We need to

find the first row which has a 1. We will store the row number in the signature. We will do this for each col of the one-hot encoding. This way we will have a signature for each text. We will do this for each text.

We will use our previous example to show how MinHashing works. First of all, we create a list with all the indexes.

[ ]:
```
example_hash_indexes: list[int] = list(range(1, len(example_vocab_hashed)+1))
print(example_hash_indexes)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 66, 67]
```

After we have the list, we shuffle it one time (in this case). We will use this shuffled list to compute the signature of each text. We will do this for each text.

[ ]:
```
from random import shuffle

shuffle(example_hash_indexes)

for i in range(1, 10):
    print(f"{i} -> {example_hash_indexes.index(i)}")
```

```
1 -> 47
2 -> 20
3 -> 17
4 -> 6
5 -> 38
6 -> 12
7 -> 65
8 -> 13
9 -> 25
```

The following class MinHashing is used to compute all the different shuffle needed to create the signature. It can be thought as an auxiliary class which assists Signature. Basically, it does what we have shown previously. In this case we can set as well how many shuffles we want to do. One shuffle is equivalent to one bit in the signature.

[ ]:
```
class MinHashing():

    def __init__(self, vocab_size: int, nbits: int):
        self.vocab_size: int = vocab_size
        self.nbits: int = nbits
        self.hashes: list[int] = []

    def create_hash_functions(self) -> list[int]:
        """
        Function for creating the hash vector / function
```

```
        """
        hash_indexes: list[int] = list(range(1, self.vocab_size+1))
        shuffle(hash_indexes)
        return hash_indexes

    def build_minhashing_functions(self) -> list[int]:
        """
        Function for building multiple minhashing vectors
        """
        hashes: list[int] = []
        for _ in range(self.nbits):
            hashes.append(
                self.create_hash_functions()
            )
        return hashes
```

1.3.3 Signature

To illustrate how the creation of the signature works, we will illustrate first how the signature finds the first bit. In this case it looks for the first index with a 1 in the column. This index will be the first bit of the signature. We will do this for each column of the one-hot encoding. This way we will have a signature for each text. We will do this for each text.

[ ]:
```
for i in range(1, len(example_vocab_hashed)+1):
    idx: int = example_hash_indexes.index(i)
    signature_value: int = matrix_one_hot_encoding[0][idx]
    print(f"{i}. Index: {idx} -> {signature_value}")
    if signature_value == 1:
        print("Match!")
        break
```

```
1. Index: 47 -> 0
2. Index: 20 -> 0
3. Index: 17 -> 0
4. Index: 6 -> 0
5. Index: 38 -> 0
6. Index: 12 -> 0
7. Index: 65 -> 0
8. Index: 13 -> 0
9. Index: 25 -> 0
10. Index: 4 -> 0
11. Index: 55 -> 0
12. Index: 35 -> 0
13. Index: 39 -> 0
14. Index: 58 -> 0
15. Index: 66 -> 0
16. Index: 44 -> 1
Match!
```

As we can see, after a several indexes, we find a 1. When we have the match, we store the index in the signature. We will do this for each text. For example, if the signature is of 20 bits, we will do this 20 times.

To compute this in a more systematic way, we have the class Signature which does all this computations. Internally, it computes the minhashing to create the different shuffled vectors for the indexes and then it computes the signature. We will do this for each text.

[ ]:
```
class Signature():

    def __init__(self, matrix_one_hot_encoding: list[list[int]], nbits: int =␣
 ↪20):
        self.vocab_size: int = len(matrix_one_hot_encoding[0])
        minhashing = MinHashing(self.vocab_size, nbits)
        self.minhash_functions: list[list[int]] = minhashing.
 ↪build_minhashing_functions()

    def create_hash(self, vector: list[int]) -> list[int]:
        """
        This function creates our signatures matching the 1s
        """
        signature: list[int] = []
        indices_of_ones = {i: 1 for i, v in enumerate(vector) if v == 1}
        for function in self.minhash_functions:
            for i in range(1, self.vocab_size+1):
                idx: int = function.index(i)
                if idx in indices_of_ones:
                    signature.append(idx)
                    break
        return signature
```

Now, we will look which are the signatures with the examples texts. Later, we will use this signatures to compute the similarity between the texts.

[ ]:
```
example_texts_signatures: Signature = Signature([example_text_1,␣
 ↪example_text_2])

example_text_1_signature: list[int] = example_texts_signatures.
 ↪create_hash(example_matrix_one_hot_encoding[0])
print(example_text_1_signature)

example_text_2_signature: list[int] = example_texts_signatures.
 ↪create_hash(example_matrix_one_hot_encoding[1])
print(example_text_2_signature)
```

```
[24, 6, 0, 16, 38, 28, 24, 0, 36, 26, 31, 26, 6, 0, 31, 26, 18, 19, 29, 29]
[39, 20, 0, 18, 1, 20, 17, 0, 5, 30, 37, 26, 0, 37, 20, 26, 18, 19, 30, 17]
```

In this examples, we can see that some values in the signature are similar. This means that the

texts are similar at least in some part. This makes sense as if we look at the examples text we can see that they have some words in common.

- Example text 1: "The quick brown fox jumps over the lazy dog"
- Example text 2: "The agile black cat leaps over the active dog"

In this example, we can expect that there is some similarity, as there are some similar words like "dog", "over" or "the". However, we can see that the similarity is not that high. This is because the texts are not that similar.

Now we do it, with our dataset, instead of the examples.

[ ]:
```
texts_signatures_func: Signature = Signature(matrix_one_hot_encoding,␣
 ↪nbits=1000)

texts_signatures: list[list[int]] = []
for text_one_hot_encoding in matrix_one_hot_encoding:
    text_signature: list[int] = texts_signatures_func.
 ↪create_hash(text_one_hot_encoding)
    texts_signatures.append(text_signature)
```

We can do the comparison of the signatures manually if we want. Nonetheless, looking at this manually for each text would be really difficult. This is where the Jaccard similarity can help to compute how similar texts are.

1.3.4 CompareSignatures

To compare the signatures and see how close those texts are we can compute the jaccard similarity. The jaccard similarity is computed by looking at the number of elements in common between two sets and dividing it by the total number of elements in both sets. In our case, we will look at the number of elements in common between two signatures and dividing it by the total number of elements in both signatures. This way we will get a value between 0 and 1. The closer the value is to 1, the more similar the texts are. The closer the value is to 0, the less similar the texts are.

For the jaccard similarity we will use the following formula:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In the following lines we will compute the Jaccard similarity of each pair of texts in the example texts.

[ ]:
```
def jaccard(x: set, y: set):
    return len(x.intersection(y)) / len(x.union(y))

jaccard(set(example_text_1_signature), set(example_text_2_signature)),␣
 ↪jaccard(set(example_text_1), set(example_text_2))
```

[ ]:
```
(0.21052631578947367, 0.6428571428571429)
```

Here are the example texts: - Example text 1: “The quick brown fox jumps over the lazy dog” - Example text 2: “The agile black cat leaps over the active dog”

In this particular case, the result is quite interesting. If we look at the similarity without doing the minhashing we can observe that the texts are somehow similar. However, if we look at the similarity after doing the minhashing we can see that the similarity is not that high. Both text they have repeated words like “The”, “over” or “dog” and quite similar length, however everything else is not exactly the same. This is why the similarity is not that high. This is a good example of how the minhashing can help us to compute the similarity between texts.

We will create a class and a function to compute the jaccard similarity in order to be more clear.

[ ]:
```
def jaccard_similarity(x: set, y: set) -> float:
    """
    This method computes the Jaccard similarity of two sets.
    """
    intersection_size: int = len(x.intersection(y))
    union_size: int = len(x.union(y))
    jaccard_similarity: float = intersection_size / union_size
    return jaccard_similarity
```

Using our dataset and computing the jaccard similarity for all the texts with the minhashing and in the original form we can see the following results:

[ ]:
```
sets_jaccard_similarities: list[list[float]] = []
signatures_jaccard_similarities: list[list[float]] = []

for text_x in texts:
    text_jaccard_similarities_with_other_texts: list[int] = []
    for text_y in texts:
        text_jaccard_similarities_with_other_texts.append(
            jaccard_similarity(set(text_x), set(text_y))
        )
    sets_jaccard_similarities.append(text_jaccard_similarities_with_other_texts)

for text_signature_x in texts_signatures:
    text_signature_jaccard_similarities_with_other_texts: list[int] = []
    for text_signature_y in texts_signatures:
        text_signature_jaccard_similarities_with_other_texts.append(
            jaccard_similarity(set(text_signature_x), set(text_signature_y))
        )
    signatures_jaccard_similarities.
↪append(text_signature_jaccard_similarities_with_other_texts)
```

If we look at the similarities pair we can spot some interesting things: - Text 1 and 2 are quite similar. This makes sense as they are the same text but with different number of paragraphs. - The rest of the texts are not that similar. This makes sense as they are different texts.

It is good to notice that the each row represent the text in the position of the row. For example, the first row represents the similarity of the text 1 with the rest of the texts. The same with the

columns. This is why the matrix is mirrored.

[ ]:
```
import numpy as np
print("Jaccard Similarities for direct sets")
print("")
print(np.round(np.matrix(sets_jaccard_similarities), 3))
print("")
print("-"*20)
print("")
print("Jaccard Similarities for signatures")
print("")
print(np.round(np.matrix(signatures_jaccard_similarities), 3))
```

```
Jaccard Similarities for direct sets

[[1.    0.717 0.761 0.647 0.567 0.583]
 [0.717 1.    0.687 0.646 0.679 0.667]
 [0.761 0.687 1.    0.718 0.6   0.615]
 [0.647 0.646 0.718 1.    0.69  0.707]
 [0.567 0.679 0.6   0.69  1.    0.976]
 [0.583 0.667 0.615 0.707 0.976 1.   ]]

--------------------

Jaccard Similarities for signatures

[[1.    0.003 0.001 0.    0.    0.   ]
 [0.003 1.    0.003 0.    0.    0.   ]
 [0.001 0.003 1.    0.    0.    0.   ]
 [0.    0.    0.    1.    0.    0.   ]
 [0.    0.    0.    0.    1.    0.494]
 [0.    0.    0.    0.    0.494 1.   ]]
```

1.4 LSH: Locality Sensitive Hashing

Now we are going to implement the LSH algorithm. This algorithm will help us to find the similarity between texts if we have bigger texts. For now, we have used really small text samples, but if we have bigger texts, the minhashing will take a lot of time to compute. This is why we will use the LSH algorithm. This algorithm will help us to find the similarity between texts without having to compute the minhashing for all the texts. This way we will save a lot of time.

The first thing we need to do is to create the bands. The bands are used to create the buckets. The buckets are used to store the signatures. The signatures are used to compute the similarity between texts. We will do this for each text.

[ ]:
```
class LSH():
    def __init__(self, b: int, r: int, t:float):
        self.b = b
        self.r = r
```

```
        self.t = t

    def split_vector(self, signature: list[int]):
        assert len(signature) % self.b == 0
        subvectors: list[list[int]] = []
        for i in range(0, len(signature), self.r):
            subvectors.append(signature[i : i+self.r])
        return subvectors

    def hash_band(self, band: list[int]):
        return hash(tuple(band))

    def find_matches(self, bands: list[list[int]]):
        hash_table = {}
        for i, band in enumerate(bands):
            band_hash = self.hash_band(band)
            if band_hash in hash_table:
                hash_table[band_hash].append(i)
            else:
                hash_table[band_hash] = [i]
        return [indices for indices in hash_table.values() if len(indices) >␣
↪self.t * self.b]

    def lsh(self, signatures: list[list[int]]):
        matches = []
        for signature in signatures:
            bands = self.split_vector(signature)
            matches.extend(self.find_matches(bands))
        return matches
```

As we can see in the following application of the LSH algorithm, we can see that the texts 1 and 2 are similar as they have a match if we use 2 bands. Even though, this could be true, in this case it is a false positive.

[ ]:
```
lsh: LSH = LSH(b=2, r=1, t=0.8)

matches = lsh.lsh([example_text_1_signature, example_text_2_signature])

print(f"Found {len(matches)} matches")
print(matches)
```

```
Found 13 matches
[[0, 6], [1, 12], [2, 7, 13], [9, 11, 15], [10, 14], [18, 19], [1, 5, 14], [2,
7, 12], [3, 16], [6, 19], [9, 18], [10, 13], [11, 15]]
```

However, if we increment the number of bands, we are decreasing the number of false positives. Therefore, if we use 5 bands, the number of matches is 0, indicating that the texts are not similar.

[ ]:
```
lsh: LSH = LSH(b=5, r=1, t=0.8)

matches = lsh.lsh([example_text_1_signature, example_text_2_signature])

print(f"Found {len(matches)} matches")
print(matches)
```

```
Found 0 matches
[]
```

1.5 Perfomance comparison

In this section we will compare the performance of the 3 algorithms (comparetexts, CompareSignatures and LSH) with a large number of texts. We will use the dataset from Drug Review Dataset (Drugs.com). We have used a sub sample of the test data for training machine learning models which can be found in the *output.txt*. The idea is to compare how efficient is each algorithm comparing the runtime. We will see that the LSH algorithm is the most efficient one.

[ ]:
```
import os

dataset_path: str = 'dataset'
filename: str = 'output.txt'
reviews: list[str] = []

with open(os.path.join(dataset_path, filename), 'r') as f:
    reviews = [line.rstrip() for line in f]
```

[ ]:
```
shingling = Shingling(5)

hashed_shingles_reviews: list[list[int]] = []

for review in reviews:
    hashed_shingles: int = shingling.get_hashed_shingles(review)
    hashed_shingles_reviews.append(hashed_shingles)

reviews_vocab_hashed: set = set()
for hashed_shingles in hashed_shingles_reviews:
    for hashed_shingle in hashed_shingles:
        reviews_vocab_hashed.add(hashed_shingle)

reviews_matrix_one_hot_encoding: list[list[int]] = []
for hashed_shingles in hashed_shingles_reviews:
    one_hot_encoding: list[int] = [1 if x in hashed_shingles else 0 for x in␣
 ↪reviews_vocab_hashed]
    reviews_matrix_one_hot_encoding.append(one_hot_encoding)
reviews_signatures_func: Signature = Signature(reviews_matrix_one_hot_encoding,␣
 ↪nbits=100)
```

```
reviews_signatures: list[list[int]] = []
for review_one_hot_encoding in reviews_matrix_one_hot_encoding:
    text_signature: list[int] = texts_signatures_func.
 ↪create_hash(text_one_hot_encoding)
    texts_signatures.append(text_signature)
```

[ ]:
```
import time

start: time = time.time()
sets_jaccard_similarities: list[list[float]] = []

for review_x in reviews:
    review_jaccard_similarities_with_other_reviews: list[int] = []
    for review_y in reviews:
        review_jaccard_similarities_with_other_reviews.append(
            jaccard_similarity(set(review_x), set(review_y))
        )
    sets_jaccard_similarities.
 ↪append(review_jaccard_similarities_with_other_reviews)
print(f"### {time.time() - start} seconds ###")

####################################

start: time = time.time()
signatures_jaccard_similarities: list[list[float]] = []
for review_signature_x in reviews_signatures:
    review_signature_jaccard_similarities_with_other_reviews: list[int] = []
    for review_signature_y in reviews_signatures:
        review_signature_jaccard_similarities_with_other_reviews.append(
            jaccard_similarity(set(review_signature_x), set(review_signature_y))
        )
    signatures_jaccard_similarities.
 ↪append(review_signature_jaccard_similarities_with_other_reviews)

print(f"### {time.time() - start} seconds ###")

########################
start: time = time.time()
lsh: LSH = LSH(b=50, r=10, t=0.8)
matches = lsh.lsh(reviews_signatures)
print(f"### {time.time() - start} seconds ###")
```

```
### 3.5808310508728027 seconds ###
### 0.00015115737915039062 seconds ###
### 9.894371032714844e-05 seconds ###
```