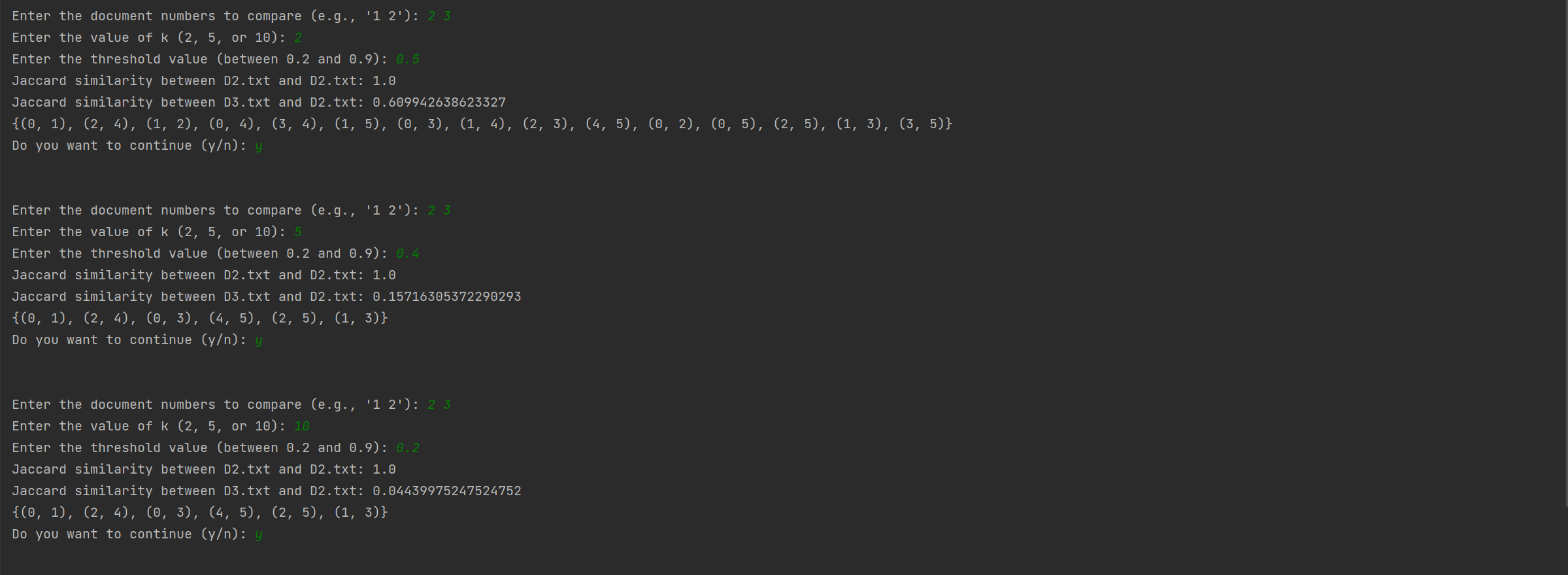
**Results:**

**DOC 1 VS DOC 2:**

**A screenshot of a computer

Description automatically generated**

**DOC 2 VS DOC 3:**

****

**DOC 1 VS DOC 3:**

A screenshot of a computer program

Description automatically generated

As we can see results if we decrease the number of shingles Jaccard similarity increase but when we use decent number of shingles (5,10) we can see more clear and accurate results same goes with the LSH if the shingles are less more common pair get extracted where as if shingles are increased common pair are less if we talk about the similarities between documents there is some similarity exist in the doc 1 and doc 3 as we can see the common candidates which is being returned by the class whereas doc 2 is kind a unique if we compare with doc 1 and 2 further more if we increase the b and r we can see more clear and accurate results for the similarity comparison.

**Explanation:**

def create\_shingles(doc, k\_values=[2, 5, 10]):  
 shingles = {}  
 for k in k\_values:  
 shingles[k] = set()  
 with open(doc, 'r', encoding='utf-8') as file:  
 text = file.read()  
 for i in range(len(text) - k + 1):  
 shingle = text[i:i+k]  
  
 shingles[k].add(shingle)  
 print(shingles)  
 return shingles

This function generates shinglesfor a given document based on specified lengths (k-values).

It reads the text from the provided document file and breaks it into shingles of different lengths specified by the k\_values parameter.

def create\_vocabulary(shingles\_list):  
 vocabulary = set()  
 for shingles in shingles\_list:  
 for k, shingle\_set in shingles.items():  
 vocabulary.update(shingle\_set)  
 return vocabulary

The function aggregates shingles from multiple documents to construct a vocabulary set. This vocabulary encompasses all unique shingles present across the documents.

It iterates through each document's shingle set and combines them to form the vocabulary set, ensuring no duplicate shingles are included.

def transform\_to\_vector(shingles, vocabulary):  
 vector = [1 if shingle in shingles else 0 for shingle in vocabulary]  
 return vector

This function converts a set of shingles into a binary vector representation using one-hot encoding. Each element in the vector corresponds to a shingle in the vocabulary, with a value of 1 indicating the presence of the shingle and 0 indicating absence.

It compares each shingle in the document with the vocabulary set and assigns 1 or 0 based on its presence or absence, respectively, in the document.

def jaccard\_similarity(set1, set2):  
 intersection = len(set1.intersection(set2))  
 union = len(set1.union(set2))  
 return intersection / union

This function basically Calculates the Jaccard similarity coefficient between two sets of shingles. The Jaccard similarity measures the intersection over the union of the sets, providing a measure of overlap between the shingles of two documents.

It computes the ratio of the number of common shingles to the total number of unique shingles in the two sets

def minhash\_signature(vector, hash\_functions):  
 signature = [min([hash\_func[i] for i, x in enumerate(vector) if x]) for hash\_func in hash\_functions]  
 return signature

Computes a MinHash signature for a binary vector representing a document. MinHashing is a technique used for estimating Jaccard similarity efficiently by reducing the dimensionality of the data.

It applies multiple hash functions to the binary vector and selects the minimum hash value for each hash function, resulting in a signature representing the document.

class LSH:  
 def \_\_init\_\_(self, b):  
 self.b = b  
 self.buckets = []  
 self.counter = 0  
 for \_ in range(b):  
 self.buckets.append({})  
  
 def make\_subvecs(self, signature):  
 l = len(signature)  
 assert l % self.b == 0  
 r = int(l / self.b)  
 # break signature into subvectors  
 subvecs = []  
 for i in range(0, l, r):  
 subvecs.append(signature[i:i + r])  
 return np.stack(subvecs)  
  
 def add\_hash(self, signature):  
 subvecs = self.make\_subvecs(signature).astype(str)  
 for i, subvec in enumerate(subvecs):  
 subvec = ','.join(subvec)  
 if subvec not in self.buckets[i].keys():  
 self.buckets[i][subvec] = []  
 self.buckets[i][subvec].append(self.counter)  
 self.counter += 1  
  
 def check\_candidates(self):  
 candidates = []  
 for bucket\_band in self.buckets:  
 keys = bucket\_band.keys()  
 for bucket in keys:  
 hits = bucket\_band[bucket]  
 if len(hits) > 1:  
 candidates.extend(combinations(hits, 2))  
 return set(candidates),print(set(candidates))

**Initialization:**

The class is initialized with a parameter b, which determines the number of hash bands.

Hash bands are partitions of the signature space used for hashing.

**Signature Processing:**

The make\_subvecs method divides a signature into subvectors based on the number of hash bands.

This process enables more efficient hashing by breaking down the signature into smaller, manageable components.

**Hashing:**

The add\_hash method hashes signatures into buckets based on their subvectors.

Each bucket represents a potential group of similar signatures.

Hashing enables the rapid identification of potential candidate pairs for similarity comparison.

**Candidate Identification:**

The check\_candidates method identifies potential candidate pairs within the hash buckets.

It iterates through the buckets and looks for instances where multiple signatures are hashed to the same bucket.

These pairs are considered potential candidates for similarity comparison.

**Purpose:**

The primary purpose of the LSH class is to enable efficient approximate nearest neighbor search.

By dividing signatures into subvectors and hashing them into buckets, LSH reduces the search space for similarity comparison.

This approach is particularly useful for high-dimensional data, where exact nearest neighbor search methods become impractical due to computational complexity.

def user\_interface(documents):  
 while True:  
 shingles\_list = []  
 vocabulary = set()  
  
 doc\_choice = input("\n\nEnter the document numbers to compare (e.g., '1 2'): ")  
 doc\_choice = list(map(int, doc\_choice.split()))  
  
 k = int(input("Enter the value of k (2, 5, or 10): "))  
 threshold = float(input("Enter the threshold value (between 0.2 and 0.9): "))  
  
 shingles\_list = [create\_shingles(doc, [k]) for doc in documents]  
  
 # Step 2: Combine Shingles of All Documents to Create Vocabulary  
 vocabulary = create\_vocabulary(shingles\_list)  
  
 # Step 5: Create hash functions for minhashing  
 hash\_functions = [np.random.permutation(len(vocabulary)) for \_ in range(100)]  
  
 for doc\_index in doc\_choice:  
 doc\_index -= 1 # Adjust index to match list indexing  
 doc = documents[doc\_index]  
  
 for i in range(len(doc\_choice)):  
 if i == doc\_index:  
 continue  
  
 other\_doc\_index = doc\_choice[i] - 1  
 other\_doc = documents[other\_doc\_index]  
  
 shingles1 = shingles\_list[doc\_index][k]  
 shingles2 = shingles\_list[other\_doc\_index][k]  
  
 vector1 = transform\_to\_vector(shingles1, vocabulary)  
 vector2 = transform\_to\_vector(shingles2, vocabulary)  
  
 if i == 0:  
 jaccard\_sim = jaccard\_similarity(shingles1, shingles2)  
 print(f"Jaccard similarity between {doc} and {other\_doc}: {jaccard\_sim}")  
  
 sig1 = minhash\_signature(vector1, hash\_functions)  
 sig2 = minhash\_signature(vector2, hash\_functions)  
  
 if i == 0:  
 lsh\_sim = lsh\_similarity(sig1, sig2, threshold)  
 print(f"LSH similarity between {doc} and {other\_doc}: {lsh\_sim}")  
  
  
  
 if input("Do you want to continue (y/n): ").lower() != 'y':  
 i==0  
 i=0  
 break

Provides a user-friendly interface for users to interactively compare documents using various similarity metrics and parameters. It streamlines the process of document comparison by accepting user inputs for document selections, shingle length, and similarity threshold.

It prompts users to input document selections, shingle length (k), and similarity threshold, and displays the computed similarity metrics and visualizations.