The Impacts of Linkage Types on Hierarchical Agglomerative Clustering for Text Mining

Sena Özbelen
Computer Engineering Department
Gebze Technical University
s.ozbelen2019@gtu.edu.tr

***Abstract*—With the improvement of Automatic Speech Recognition (ASR), the whole conversation can be processed and information can be retrieved using data mining techniques. This project focuses on one of the most common classification techniques. Clustering is highly used to group documents and it can be used to group ASR outputs as well. This project uses the TED Talks English Dataset as ASR output and assesses the results of each linkage type for Hierarchical Agglomerative Clustering (HAC).**

Keywords*—*text mining, TF-IDF, text preprocessing, document clustering, hierarchical agglomerative clustering, linkage types.

I. Introduction

Information Retrieval (IR) is widely used to extract useful information from data. One of the techniques to perform IR is clustering. Clustering is an unsupervised learning method which is not dependent on ground truths. It basically groups the similar data entries regarding the similarity matrix. For text mining, this is called "Document Clustering". This method groups all the documents according to their numerical representations. Therefore, these documents should be preprocessed and converted. With this conversion, the similarity matrix can be calculated and the clustering can be performed. Hierarchical clustering are used in text mining. This clustering technique gives the results depending on the linkage type. Linkage types are used to split or merge the clusters. The most common types are "Single Linkage", "Complete Linkage" and "Average Linkage". This project focuses on the impacts of these linkage types on the hierarchical clustering results, specifically agglomerative clustering.

II. Literature Analysis

In general, document clustering is performed by using K-Means clustering. Some papers suggest different approaches by using hierarchical clustering. The paper "Pattern and Cluster Mining on Text Data" shows the results for both K-Means clustering and hierarchical clustering. The papers about hierarchical clustering mostly discuss the impacts of the hierarchical agglomerative clustering on text mining which can be seen in the papers "Hierarchical Agglomerative Clustering Using Common Neighbours Similarity" [2] and "Hierarchical document clustering based on cosine similarity measure" [3]. These papers also show that cosine similarity matrix is widely used for document clustering. For the preprocessing part, the paper "Pattern and Cluster Mining on Text Data" [1] suggests a pathway which is basically stop-word removal, stemming and then TF-IDF calculation.

III. The Methods

This approach includes these steps: text preprocessing, clustering implementation by using three linkage types, post-processing (interpretation of the result).

A. Text Preprocessing

Texts have to be preprocessed before calculating TF-IDF. There is a need for only the words which can represent the whole text. Therefore, we do not need punctuation, numbers, contractions (I'll = I will) and stop words (the most commonly used words). These are easy to remove and handle properly. Also, there are some text-specific words such as names in dialogues, words between parentheses defining actions such as "(Laughter)", special names, speaking only words such as "umm". These are not easy to handle. Therefore, they are partially handled.

- Regular Expression Library : Removing punctuation, numbers, words between parentheses defining actions, replacing contractions with the longer versions
- NLTK : Removing stop words

After removing or replacing the specific words or characters, the rest of the words should be handled not to affect the count because suffixes can affect the count. For instance, the word "cat" and the word "cats" lead to the same meaning but they will count individually. The papers suggest stemming but lemmatization is widely used and it works better. Therefore, lemmatization from NLTK is used.

B. Clustering Implementation

Before implementing the clustering itself, text data needs to be converted to numerical values. TF-IDF is used for this purpose. It basically calculates the term frequency and inverse document frequency. TfidfVectorizer function from scikit-learn is used to get TF-IDF results. Figure 1 shows how to calculate and get the TF-IDF results for the given texts. It has additional two parameters which determines the upper and lower limits for the count values.

After calculating TF-IDF, the similarity matrix based on cosine similarity is needed. The problem for calculating this matrix is that TF-IDF matrix is a sparse matrix which is full of zeros. Therefore, there is a need for optimization. The library "scipy" has a function changing the format of the matrix to CSR (Compressed Sparse Row) format. After this conversion, the library "scikit-learn" is used to calculate the cosine similarity matrix. Clustering needs a dissimilarity matrix so subtract the matrix from 1 gives us the dissimilarity matrix.

$$\mathbf{tf}(t,d) = \frac{f_d(t)}{\max\limits_{w \in d} f_d(w)}$$

$$\mathbf{idf}(t,D) = \ln\left(\frac{|D|}{|\{d \in D : t \in d\}|}\right)$$

$$\mathbf{tfidf}(t,d,D) = \mathbf{tf}(t,d) \cdot \mathbf{idf}(t,D)$$

$$\mathbf{tfidf}'(t,d,D) = \frac{\mathbf{idf}(t,D)}{|D|} + \mathbf{tfidf}(t,d,D)$$

$f_d(t) :=$ frequency of term t in document d

$D :=$ corpus of documents

Figure 1: TF-IDF Formulas.

Hierarchical agglomerative clustering with three linkage types are implemented. Hierarchical clusterings basically divide or merge sub-clusters. Agglomerative approach takes every data as a subcluster at first. Then it calculates the distance between clusters by using the given linkage type. Single linkage takes the smallest distance between clusters. Complete linkage takes the highest distance between clusters and average linkage takes the average distance between clusters. Figure 2 shows the linkages.

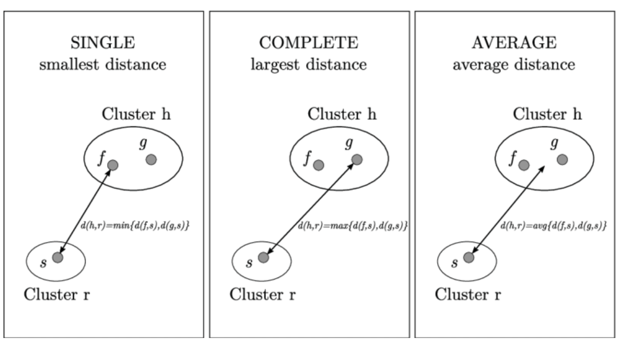

Figure 2: Linkage Types.

This algorithm merges the sub-clusters until there is only one cluster. It merges the clusters which have the minimum distance. It gives us a dendrogram to visualize the merging steps.

C. Post-processing

The results are visualized by using dendrograms. For dendrograms, scipy library has a function to draw. This library is used to visualize the result.

IV. The Results

For an easy interpretation, I used a sample (10 texts) from the dataset.

Figures show the results of both my implementation and the library function as a dendrogram. Since its format is not supported by the dendrogram function, I used it to visualize my results. They give the same results.

We can see that each linkage type gives a unique result for the sample. Therefore, linkage type is one of the most crucial parameter for this algorithm.

```
After merging : [[0], [1], [2, 7], [3], [4], [5], [6], [8], [9]]
After merging : [[0], [1], [2, 7], [3, 6], [4], [5], [8], [9]]
After merging : [[0], [1], [2, 7, 3, 6], [4], [5], [8], [9]]
After merging : [[0], [1, 2, 7, 3, 6], [4], [5], [8], [9]]
After merging : [[0, 1, 2, 7, 3, 6], [4], [5], [8], [9]]
After merging : [[0, 1, 2, 7, 3, 6, 8], [4], [5], [9]]
After merging : [[0, 1, 2, 7, 3, 6, 8, 4], [5], [9]]
After merging : [[0, 1, 2, 7, 3, 6, 8, 4, 9], [5]]
After merging : [[0, 1, 2, 7, 3, 6, 8, 4, 9, 5]]
[[ 2.         7.          0.5757763   2.       ]
 [ 3.         6.          0.63571122  2.       ]
 [ 2.         3.          0.64147336  4.       ]
 [ 1.         2.          0.64643212  5.       ]
 [ 0.         1.          0.71874205  6.       ]
 [ 0.         3.          0.73326982  7.       ]
 [ 0.         1.          0.73540336  8.       ]
 [ 0.         2.          0.79431525  9.       ]
 [ 0.         1.          0.81074201 10.       ]]
```

Figure 3: Single Linkage Results.

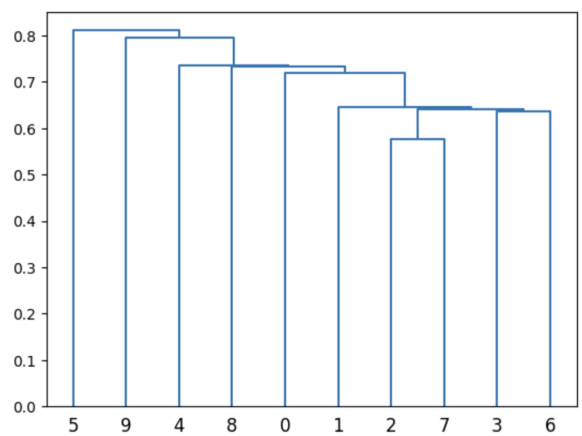

Figure 4: Single Linkage Dendrogram.

```
After merging : [[0], [1], [2, 7], [3], [4], [5], [6], [8], [9]]
After merging : [[0], [1], [2, 7], [3, 6], [4], [5], [8], [9]]
After merging : [[0], [1, 2, 7], [3, 6], [4], [5], [8], [9]]
After merging : [[0, 4], [1, 2, 7], [3, 6], [5], [8], [9]]
After merging : [[0, 4], [1, 2, 7, 8], [3, 6], [5], [9]]
After merging : [[0, 4], [1, 2, 7, 8, 3, 6], [5], [9]]
After merging : [[0, 4], [1, 2, 7, 8, 3, 6], [5, 9]]
After merging : [[0, 4, 1, 2, 7, 8, 3, 6], [5, 9]]
After merging : [[0, 4, 1, 2, 7, 8, 3, 6, 5, 9]]
[[ 2.         7.          0.5757763   2.       ]
 [ 3.         6.          0.63571122  2.       ]
 [ 1.         2.          0.68304877  3.       ]
 [ 0.         3.          0.73540336  2.       ]
 [ 1.         4.          0.77552369  4.       ]
 [ 1.         2.          0.8475494   6.       ]
 [ 2.         3.          0.85690866  2.       ]
 [ 0.         1.          0.90771415  8.       ]
 [ 0.         1.          0.942579   10.       ]]
```

Figure 5: Complete Linkage Results.

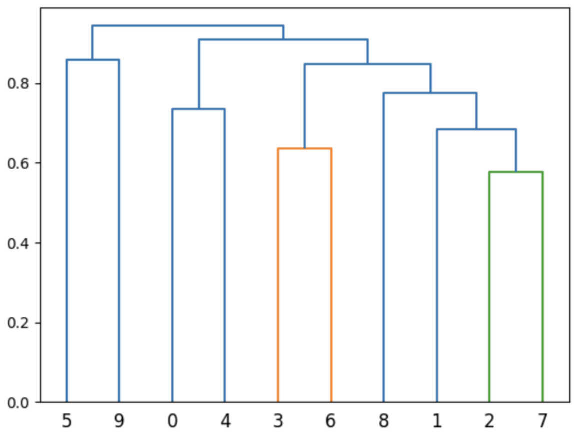

Figure 6: Complete Linkage Dendrogram.

```
After merging : [[0], [1], [2, 7], [3], [4], [5], [6], [8], [9]]
After merging : [[0], [1], [2, 7], [3, 6], [4], [5], [8], [9]]
After merging : [[0], [1, 2, 7], [3, 6], [4], [5], [8], [9]]
After merging : [[0, 4], [1, 2, 7], [3, 6], [5], [8], [9]]
After merging : [[0, 4], [1, 2, 7, 3, 6], [5], [8], [9]]
After merging : [[0, 4], [1, 2, 7, 3, 6, 8], [5], [9]]
After merging : [[0, 4, 1, 2, 7, 3, 6, 8], [5], [9]]
After merging : [[0, 4, 1, 2, 7, 3, 6, 8], [5, 9]]
After merging : [[0, 4, 1, 2, 7, 3, 6, 8, 5, 9]]
[[ 2.         7.         0.5757763   2.        ]
 [ 3.         6.         0.63571122  2.        ]
 [ 1.         2.         0.66474045  3.        ]
 [ 0.         3.         0.73540336  2.        ]
 [ 1.         2.         0.73989061  5.        ]
 [ 1.         3.         0.78138064  6.        ]
 [ 0.         1.         0.82166198  8.        ]
 [ 1.         2.         0.85690866  2.        ]
 [ 0.         1.         0.86790786 10.        ]]
```

Figure 7: Average Linkage Results.

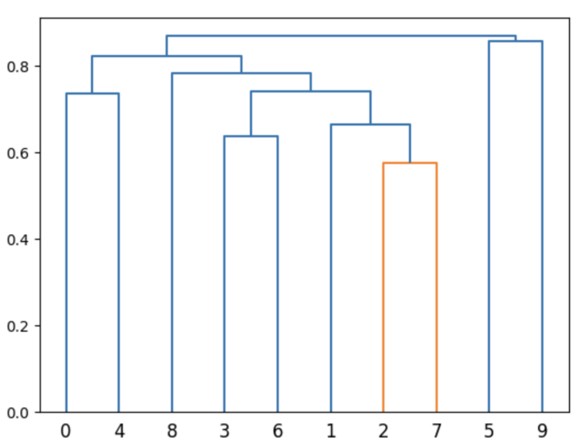

Figure 8: Average Linkage Dendrogram.

All of them merged text 2 and text 7 at first.

Text 2 = [’business’, ’entrepreneur’, ’global issues’, ’poverty’, ’social change’, ’women in business’]

Text 7 = [’Planets’, ’art’, ’poetry’, ’life’, ’love’, ’empathy’, ’humanity’, ’personal growth’, ’visualizations’, ’creativity’, ’community’]

Then, they merged text 3 and text 6.

Text 3 = [’health’, ’health care’, ’medical research’, ’medicine’, ’obesity’, ’public health’]

Text 6 = [’Surgery’, ’health care’, ’invention’, ’medical research’, ’prosthetics’]

Changes begin after this point. Single linkage merged {2,7} and {3,6} clusters. Complete and average linkages merged text 1 and {2,7}. Apparently, text 1 is much closer to the text 2 and 7.

Text 1 = [’TEDx’, ’business’, ’communication’, ’culture’, ’leadership’, ’society’, ’indigenous peoples’]

Then, single linkage merged text 1 and {2,7,3,6}. Complete and average linkage merged text 0 and text 4. Both texts are quite similar as topics show. Text 0 is about creating of our own reality and text 4 is creation of sophisticated behaviors and functions from large groups of simple elements.

Text 0 = [’brain’, ’choice’, ’fear’, ’humanity’, ’identity’, ’motivation’, ’personal growth’, ’success’, ’blindness’]

Text 4 = [’psychology’, ’TED-Ed’, ’animation’, ’fish’, ’animals’, ’water’, ’oceans’, ’memory’, ’rivers’, ’brain’, ’consciousness’]

Single linkage merged text 0 and {1,2,7,3,6}. Complete linkage merged text 8 and {1,2,7}. Average linkage merged {1,2,7} and {3,6}. Text 3 and 6 seem much closer to text 1, 2 and 7 than text 8 since they are all related to humans.

Text 8 = [’trees’, ’art’, ’plants’, ’natural resources’, ’nature’, ’ecology’, ’biodiversity’, ’botany’, ’environment’, ’community’]

Single linkage merged text 8 and {0,1,2,7,3,6}. Complete linkage merged {1,2,7,8} and {3,6}. Average linkage merged text 8 and {1,2,7,3,6}.

Single linkage merged text 4 and {0,1,2,7,3,6,8}. Complete linkage merged text 5 and text 9. Average linkage merged {0,4} and {1,2,7,3,6,8}. Text 5 and 9 have some similar topics.

Text 5 = [’refugees’, ’mental health’, ’humanity’, ’children’, ’communication’, ’community’, ’compassion’, ’emotions’, ’empathy’, ’global issues’, ’education’, ’war’, ’society’, ’social change’, ’Syria’, ’TED Fellows’]

Text 9 = [’history’, ’war’, ’animation’, ’TED-Ed’, ’world cultures’, ’activism’, ’culture’, ’women’, ’government’, ’politics’, ’social change’, ’youth’, ’society’]

Single linkage merged text 9 and {0,1,2,7,3,6,8,4}. Complete linkage merged {0,4} and {1,2,7,8,3,6}. Average linkage merged text 5 and text 9.

Single linkage merged text 5 and {0,1,2,7,3,6,8,4,9}. Complete and average linkage merged {5,9} and {0,4,1,2,7,8,3,6}.

The result is that there is no correct linkage type to use because these results are based on this sample. For another sample, the results differ. The most appropriate choice will be is do some experiments with different linkage types to see which one gives a better result.

References

[1] D. Agnihotri, K. Verma and P. Tripathi, "Pattern and Cluster Mining on Text Data," 2014 Fourth International Conference on Communication Systems and Network Technologies, Bhopal, India, 2014, pp. 428-432, doi: 10.1109/CSNT.2014.92.

[2] M. Makrehchi, "Hierarchical Agglomerative Clustering Using Common Neighbours Similarity," 2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI), Omaha, NE, USA, 2016, pp. 546-551, doi: 10.1109/WI.2016.0093.

[3] S. K. Popat, P. B. Deshmukh and V. A. Metre, "Hierarchical document clustering based on cosine similarity measure," 2017 1st International Conference on Intelligent Systems and Information Management (ICISIM), Aurangabad, India, 2017, pp. 153-159, doi: 10.1109/ICISIM.2017.8122166.

[4] https://towardsdatascience.com/tf-idf-term-frequency-and-inverse-dense-frequency-techniques-472bf1ba311b

[5] https://www.researchgate.net/figure/Different-linkage-methods-for-hierarchical-clustering_fig5_329208978

V. Appendix

Listing 1: The Video Link

```
https://youtu.be/GLcXDQNAPBo
```

Listing 2: Jupyter Notebook Code

```
# Import necessary libraries

# In[1]:


import pandas as pd
import numpy as np
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import re
import ssl
from sklearn.feature_extraction.text
    import TfidfVectorizer


# Download NLTK Wordnet

# In[2]:


try:
    _create_unverified_https_context = ssl
        ._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context =
        _create_unverified_https_context

nltk.download('wordnet')


# Read the dataset

# In[3]:


df = pd.read_csv('transcripts/ted_talks_en
    .csv')
df.head()


# In[4]:


df.transcript[0]


# In[5]:


df.shape


# List the contractions for the
    preprocessing part

# In[6]:


contractions = {
    "i'm": "i␣am",
    "i'm'a": "i␣am␣about␣to",
    "i'm'o": "i␣am␣going␣to",
    "i've": "i␣have",
    "i'll": "i␣will",
    "i'll've": "i␣will␣have",
    "i'd": "i␣would",
    "i'd've": "i␣would␣have",
    "Whatcha": "What␣are␣you",
    "amn't": "am␣not",
    "ain't": "are␣not",
    "aren't": "are␣not",
    "'cause": "because",
    "can't": "cannot",
    "can't've": "cannot␣have",
    "could've": "could␣have",
    "couldn't": "could␣not",
    "couldn't've": "could␣not␣have",
    "daren't": "dare␣not",
    "daresn't": "dare␣not",
    "dasn't": "dare␣not",
    "didn't": "did␣not",
    "didn't": "did␣not",
    "don't": "do␣not",
    "don't": "do␣not",
    "doesn't": "does␣not",
    "e'er": "ever",
    "everyone's": "everyone␣is",
    "finna": "fixing␣to",
```

```
"gimme": "give␣me",
"gon't": "go␣not",
"gonna": "going␣to",
"gotta": "got␣to",
"hadn't": "had␣not",
"hadn't've": "had␣not␣have",
"hasn't": "has␣not",
"haven't": "have␣not",
"he've": "he␣have",
"he's": "he␣is",
"he'll": "he␣will",
"he'll've": "he␣will␣have",
"he'd": "he␣would",
"he'd've": "he␣would␣have",
"here's": "here␣is",
"how're": "how␣are",
"how'd": "how␣did",
"how'd'y": "how␣do␣you",
"how's": "how␣is",
"how'll": "how␣will",
"isn't": "is␣not",
"it's": "it␣is",
"'tis": "it␣is",
"'twas": "it␣was",
"it'll": "it␣will",
"it'll've": "it␣will␣have",
"it'd": "it␣would",
"it'd've": "it␣would␣have",
"kinda": "kind␣of",
"let's": "let␣us",
"luv": "love",
"ma'am": "madam",
"may've": "may␣have",
"mayn't": "may␣not",
"might've": "might␣have",
"mightn't": "might␣not",
"mightn't've": "might␣not␣have",
"must've": "must␣have",
"mustn't": "must␣not",
"mustn't've": "must␣not␣have",
"needn't": "need␣not",
"needn't've": "need␣not␣have",
"ne'er": "never",
"o'": "of",
"o'clock": "of␣the␣clock",
"ol'": "old",
"oughtn't": "ought␣not",
"oughtn't've": "ought␣not␣have",
"o'er": "over",
"shan't": "shall␣not",
"sha'n't": "shall␣not",
"shalln't": "shall␣not",
"shan't've": "shall␣not␣have",
"she's": "she␣is",
"she'll": "she␣will",
"she'd": "she␣would",
"she'd've": "she␣would␣have",
"should've": "should␣have",
"shouldn't": "should␣not",
"shouldn't've": "should␣not␣have",
"so've": "so␣have",
"so's": "so␣is",
"somebody's": "somebody␣is",
"someone's": "someone␣is",
"something's": "something␣is",
"sux": "sucks",
"that're": "that␣are",
"that's": "that␣is",
"that'll": "that␣will",
"that'd": "that␣would",
"that'd've": "that␣would␣have",
"'em": "them",
"there're": "there␣are",
"there's": "there␣is",
"there'll": "there␣will",
"there'd": "there␣would",
"there'd've": "there␣would␣have",
"these're": "these␣are",
"they're": "they␣are",
"they've": "they␣have",
"they'll": "they␣will",
"they'll've": "they␣will␣have",
"they'd": "they␣would",
"they'd've": "they␣would␣have",
"this's": "this␣is",
"this'll": "this␣will",
"this'd": "this␣would",
"those're": "those␣are",
"to've": "to␣have",
"wanna": "want␣to",
"wasn't": "was␣not",
"we're": "we␣are",
"we've": "we␣have",
"we'll": "we␣will",
"we'll've": "we␣will␣have",
"we'd": "we␣would",
"we'd've": "we␣would␣have",
"weren't": "were␣not",
"what're": "what␣are",
"what'd": "what␣did",
"what've": "what␣have",
"what's": "what␣is",
"what'll": "what␣will",
"what'll've": "what␣will␣have",
"when've": "when␣have",
"when's": "when␣is",
"where're": "where␣are",
"where'd": "where␣did",
"where've": "where␣have",
"where's": "where␣is",
"which's": "which␣is",
"who're": "who␣are",
"who've": "who␣have",
"who's": "who␣is",
"who'll": "who␣will",
"who'll've": "who␣will␣have",
"who'd": "who␣would",
"who'd've": "who␣would␣have",
"why're": "why␣are",
"why'd": "why␣did",
"why've": "why␣have",
"why's": "why␣is",
"will've": "will␣have",
"won't": "will␣not",
```

```
    "won't've": "will␣not␣have",
    "would've": "would␣have",
    "wouldn't": "would␣not",
    "wouldn't've": "would␣not␣have",
    "y'all": "you␣all",
    "y'all're": "you␣all␣are",
    "y'all've": "you␣all␣have",
    "y'all'd": "you␣all␣would",
    "y'all'd've": "you␣all␣would␣have",
    "you're": "you␣are",
    "you've": "you␣have",
    "you'll've": "you␣shall␣have",
    "you'll": "you␣will",
    "you'd": "you␣would",
    "you'd've": "you␣would␣have"
}


# Define the preprocessing function

# In[7]:


def preprocessing_data(texts):

    new_text = []

    for text in texts:
        text = text.lower() # make the
            text lower

        for contraction in contractions: #
             change the contractions
             if contraction in text:
                 text = re.sub(contraction,
                     contractions[
                    contraction], text)

        text = re.sub(r'\([^)]*\)', '',
            text) # remove texts with
            paranthesis
        text = re.sub(r'[^\w\s]', '', text
            ) # remove all the
            punctuations
        text = re.sub(r'\d', '', text) #
            remove all the numbers

        words = text.split() # tokenize
            the text

        # remove the stop words
        nltk.download('stopwords')
        stop_words = set(stopwords.words('
            english'))
        filtered_words = [word for word in
             words if word not in
            stop_words]

        # apply lemmatization
        lemmatizer = WordNetLemmatizer()
        lemmatized_words = [lemmatizer.
            lemmatize(word) for word in
                filtered_words]
        cleaned_text = "␣".join(
            lemmatized_words)
        new_text.append(cleaned_text)

    return new_text


# Preprocess texts and add them to the
    data frame

# In[8]:


preprocessed_text = preprocessing_data(df.
    transcript)
df["preprocessed_text"] =
    preprocessed_text


# Choose 10 texts to visualise easily

# In[9]:


selected = df.sample(10)


# TF-IDF Calculation

# In[10]:


# Create a vectorizer for the text data
# min_df = minimum frequency for the words
     in the dataset
# max_df = maximum frequency for the words
     in the dataset

vectorizer = TfidfVectorizer(min_df=2,
    max_df=0.95)
tf_idf_data = vectorizer.fit_transform(
    selected["preprocessed_text"])

tf_idf_data.shape


# The similarity matrix based on cosine
    similarity is built

# In[11]:


from sklearn.metrics.pairwise import
    cosine_similarity
from scipy.sparse import csr_matrix

# Convert sparse matrix to CSR format
sparse_matrix_csr = csr_matrix(tf_idf_data
    )

# Calculate cosine similarity
```

```
cosine_similarity_matrix =
    cosine_similarity(sparse_matrix_csr,
    sparse_matrix_csr)

# 'cosine_sim_matrix' now contains the
    cosine similarity values
print("Cosine Similarity Matrix:")
print(cosine_similarity_matrix)


# In[12]:


dissimilarity_matrix = 1.0 -
    cosine_similarity_matrix
np.fill_diagonal(dissimilarity_matrix,
    0.0)
print(dissimilarity_matrix)


# Implement hierarchical agglomerative
    clustering

# In[13]:


class agglomerative_clustering:
    def __init__(self, linkage):
        self.linkage = linkage # the
            linkage type
        self.linkage_matrix = []

    def get_linkage_matrix(self):
        return np.array(self.
            linkage_matrix)

    # Linkage Types
    # Single Linkage: the maximum
        similarity between two clusters
        are considered
    def single_linkage(self,
        dissimilarity_matrix, cluster1,
        cluster2):
         distances = dissimilarity_matrix[
            np.ix_(cluster1, cluster2)].
            min()
         return distances

    # Complete Linkage: the minimum
        similarity between two clusters
        are considered
    def complete_linkage(self,
        dissimilarity_matrix, cluster1,
        cluster2):
         distances = dissimilarity_matrix[
            np.ix_(cluster1, cluster2)].
            max()
         return distances

    # Average Linkage: the average
        similarity between two clusters
        are considered
    def average_linkage(self,
        dissimilarity_matrix, cluster1,
        cluster2):
         distances = dissimilarity_matrix[
            np.ix_(cluster1, cluster2)].
            mean()
         return distances


    def fit(self, dissimilarity_matrix):
        # get the number of entries in the
              dataset
        num_entry = dissimilarity_matrix.
            shape[0]
        # create clusters for each entry
            and assign each entry to the
            corresponding cluster
        clusters = [[i] for i in range(
            num_entry)]

        # until the desired number of
            clusters are created
        while len(clusters) > 1:
            # set the minimum distance to
                infinity
            min_distance = float('inf')
            # clusters to be merged
            cluster_indices = (0, 0)

            for i in range(len(clusters)
                -1):
                for j in range(i+1, len(
                    clusters)):
                    if self.linkage == '
                        single':
                         distance = self.
                            single_linkage
                            (
                            dissimilarity_matrix
                            , clusters[i],
                             clusters[j])
                    elif self.linkage == '
                        complete':
                         distance = self.
                            complete_linkage
                            (
                            dissimilarity_matrix
                            , clusters[i],
                             clusters[j])
                    elif self.linkage == '
                        average':
                         distance = self.
                            average_linkage
                            (
                            dissimilarity_matrix
                            , clusters[i],
                             clusters[j])
                    else:
                         raise ValueError(f
                            "Unsupported␣
                            linkage␣type:␣
                            {self.linkage}
```

```
                    ")

                if distance <
                    min_distance:
                     min_distance =
                         distance
                     cluster_indices =
                         (i, j)

        # Record the merging step in
            the linkage matrix
        self.linkage_matrix.append([
            cluster_indices[0],
            cluster_indices[1],
            min_distance, len(clusters
            [cluster_indices[0]]) +
            len(clusters[
            cluster_indices[1]])])

        # Merge the two closest
            clusters
        merged_cluster = clusters[
            cluster_indices[0]] +
            clusters[cluster_indices
            [1]]
        del clusters[cluster_indices
            [1]]
        clusters[cluster_indices[0]] =
             merged_cluster
        print("After merging : ", end=
            '')
        print(clusters)



# Results

# In[14]:


model_cosine = agglomerative_clustering(
    linkage='single')
model_cosine.fit(dissimilarity_matrix)

print(model_cosine.get_linkage_matrix())


# In[15]:


model_cosine = agglomerative_clustering(
    linkage='complete')
model_cosine.fit(dissimilarity_matrix)

print(model_cosine.get_linkage_matrix())


# In[16]:


model_cosine = agglomerative_clustering(
    linkage='average')
model_cosine.fit(dissimilarity_matrix)

print(model_cosine.get_linkage_matrix())


# In[17]:


from scipy.spatial.distance import
    squareform
from scipy.cluster.hierarchy import
    linkage, dendrogram
import numpy as np
import matplotlib. pyplot as plt

# Ensure the distance matrix is symmetric
dissimilarity_matrix_r = np.maximum(
    dissimilarity_matrix,
    dissimilarity_matrix.T)

# Apply linkage
linkage_matrix_s = linkage(squareform(
    dissimilarity_matrix_r), method='
    single')

# Plot dendrogram
dendrogram(linkage_matrix_s)
plt.show()


# In[18]:


# Apply linkage
linkage_matrix_c = linkage(squareform(
    dissimilarity_matrix_r), method='
    complete')

# Plot dendrogram
dendrogram(linkage_matrix_c)
plt.show()


# In[19]:


# Apply linkage
linkage_matrix_a = linkage(squareform(
    dissimilarity_matrix_r), method='
    average')

# Plot dendrogram
dendrogram(linkage_matrix_a)
plt.show()


# In[20]:


for i in range(10):
    print(f"Description of Text {i} : ",
```

```
        end='')
    print(selected.iloc[i, selected.
        columns.get_loc('description')])
    print()


# In[21]:


for i in range(10):
    print(f"Topics of Text {i} : ", end=''
        )
    print(selected.iloc[i, selected.
        columns.get_loc('topics')])
    print()
```