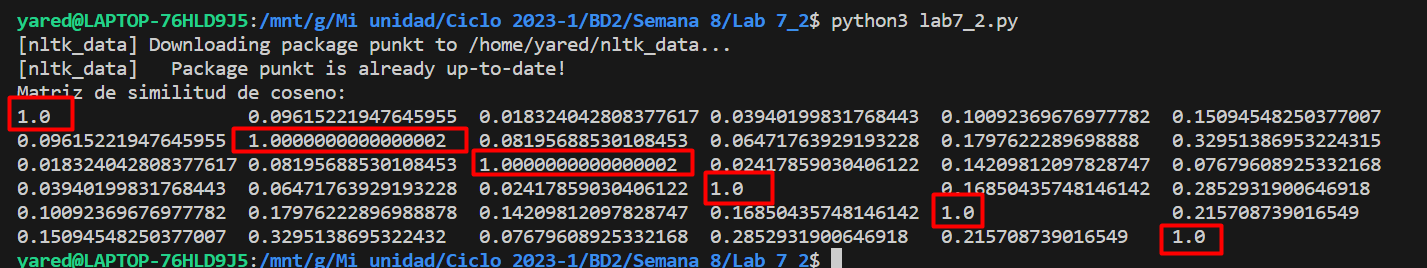
**Ranking Retrieval**

Profesor Heider Sanchez

**P3.** Utilice Notebook de Python y elabore una matriz de similitud de coseno entre los documentos de la colección “El Señor de los Anillos”. Debe aplicar los pesos TF-IDF.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Doc1 | Doc2 | Doc3 | Doc4 | Doc5 | Doc6 |
| Doc1 | 1 |  |  |  |  |  |
| Doc2 |  | 1 |  |  |  |  |
| Doc3 |  |  | 1 |  |  |  |
| Doc4 |  |  |  | 1 |  |  |
| Doc5 |  |  |  |  | 1 |  |
| Doc6 |  |  |  |  |  | 1 |



**P4.** Además**,** Implemente el índice invertido para dar soporte al modelo de **ranked retrieval** con la similitud de coseno.

Inverted

Index

1. Doc1

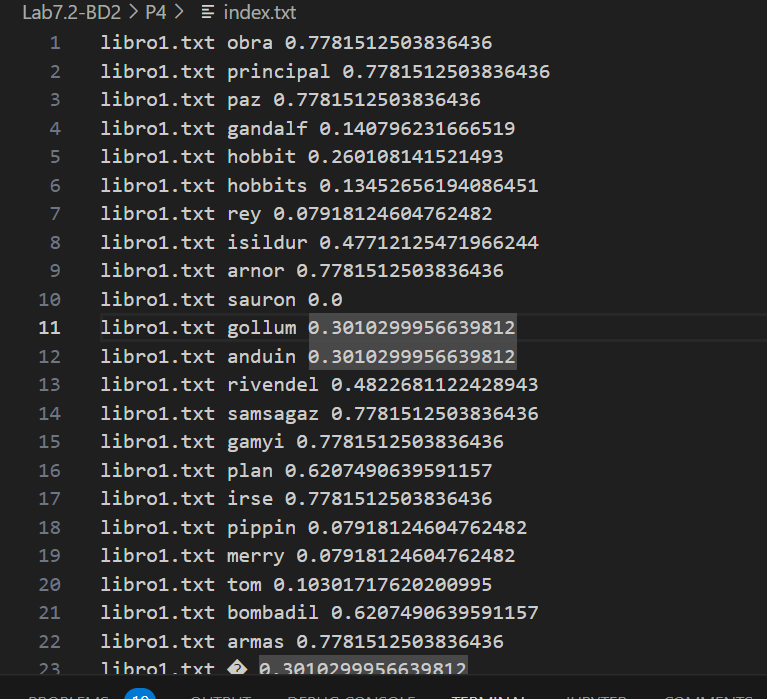
2. Doc2

3. Doc3

.

.

1. Estructura del índice invertido en Python:



1. Algoritmo para construir el índice:

def building(self):

        if os.path.exists(self.index\_file): #Si el índice ya fue creado, no hacer nada

            pass

        else:

            num\_roots = 500

            derivador = SnowballStemmer('spanish')

            libros = ['libro1.txt', 'libro2.txt', 'libro3.txt','libro4.txt','libro5.txt','libro6.txt']

            # Initialize a set for stopwords

            stoplist = set()

            # Read stop words from file into the set

            with open('stoplist.txt', 'r',encoding='ISO-8859-1') as file:

                stoplist = set(file.read().split('\n'))

            # Initialize defaultdict for tokens

            tokens = defaultdict(set)

            # Process each book

            for file\_id, archivo in enumerate(libros, start=1):

                with open(archivo, 'r',encoding='ISO-8859-1') as file:

                    tokens\_in\_file = nltk.word\_tokenize(file.read().lower())

                    tokens\_in\_file = [token for token in tokens\_in\_file if token.isalpha()]  # filter out non-alphanumeric characters

                    for token in tokens\_in\_file:

                        root = derivador.stem(token)

                        if root not in stoplist:

                            tokens[root].add(file\_id)

            # Limit tokens to the 500 most frequent ones

            tokens = dict(sorted(tokens.items(), key=lambda x: len(x[1]), reverse=True)[:num\_roots])

            # Sort the dictionary alphabetically

            tokens = dict(sorted(tokens.items()))

            # tf

            tf = {

                "libro1.txt": {},

                "libro2.txt": {},

                "libro3.txt": {},

                "libro4.txt": {},

                "libro5.txt": {},

                "libro6.txt": {}

            }

            ## Cuanto cuántas veces aparece cada palabra en cada libro (TF)

            for file\_id, archivo in enumerate(libros, start=1):

                with open(archivo, 'r',encoding='ISO-8859-1') as file:

                    tokens\_in\_file = nltk.word\_tokenize(file.read().lower())

                    tokens\_in\_file = [token for token in tokens\_in\_file if token.isalpha()]

                    for token in tokens\_in\_file:

                        if(token in tokens):

                            root = derivador.stem(token)

                            if root in tf[archivo]:

                                tf[archivo][root] += 1

                            else:

                                tf[archivo][root] = 1

            # idf

            idf = {}

            for token in tokens:

                idf[token] = np.log10(len(libros) / len(tokens[token])) #idf = log(numero total de documentos / numero de documentos en los que aparece la palabra)

            for doc in tf:

                for word in tf[doc]:

                    tf[doc][word] = 1 + np.log10(tf[doc][word])

            self.tf = tf

            self.idf = idf

            # Guardo el idf en un archivo

            with open('idf.txt', 'w',encoding='ISO-8859-1') as file:

                for word in self.idf:

                    file.write(f"{word} {self.idf[word]}\n")

            # Calcular tf-idf

            textos\_tfidf = {}

            for doc in tf:

                textos\_tfidf[doc] = {}

                for word in tf[doc]:

                    #Producto punto entre escalares es el producto normal.

                    # No realizo producto entre arrays porque estoy usando diccionarios, pero es la misma idea

                    textos\_tfidf[doc][word] = tf[doc][word] \* idf[word]

            self.tfidf = textos\_tfidf

            # Escribo en el archivo index.txt el tf-idf de cada palabra en cada documento

            with open(self.index\_file, 'w',encoding='ISO-8859-1') as file:

                for doc in self.tfidf:

                    for word in self.tfidf[doc]:

                        file.write(f"{doc} {word} {self.tfidf[doc][word]}\n")

1. Función de recuperación usando la similitud de coseno:

def retrieval(self, query, k): #k es el numero de documentos mas relevantes que se quieren obtener

        # diccionario para el score

        score = {}

        # preprocesar la query: extraer los terminos unicos

        query = nltk.word\_tokenize(query.lower())

        query = [token for token in query if token.isalpha()]

        # calcular el tf del query

        tf\_query = {}

        for token in query:

            if token in tf\_query:

                tf\_query[token] += 1

            else:

                tf\_query[token] = 1

        for word in tf\_query:

            tf\_query[word] = 1 + np.log10(tf\_query[word])

        # Leo el idf desde el archivo idf.txt

        with open('idf.txt', 'r',encoding='ISO-8859-1') as file:

            for line in file:

                word, idf = line.split()

                self.idf[word] = float(idf)

        # calcular el idf del query

        idf\_query = {}

        for word in tf\_query:

            if word in self.idf:

                idf\_query[word] = self.idf[word]

            else:

                idf\_query[word] = 0

        # calcular el tf-idf del query

        tfidf\_query = {}

        for word in tf\_query:

            tfidf\_query[word] = tf\_query[word] \* idf\_query[word]

        # cargo el tf-idf desde el archivo index.txt

        with open(self.index\_file, 'r',encoding='ISO-8859-1') as file:

            for line in file:

                doc, word, tfidf = line.split()

                tfidf = float(tfidf)

                if doc not in self.tfidf:

                    self.tfidf[doc] = {}

                self.tfidf[doc][word] = tfidf

        # aplicar similitud de coseno y guardarlo en el diccionario score

        for doc in self.tfidf:

            score[doc] = InvertIndex.cosine\_sim(tfidf\_query, self.tfidf[doc])

        # ordenar el score de forma descendente

        result = sorted(score.items(), key= lambda tup: tup[1], reverse=True)

        # retornamos los k documentos mas relevantes (de mayor similitud al query)

        return result[:k] #retorna una lista de tuplas (nombre del documento, similitud)

    def cosine\_sim(Q, Doc):

        # Calcula la suma de los productos de los elementos de 2 diccionarios

        # Si un elemento no está en el otro diccionario, se asume que su valor es 0

        # Esto es porque el producto de 0 por cualquier número es 0

        # 1. Obtengo las palabras de ambos documentos

        words = set(Q.keys()) | set(Doc.keys())

        # 2. Calculo el producto punto

        dot\_product = sum(Q.get(word, 0) \* Doc.get(word, 0) for word in words)

        # 3. Calculo las normas

        norm\_Q = np.sqrt(sum(Q.get(word, 0) \*\* 2 for word in words))

        norm\_Doc = np.sqrt(sum(Doc.get(word, 0) \*\* 2 for word in words))

        # 4. Calculo la similitud

        return dot\_product / (norm\_Q \* norm\_Doc)