# Foreword

“Machine intelligence is the last invention that humanity will ever make” - Nick Bostrom

When I was 8 I began my coding journey.  
My interest for computer programming began at a very young age. I began to have a bond with computer language and mechanics. The possibilities were endless and my young and creative mind saw the computer world as a fun place to relax and challenge myself.

When I heard about the opportunity to write a thesis about computer programming I decided it could be a fun challenge. That is why I decided to write a computer programming project.

In computer programming there are hundreds of subtopics and terms to learn. The main reason I decided to write a project on machine learning is that I studied the subject a year before. The idea that a computer can learn and think just like a human fascinated me and pulled me into the extraordinary world of machine learning.

Machine learning is still a developing subject. There are projects today that are beyond anything that anyone thought a computer could accomplish 5 to 10 years ago. The capabilities of a computer are now defined by two factors:  
The processing power of the computer, and the code the programmer developed. Even though these are essential for any coding project, there are many projects that aim to create self thinking machines and language understanding super computers.

Even though I am no super computer building scientist capable of managing multiple computers and hacking into government secureties, I have an ambition to achieve and create an impressive project using the skills I acquire during the process.

# Table of contents

[**Foreword**](#_sliljjpglwng) **1**

[**Table of contents**](#_17ebykyfw41q) **2**

[**Introduction**](#_8h5xra9zpp79) **3**

[**Chapter one: the Corpus**](#_f00btk5citts) **4**

[**Chapter two: Text-Mining**](#_kblithjf7a2e) **15**

[**Chapter 3 Unsupervised learning**](#_6qginja2i4cq) **28**

[**Chapter 4 - analyzing the data**](#_m0p5z2anlhde) **36**

[**Conclusions and what I learned**](#_7z7thdv9dbv4) **45**

[**Bibliography:**](#_jiy0kxq5qeg8) **46**

[**CODE**](#_8stokk10tsvm) **47**

# Introduction

Using code to help computers understand text and language is a subtopic in machin learning called Text-mining. In this project I will attempt to uncover some interesting findings using two text-mining terms:

Word-Embedding, a technique used by many to uncover connections and similarities between words. This is done using a neural network and will be further discussed in the Text-mining chapter.

Statistical measures, a term for measuring the importance and weight of words inside documents. This is done by creating a mathematical algorithm for transforming a dataset of documents into a data frame of words and running the Statistical measures technique, Tf-Idf, on the documents.

The purpose of the project:

The main focus of my project is to enjoy and understand an advanced subtopic in machine learning. This project is the first time I have ever heard about the fascinating world of text-mining and am anticipating a fun workflow.

In a more scientific approach, in this project I am aiming to challenge myself and create a python script that can instruct the computer on how to understand language, which is absolutely fascinating.

# Chapter one: the Corpus

Introduction to the Corpus:

In the world of Data science and Data mining, there are many data types and

collections. Big companies tend to work with unsupervised learning methods to find

loops and patterns in big datasets.

A Corpus is a large and structured linguistical dataset. The corpus is used in many

applications worldwide. It is not uncommon for companies to use a corpus of reviews on

a particular product to visualize and manage the data and feedback they receive.

A corpus has a vital part in a data mining project. The data in each corpus is unique and

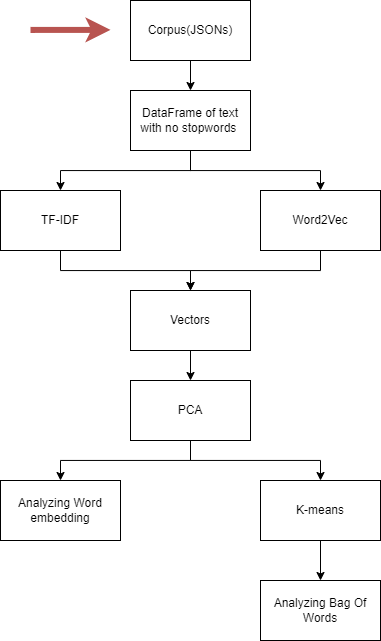
is dissected differently according to the needs of the users and the purpose it was built

for. In many instances, the data quality in itself is not the most important feature

affecting the accuracy of processing and quality of learning. The quality of the data

processing process in itself is directly affected by the way the data has been

incorporated into the relevant files and is presented to the processing algorithms.



My Corpus:

When I looked for a database to analyze and run my program, I searched in Kaggle and found a covid-19 corpus of public research and papers released since 2019. This corpus contains about 240 thousand JSON files, but I will run the program on 100 thousand files instead because my computer does not have enough RAM to complete the full computations.

JSON files:

In my project, the data is stored inside a collection of JSON files. The corpus, in my case, is the 25 gigabytes of documents stored as JSON files inside the computer.

A corpus could be made from any file type as long as it stores text. When working on a standard Word document, using the .DOC extension isn’t uncommon. We use the .PDF extension when looking at finished pieces of text. The main difference between a JSON file and a typical file you’d encounter on any office computer is that JSON files are formatted in a way that would be easier for the computer to understand.

Usually, turning a regular document into a JSON file is the human’s job and not the computer’s because every article is different. Transforming articles or documents is not always the same because different websites format papers differently.

For instance, in many sites, one may find links, popups, or commercial content embedded within the body of the article and would be extracted as an integral part of the text if a machine were to have done this process.

Converting a document into a JSON-type file is done in the following manner:





As you can see, the document is transformed by organizing the data in the unprocessed paper into “instructions.” I organized the data using a CSV file containing additional information about every document and a corpus containing one hundred thousand JSON files.

The first step of converting these files is to process the data and insert it from the JSON files into my data frame. The data frame has multiple columns. These columns are;

ID: a string representing a specific document. An ID labels each document. This parameter is crucial because the CSV file is organized and sorted according to the file’s ID, meaning the CSV, which contains unique data, can only be sorted using this parameter.

Bod: the body of the article. It is the main focus of the algorithm. In this step, I parsed the text from the body parameter of the JSON files using the ‘instructions’ given by the formatted document. This step allowed me to organize the text. It is also one of the most important parts of this process.

Abst: the document's abstract. In this attribute, I stored the abstract of each document. Unlike the id and the body, not every document had an abstract to process. The absence of an abstract was not a problem because it was not part of the processed data but instead was saved for a less impactful purpose.

Doi: Like the id, it is another label for the document. I have found that using this attribute to analyze and sort the files was much less effective in the grand scheme of things.

Url: the URL of the document. I chose to implement this attribute into my data frame because I added the abstract. Adding these attributes now will help identify documents and find sources if needed.

Author: the author of the report. The authors of the documents are important for citation and crediting sources. Giving credit to a report’s author is very important and is taken very seriously in the scientific community.

Title: the title of the document. This attribute is important because it helps identify whether or not the document’s subject and the topic are accurate. That is why adding the title as an attribute could assist in dissecting the results of the algorithm.

Natural Language Processing:

Transforming the data inside my corpus into clean data points is a process called Natural Language Processing.

The phrase Natural Language Processing can be divided into two parts:

The Natural Language is any language evolved by humans and is still spoken today. Processing is the process of extracting pure data and words from a document.

Natural Language Processing is the first step of implementing data into an application. This process can be divided into three stages.

The first stage in Natural Language Processing is importing the data kept inside the corpus into a data frame. This step is complete by using the JSON module in python. With the assistance of the json.load function, I created a class for the JSON-type files. In this class, we have a parameter for the attributes representing the JSON document and other data according to the CSV file as stated before in the JSON file section of the chapter.

| class Read:  def \_\_init\_\_(self, path, df):  file = open(path)  data = json.load(file)  self.id = data['paper\_id']  self.abstract = []  self.body = []  self.doi = []  self.url = []  self.author = []  self.title = []  self.journal = []  abst = []  bod = []  for i in data['abstract']:  abst.append(i['text'])  for i in data['body\_text']:  bod.append(i['text'])  self.abstract = ' '.join(abst)  self.body = ' '.join(bod)  self.doi = df['doi'].loc[df['sha'] == self.id]  self.url = df['url'].loc[df['sha'] == self.id]  self.author = df['authors'].loc[df['sha'] == self.id]  self.title = df['title'].loc[df['sha'] == self.id]  self.journal = df['journal'].loc[df['sha'] == self.id] |
| --- |

As you can see, I have created a class that receives a path of a JSON file and a pandas data frame. Using the CSV file of all the folder paths of the JSON documents and the id of each document in the corpus, I can deduce which file is which and organize the data.

Corrupted files:

Using a corpus could be challenging, especially if you did not create it. The corpus may have some documents that are not compatible with your work, for example, a corrupted file that the code could not read.

Usually, errors in files like corruption or empty documents are dealt with using a simple line of code that skips the error:

| try:  data = Json\_reader.Read(i, metaDf) except Exception:  continue |
| --- |

In the code above, I used the try function to process the data before the computer executed anything on that file.  
If a file has failed to load or cannot be processed, the code skips the file to continuously process data with no breaks or problems.

After using the class with a for loop, I finally have all the documents inside one data frame. Now that I have completed the first stage of Natural Language Processing, I can move on to an intermediate step less important than the other three.

Language check:

After processing all of the files, we have a simple pandas data frame. This data frame contains every piece of literature and extra parameters from the CSV file. Before moving on to the second major step in Natural Language Processing, we need to make sure there are no documents in languages other than English.   
Not all corpora contain documents in different languages. Some only contain a single language. It is usually best to make sure the corpus is in a single language. This is done by using the langdetect function to determine in what language the document is in.

Google’s language-detection library is a java library that was created to help programers detect the language a document is in.

How it works:

The langdetect function is a matrix of probabilities that two letters will be in a sequence in a particular language. By computing Wikipedia articles in different languages, Google had created a database of these probabilities. For example: if I took the word ‘oblivious,’ the computer will run the model on the sequence of letters, and the likelihood of two letters in an alphabet is like this. I.e., o→b = 73% English, b→l 84% English, and so on. The model returns the prediction is made according to the language the function found has the highest probability.

I created a function inside my “parse” module that removes any document that is not in English from the data frame:

| def checkLanguage(df):  langs = []  for i in tqdm(df['body']):  a = i.split(' ')  if len(a) > 50:  lang = detect(" ".join(a[:50]))  elif len(a) > 0:  lang = detect(" ".join(a))  langs.append(lang)  df['language'] = langs |
| --- |

As you can see, I created a list variable called “langs.” I added the language of each document to the list and added it to the data frame as a column named “language.”

In the final function I created called "parsedata," I removed the documents with a language other than English from my data frame. The removal was done like so:

| checkLanguage(df) df = tempdf[df['language'] == 'en'] |
| --- |

This process removed any document in a foreign language and was a small but important step before I started the second stage of Natural Language Processing.

The second stage of Natural Language Processing is “cleaning” the data. This stage is accomplished by using a list of words commonly referred to as “stop-words”;

Stop-words:

To apply NLP on a document, we remove unimportant words and, by doing so, reduce the size of our data. The unimportant words are called “stop-words.” Stop words are common in any language. These words are usually used to connect parts of a sentence.

e.g., in the sentence: “the cat had a hat on,” the words “the” “a” and “on” are stop-words. And by removing the stop-words from the sentence, we are left with the sequence- “cat had hat” the Idea of the sentence is still there, even though we managed to remove half of the words used in the sentence.

There are no defined rules for stop-words, neither are there any agreed-upon word lists that are universally used. Instead, there are lists of stop-words premade for projects that use NLP. And since we don’t care for the meaning of the sentences and whether a sentence is negative or positive, we can choose freely from the great variety of lists.

Removing Stop-words can cut the code's time to process the data.

In my code, I will be using stop-words to rid of futile data from the extensive weight of the data’s size. Many of the words in the Covid-19 corpus are normal terms used in a treatise. Formal terms that don’t add anything to the text but are mandatory for a fluent and informative article. These words aren’t usually found in normal texts and thus won’t appear in most stop-word lists. This means we must either add them to the stop-word register ourselves or find a list with fixed words normally found in professional articles.

My stop-words:  
As a base for my list, I have chosen to work with the SciSpacy list for stop-words. I made this decision because of two factors:

· The size of the preexisting list- The list consists of nouns, pronouns, verbs, and adverbs. It is full of commonly used words and linguistics used in every English-written document worldwide. The variety of words makes it so I will only have data with significant and uncommon words.

· The scientific words- The SciSpacy package was built to be implemented into scientific studies and projects in the scientific subjects. The list consists of words used in normal conversations and various uncommon words used in academic or scientific papers. I.e., using this list as a stop-word register is a great fit for my project.

After running the program’s code for the first time and looking at the significance of every word, I noticed some unrelated words that appear more frequently than I anticipated. These occurrences meant I had to add some custom words using a new list.

The function I created in my “Parse” module for removing stop-words from the text is called “stopwordreduce” and is written like so:

| no\_stop\_words = body.lower() # Full words reduction: for i in stopwords:  no\_stop\_words = re.sub(r'\s\*\b' + i + r'\b\s\*', ' ', no\_stop\_words) # Reduction of parts of other words for i in parts:  no\_stop\_words = no\_stop\_words.replace(str(i), '') |
| --- |

The “parts” variable is a list of all of the punctuation marks and contractions that I have noted to have appeared in the corpus regularly.

The “stopwords” variable contains a list of all the stopwords in the SciSpacy library.

Using this piece of code, I can remove unwanted words and contractions while only keeping the necessary words of the text.

The parsedata function:

After creating every function needed for parsing the data from the corpus, we create a function that adds a column to the dataset. This column is called “processed\_body” and we add it by applying every function we created. The function is called parsedata and can be used externally by calling the Parse module. This allows us to create a parsed dataframe by:

Implementing NLP into the “main” module:



| for i in tqdm(Jp, total=t):  try:  data = Json\_reader.Read(i, metaDf)  except Exception:  continue  if len(data.body) == 0:  continue  id = data.id  abstract = data.abstract  body = data.body  doi = data.doi  url = data.url  author = data.author  title = data.title  row = [id, abstract, body, doi, url, author, title]  a.append(row) clean = pd.DataFrame(columns=['id', 'abstract', 'body', 'doi', 'url', 'author', 'title'], data=a) parsedclean = Parse.parsedata(clean) |
| --- |

In the code:

Jp is a list of all of the directories of the JSON files. I then enter a for loop and with every repetition, the i value is a different folder destination. Every repetition, the computer attempts to read the JSON file using the Json\_reader module that I created.

I then create a list of the different attributes of the JSON file and the CSV and append the list into a new column. Finally, I run the parsedata function on the data frame and with that, I finish the NLP process.

Summary for chapter 1:

Summary:

As we have covered in the above chapter, the written data comprising the Corpus is the

core for any Big Data processing. The Corpus may be a text based one or another form

of data. In many instances the Corpus is built by an external supplier. Such outsourcing

allows those with limited resources access to large and refined data sets, as I used

within the scope of this project. When considering this fact of continued use of data sets

which are being constantly enlarged, yet many studies and users are reliant on them, I

believe that to some extent we might find a bias towards well cited and trusted sources

whilst other sources may not be included in these well forged and bound data sets.

Although this fact adds much credibility to my personal work and I am happy with the

\*0fact that the Corpus of data I used is a well established and formidable one I should

state that I am also weary of the fact that the data on the world wide web not included in

the scope of the Corpus I used may bring about a misrepresentation or skew the

eventual findings of my work. Inclusion of unorthodox texts and those which are of a

questionable source may affect the findings as well since these are not necessarily

reliant on any real findings or facts. Hence, I believe use of a Corpus of data which is

based solely on accredited data sources, with an authentic and recognizable aurthor to

be the best choice for the Big Data enthusiast, such as me.

I think that the fact such formidable sets of data can be found on the world wide web is

key for allowing young people to experiment with Big Data analysis.

Within the scope of the designated Corpus the Json files are the key element. These

files allow the program to draw the raw text from them. In turn, the raw extracted text is

processed and Stop words removed from it. The Stop words add to the amount of data

to be processed without having any true benefit. In actuality, even when one is reading

a text these words do not necessarily add to understanding the context and hence are

redundant. When a computer program has to tackle such great amounts of wording and

data, the computing effort and time should be precise and accurate. These efforts

should also be concise and to the point in order to efficiently use the limited computing

resources we have. When the stop words have been removed the data sets can be

processed more efficiently and the results scrutinized readily. The removal of the Stop

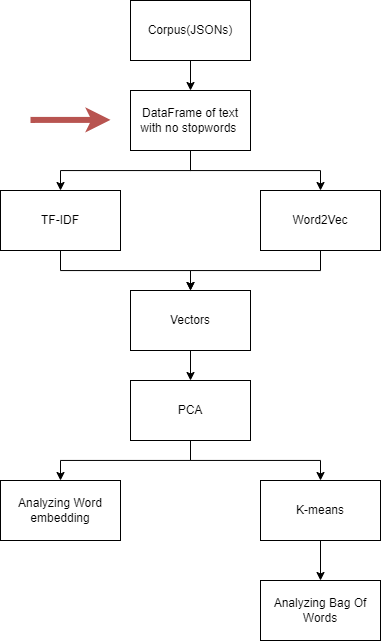
words shortens the processing time and hence allows the user to perform multiple

assessments of the data in a short period of time, and relatively small computing

resources.

The well processed body of text stored and processed within the Corpus is the key

element allowing the next step of text mining to proceed.



# Chapter two: Text-Mining

Introduction to Text-Mining:

Wikipedia defines Text mining also referred to as text data mining as the process of

deriving high quality information from text. This is performed by the software which

transforms the text into values the computer can interpret. The text is processed after

parts of it have been removed with emphasis on stop words. This allows the data to be

of higher quality and hopefully meaning to the processing software. Text mining

technologies are used by many companies and institutions. These technologies are

used for finding documents, performing searches, performing analyses and for

assessment of research and publication trends. Text mining is a tool, which when

mastered, can be an excellent vehicle for performing quick assessments and powerful

searches.

Vectors:

According to Britannica, a general knowledge encyclopedia, in physics, vectors are a quantity that has both magnitude and direction.

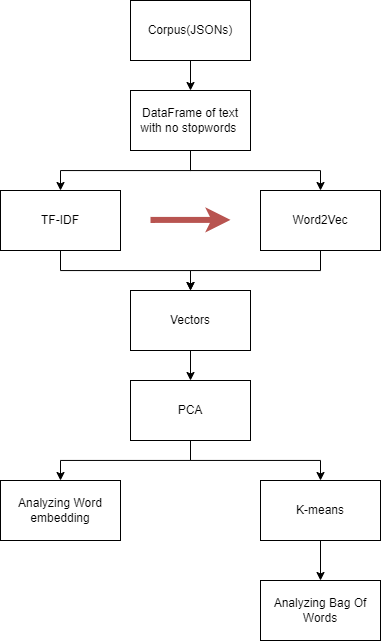
Vectors usually have multiple variables, representing various attributes at once. In computers, Vectors can have any amount of attributes, Much like an array. The texts stored inside our data frame could be visualized as real-valued vectors once the word embedding process is complete. To do so, we must first understand how to represent a word in a document as a numerical value.

Vector space:

According to the machine learning wiki article about vector spaces that was edited by the linear algebra wing of MIT, a vector space is the set of vectors that can be modified in a machine learning algorithm. A vector space can be high dimensional and can be transformed using dimensionality reduction. In this project, the corpus will be transformed using multiple functions into two different vector spaces. The first is a Word-embedding space and the second a statistical measure embedding-space.

Word embedding:

Before we run any machine learning algorithms on our data, we must turn our data into vectors using a word embedding technique. In text-mining, word embedding is transforming data from text into vectors of numbers. We use word embedding because the computer can only process numerical values. In this project, word embedding is used to learn and understand words and connections between words inside a corpus via vectors of words. This means we must first choose a specific technique to measure the importance of a term in a document.



Word2Vec:

Word2Vec is a word embedding process that creates vectors of words according to the similarity between the words. The computer does this by plotting the similar words closer to one another base on the vectorial values.

Word2Vec is a great tool for visualizing and understanding the connection between words in a corpus. Analyzing the plotted data can elicit an explanation on the words in the text and help us understand how certain words are perceived in the corpus.

Word2Vec is used globally by big companies that are trying to learn and understand more about their products and similarities between common search words. Word2Vec can also be used on non linguistic data bases. E.g., analyzation of search histories, analyzation of categorial features, Speech recognition.

The information needed for this part of the project is from the gensim word2vec model documentation. A link to the source is in the bibliography chapter.

Training the model:

Word2vec uses an unsupervised learning model that is trained via two neural networks: The Continuous Bag Of Words training method and the Skip Gram training method.

Using these methods allows us to create connections and train the model into better understanding the connections between words.

To train the model we must create sentences by transforming our corpus into a list of lists. To do so we must first understand how to define a sentence using a period.

The power of the period:

In any given language, the period defines when a sentence ends and another begins. Using this fact we can create a function that devides a document into a list of sentences by recording the placement of words that contain periods. To do this we must first turn our text into a list using the .split() function.

Before looking for periods we must understand that not every word that contains a period is the last word of a sentence. There are instances where periods can be found in the middle of a word, for example- an internet link. In order to combat this we must create a function that removes periods that are not in the end of a word. I Added another loop to the stop-words function from the previous chapter. The code for the for loop was the following:

| # Removing periods from the middle of words nswlist = no\_stop\_words.split() for i in range(0, len(nswlist)):  if '.' in nswlist[i]:  if (nswlist[i].count('.') > 1) and (nswlist[i][-1] == '.'):  nswlist[i].replace('.', '', (nswlist[i].count('.') - 1))  elif (nswlist[i].count('.') == 1) and (nswlist[i][-1] == '.'):  continue  else:  nswlist[i].replace('.', '')  # Adding a period to the last word of the list if needed if nswlist[-1][-1] != '.':  nswlist[-1] = nswlist[-1] + '.' no\_stop\_words = ' '.join(nswlist) |
| --- |

In the code above, I started by first removing periods from the middle of words. This was done by using an if function to see if a word contains a period. If it did, the next if function checks if the last character of a word was a period and if there were two or more periods. This makes it so the function has 4 different states it could be in:

1. If there are no periods, the function continues to the next iteration.
2. If there are two or more periods and one of them is at the end of the string, the function will remove all the periods except for the final one using the count parameter in the .replace function.
3. If there is only one period and it is at the end of the word, skip the current iteration.
4. Finally, if none of the above, the function removes all the periods with no regards to the placement in the string

Finally, after removing unnecessary periods we can use another function to create a list of lists.

The function for creating sentences out of the document is called createsentences and is implemented in the Word2Vec module I created:

| def createsentences(doc):  sentences = []  where = [0]  docl = doc.split()   # Create a list of indexes of words with periods  for i in range(0, len(docl)):  if '.' in docl[i]:  where.append(i)  where.append(i + 1)   # Create a list of sentences according to the indexes of words with periods  for i in range(0, int(len(where) / 2)+1, 2):  sentence = []  for j in range(where[i], where[i + 1]+1):  sentence.append(docl[j])  sentences.append(sentence)   # Remove periods from final sentences list  for i in range(0, len(sentences)):  sent = sentences[i]  for j in range(0, len(sent)):  sent[j] = sent[j].replace('.', '',)   return sentences |
| --- |

In the code above, I created a list of sentences in documents by recording the index number of each word that contains a period and the word after that. So for example, if the first sentence would end in word number 9, the function would add the numbers 9 and 10 to the list.

Then, another for loop would create a list of lists by adding the word between every two numbers.

Finally, I removed every period to ensure the word2vec model isn’t being trained with words containing periods.

CBOW:

The Continuous Bag Of Words, or CBOW for short, is a word2vec model imported from the gensim module by changing the sg parameter in the model initialization step. In this model the algorithm processes a sentence and then trains the model by predicting a missing word. The words that surround a certain term in a sentence are marked by weight (importance) and are taken more or less seriously depending on the distance from that specific term. For example:

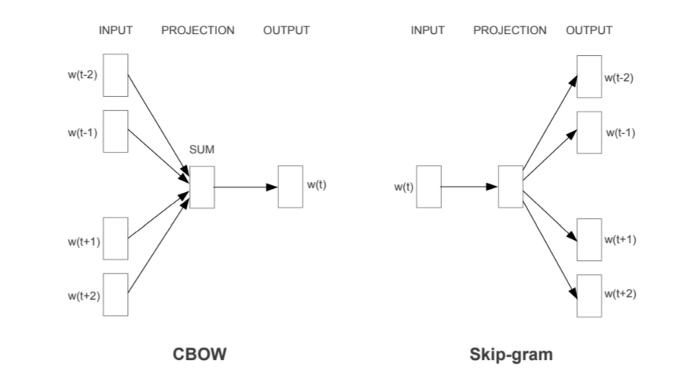
In the sentence, “I had a wonderful picnic today”, for the term “wonderful” the words “a” and “picnic” would have a higher weight (w(t-1) or w(t+1)) while the words “I” and “today” would have a lower weight.

The first layer of the CBOW model is called the Input layer. In this layer we insert an average of multiple context words like I have shown above. The next layer is called the Projection layer. The weights that are calculated before are applied in this layer. Finally we train the model using the output layer that returns the projection of the neural network.

SG:

The second word2vec model is the Skip-Gram model. Similar to CBOW, the SG model is a 3 layer neural network with an Input layer, a Projection layer, and an Output layer. Unlike the CBOW method, the SG method analyzes words by other words around them. With a term called n-gram we can define the number of words around a certain word we want the model to analyze. This is basically an opposite method to CBOW but will still help the model better understand language and the definitions of terms.

In the diagram below is a visual representation of both the CBOW model and the SG model. Both are efficient and were used to train my final model.



The code to initialize and train a new model using the sentences we created via the createsentences function is as follows:

| def createmodel(df):  print('create model:')  docs = df['processed\_body']  asentences = createsentences(docs.iloc[0])  model = models.Word2Vec(sentences=asentences, min\_count=0, vector\_size=200, workers=5, window=7, sg=1)  for i in tqdm(range(1, len(df))):  Asentences = createsentences(df['processed\_body'].iloc[i])  token\_count = len(Asentences)  model.train(corpus\_iterable=Asentences, word\_count=0, total\_examples=token\_count, epochs=10)  return model |
| --- |

In the code above we initialized the model using the gensim word2vec model. I learned how to use this code using the gensim word2vec documentation and word2vec demo.

In the function above we create a model variable for our model that we first train using a single document. In this step we set some parameters that can affect the success of the model.

The main parameters that are important to note are:

Min\_count- this parameter defines the number of times a certain words has to be included to be diagnosed. Unfortunately this parameter has made some errors and denied the model from initializing, that is why I set it to 0.

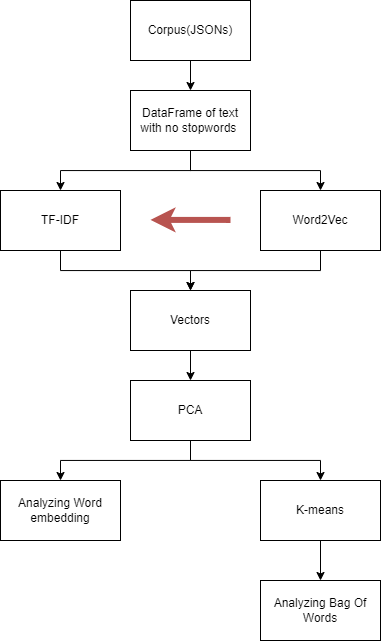
Vector\_size - this parameter is quite straight forward and simple to understand. This parameter defines the size of the vector space and the size of the output vectors that we can utilize in the model.

Window- for both the SG and the CBOW method, the window parameter defines the n-gram value. I.E., the number of words around the tested term that are taken into account when the model trains.

After initializing the model we only need to train it on every other document. This is simply done by running the train function and setting the corpus-iterable parameter to the list of sentences of the specific document.

Statistical measures:

After transforming the corpus into vectors of words by using the Word2Vec model, I will be applying another technique in Text-Mining, Statistical measures. Using this term, we can understand and create vectors for the documents meaning this step of the project is on par with word embedding and is another process that utilizes the corpus.



Statistical measures isn't a machine learning algorithm. But instead, it uncovers data and values of words inside documents using mathematical algorithms. If we look at the complexity of this term, we can see that it is not as advanced as word-embedding because there are no machine learning models used in this process.

Tf-Idf:

There are many techniques and algorithms that transform a document into a vector. I chose to work with the Tf-Idf algorithm because of its applicability. Using this technique is quite simple, and when looking at the mathematic algorithms, we can easily understand how it works.

Tf-Idf stands for Term frequency-Inverse document frequency and is split into two main algorithms:

Term frequency:

The importance of a word inside a single piece of text can be calculated via the followin equation:

where:

* is simply the number of times a specific term t occurs in document d.
* is the sum of the number of words in the d document.

The Term frequency of a document is fundamental. Thinking logically, this makes sense when you think about measuring the importance of a term in a document. The number of times a term appears divided by the number of words in a document measures how unique and frequent a term is.

The calculation of the Term frequency was done by creating a new function tf:

| def tf(df, uni):  tfdf = pd.DataFrame(index=uni)  print('tf:')  for i in tqdm(range(0, len(df))):  body = df['processed\_body'].iloc[i]  bodyl = body.split()  column = []  for word in uni:  count = bodyl.count(word)  column.append(count / len(bodyl))  index = str(i)  tfdf[index] = column  return tfdf |
| --- |

In the code above, I created a function in my TfIdf module. This function returns a data frame where each column is a word, and each row is a different document. It returns the Term-Frequency values according to the text and the word it is calculating.

The first thing I did was create a list of all the unique words using another function. The list of words I had was found using the following function:

| def uniquewords(df):  print('unique:')  for i in tqdm(range(0, len(df))):  body = df['processed\_body'].iloc[i]  bodyl = body.split()  unique = []  for word in bodyl:  if unique.count(word) == 0:  unique.append(word)  print(len(unique))  return unique |
| --- |

As you can see, the uniquewords function receives a data frame of all the processed documents. It adds a word to a list called unique if it is not already in the list. At the end of this process, the function returns the list.

I used this function once and then used the same list of unique words for the rest of my code.

The tf function receives two inputs. The df variable is the data frame of all the processed documents from the first chapter, and the uni variable is the list of unique words.

I then create a new data frame tfdf with an index of my uni list. The function goes into a for loop. In each iteration, the function counts the number of times a particular term is written and divides it by the number of words in that document.

At the end of the process, we are left with a data frame of all the Term-Frequency values. We will use this further in the code to calculate the Tf-Idf of each word in the corpus.

Document frequency:

We use document frequency to measure the importance of documents in an entire corpus. I.e., the number of documents the term appears in. The equation for Document frequency is very similar to the equation for Term frequency:

Where:

* is the number of occurrences of a term in the corpus.

The Document frequency is used in the Inverse document frequency equation. Once we have the number of term occurrences inside our corpus, we can finally calculate our IDF.

Inverse document frequency:

Inverse document frequency allows us to calculate how much information a term provides, i.e., if the word is used frequently across all the documents. The IDF is the importance of a word in your corpus. For instance, if the corpus were a single document, the IDF would be equal to 1.

The equation to find the IDF of the term t in the corpus D is as follows:

Where:

* is the number of documents in the corpus
* is the number of occurrences of the term in the corpus .

This equation isn’t complete because of two minor changes we have to make:

First, the number given to us by this equation can be so small that it becomes insignificant. To fight this we use the mathematic function “log”. Using log will scale the numbers higher and still keep the ratio between the values.

The equation with an added log will look like this:

Second, the function is built well but in the case that a word is not in the corpus the Document-Frequency will be 0, meaning we will have a calculation like so:

This equation is a math error and that is why we will add 1 to the denominator.

The final equation will like so:

The calculation of the Inverse-Document Frequency was done by creating a new function idfl:

| def idfl(df, uni):  idfdf = pd.DataFrame(index=uni)  col = []  print('idf:')  for word in tqdm(uni):  num = 0  for i in range(0, len(df)):  body = df['processed\_body'].iloc[i]  bodyl = body.split()  if(word in body):  num = num + 1  col.append(num)  idfdf['docf'] = col  idf = []  for i in idfdf['docf']:  n = len(df)  iidf = math.log(n / (i + 1))  iidf = abs(iidf)  idf.append(iidf)  idfdf['idf'] = idf  return idfdf |
| --- |

In the code above, I created a data frame idfdf with an index of the unique variable. Then I entered a for loop, and with each iteration, the function calculates the Document-frequency for each term.

After that, the list variable col is added to the idfdf data frame as the docf column. Finally, according to the first column, the function calculates the Inverse-Document Frequency for each term and adds it as the idf column.

Term frequency-Inverse Document frequency:

Finally, we have arrived at the final step of the word embedding of the data frame- Calculating the Term frequency-inverse Document frequency.

Calculating the Tf-Idf is very simple. And the equation for it is as follows:

Implementing Term frequency-Document inverse frequency into the code was done by creating the function TfIdftransform:

| def TfIdftransform(df):  unique = uniquewords(df)  tfidf = pd.DataFrame(index=unique)  tfdf = tf(df, unique)  idfdf = idfl(df, unique)  print('tfidf:')  for i in tqdm(range(0, len(df))):  ii = str(i)  col = []  for word in range(0, len(unique)):  tfi = tfdf[ii].iloc[word]  idfi = idfdf['idf'].iloc[word]  tfidfi = tfi \* idfi  col.append(tfidfi)  tfidf[ii] = col  return tfidf |
| --- |

In this piece of code, I initiated and utilized every other data frame from this chapter. At the beginning of the function, I created the tfidf data frame, which is the data frame we returned at the end of the code. Afterward, I initialized the unique list and the tfidf and idfdf data frames using the previous functions from this module.

Next, I ran a for loop, and with each iteration, I multiplied the idfdf column ‘idf’ of the current term by the df value of that exact term in a specific document. I.e., I calculated the Term frequency-Inverse document frequency by multiplying the two values.

Finally, once calculated, I added the value to a column and that column to the data frame tfidf.

Conclusion for chapter 2:

Word embedding and statistical measures are text mining algorithms used in many global applications.

Word Embedding is the process of transforming words into vectors that represent the meaning and connections between multiple words. The space in which these vectors are stored is called the Vector space or more commonly the embedding space. To create these vectors we must use a word embedding algorithm called word2vec and after training a machine learning algorithm we can create a simpler word embedding space using the gensim module.

Tf-idf is the process of interpreting text into mathematical values. In turn, these values may be represented as vectors. These vectors may represent several features of the text. The Tf-idf technique is a statistical measuring technique that uses mathematical calculations in order to

assess the importance of a word in a certain text. The concept of Tf – the

term frequency feature when compared with the idf – inverse document

frequency is an important part of learning of the importance of a specific

word or term within the assessed text. While Tf-idf is divided into two

different values; the Tf and idf their individual contribution is not equivalent

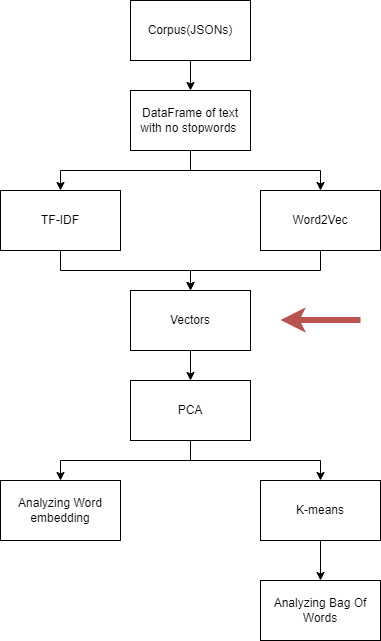
to their combined value. When these two values are assessed

independently one may learn of the context but when utilized in tandem

they represent the quality and significance of a term within a said text in an

avid fashion.

Text-mining is very important in todays day and age. Word embedding and Statistical measuring are two subtopics of an even larger genre in machine learning.



# Chapter 3 Unsupervised learning

An introduction to Machine learning:

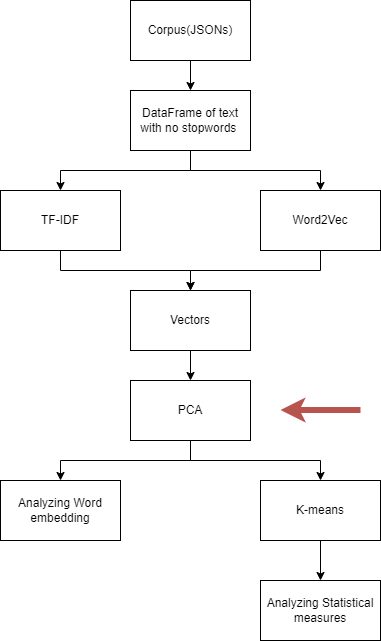
In unsupervised learning we tend to use neural network to create connections between objects using a method that is based on the way the human mind works. The mind is based on the function of neural networks which are comprised of neurons. Neural networks are based on the fact that you can actually cause computers to work like the human brain. The human brain in turn uses the neural networks as a way to learn and by performing asks over and over again you can end up with a smarter brain. The same is true for machine learning. In machine learning we use neural networks to find and create patterns in the data. The machine learning algorithm looks for certain patterns as well as acts to find certain solutions to problems it detects. The key attribute of machine learning is the capability to learn or perform tasks without having to explicitly program the computer. A well built machine learning algorithm can find solutions to real problems. As is inferred from the name machine learning it is about having the computer learn by itself. The applications for this technology are endless and it is being implemented in a growing number of fields.

unsupervised learning:

In machine learning, we have three main learning algorithms: Supervised Learning, Unsupervised Learning, and Deep Learning. Unsupervised learning assigns data that has not been organized nor labeled into multiple groups that give the data more "structure." For example, A weatherman has a weather database from all around Israel. The database consists of the temperature, the moisture percentage, the area in Israel the data was documented in, and more.  
In a supervised learning algorithm, the point of the machine would be to find a connection between a single attribute and the rest of the data. For example, a programmer chose to write a program that takes the temperature of a particular area and the rest of the dataset and returns the success rate of finding the temperature of that area depending on the rest of the data.  
In an unsupervised learning algorithm, the point would be to find similarities between multiple attributes without picking a single feature to base the identification on. For example, A programmer decided to find groups of data without dividing the data the weatherman collected. The weatherman gave the programmer the database, and the programmer started working. He wrote the code, and the machine returned groups of similarities in the data.

PCA:

Principal component analysis or PCA for short, is a dimensionality reduction process that transforms high dimensional data sets into lower-dimensional data frames. Using PCA is an important step in lowering dimensions and unimportant features. When trying to visualize more than three columns of data we run into a problem. Because we are living in the third dimension, we do not have any engines that can visualize more than three dimensions. Using Principle Component Analysis I will reduce the dimensionality of my data frame while saving 95% of the variance.



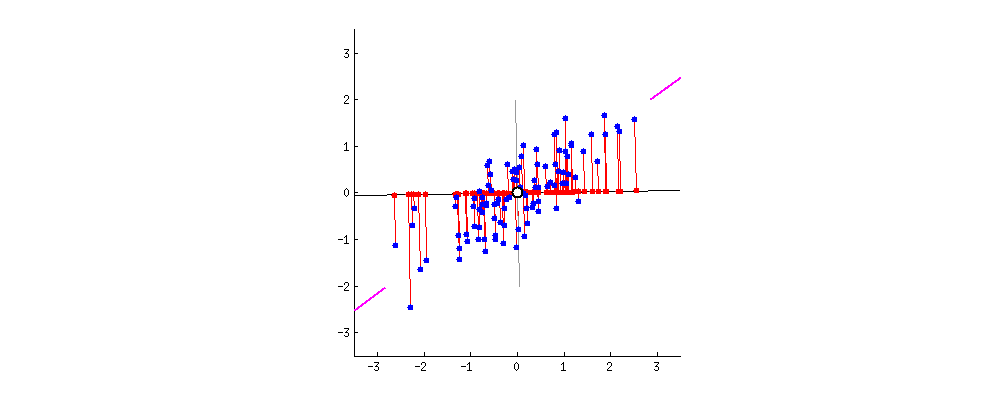
Variance:

The variance is the relation of the mean and two principal components in correlation to one another. I.e., the relation between two features in comparison to the average value. The percent of variance represents the percent of information explained by the principal components. The higher the variance, the more data is explained.

We use PCA to lower the dimensionality of the data and for removing unimportant features. To calculate the amount of variance for the first principal component we can use the following equation:

Where the Xm and Ym are the coordinates for the closest point on the mean to another data point. And the xd and yd are the coordinates for that specific data point.

The mean of the variables can be visualized as a line that goes through a centroid of a plot. The process can be visualized well in the figure below:



Using the PCA module is simple, by downloading the sklearn.decompositioning module and importing the built-in PCA function we can create a new PCA module that removes, as many features as possible while still saving 95% of the variance. The code to lower the dimensionality to 95% variance is:

| myPCA = PCA(n\_components=0.95) nda = myPCA.fit\_transform(df) ndf = pd.DataFrame(nda) return ndf |
| --- |

Now we have reduced the dimensionality of the data set while still keeping the percent of the variance we would like to explain. And we can finally clusterize the data.

K-means:

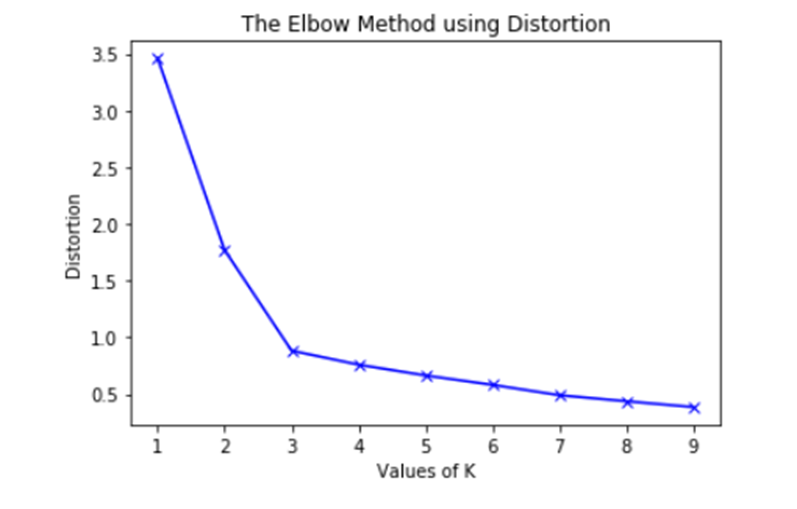
In unsupervised learning, there are many ways to explore clustered data, one of these techniques is called the "K-means function". What this function does is it divides the data into several groups called "clusters". The number of cluster points is has to be determined prior to running the function.

The process of using the K-means method is as follows:

1. Using an elbow graph to visualize and deduce the pivotal number of clusters
2. Running K-means function using the optimal number of clusters.
3. Visualizing the data using a plot.

The elbow method:

To find the number of cluster points that are best suited for our K-means function, we need to use the elbow method. The elbow method creates a two-dimensional graph that measures the ideal number of clusters. The first dimension is the number of cluster points and the second dimension is the “Distortion” which measures the variation that the number of cluster points has on the precision of the cluster.

An elbow graph would look like the following example:

In the graph above we see the two dimensions previously mentioned. Now that we have an elbow graph, we’ll look for the value of K (number of clusters) which is the point at which additional clusters will have a diminished effect on the elbow graph.

We can create an elbow graph by running the KMeans function from the sklearn.cluster module. Then creating a list and adding the distortions that are created by the k-means function for a range of numbers.  
We then plot the list into a plt plot where the X axis is the number of clusters and the Y axis is the amount of distortion explained.

To calculate the distortion we must download a scipy.spatial.distance module and import the cdist function. This will help us calculate the distances between the nearest cluster center and the X and Y of a data point. The function to calculate the distortion is as follows:

Where XC and YC are lists of the coordinates of every cluster center and xd and yd are the coordinates for the datapoints. The distortion sum of all the distances between the data points and the cluster closest to a specific data point.

To find the distortion of a number of clusters we divide the sum of all distortions by the number of data points. Leaving us with the final equation:

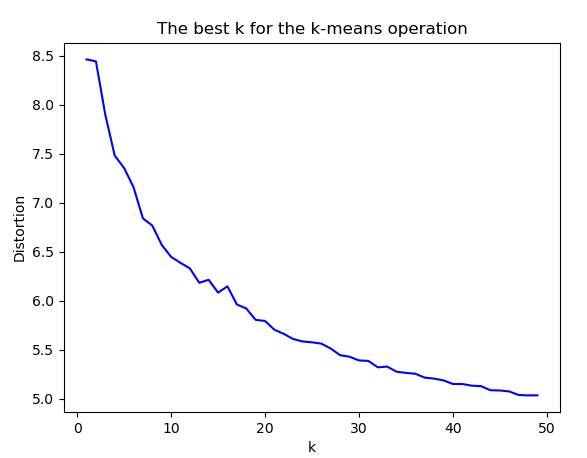
Where is the number of data points in the corpus and is the sum of the distortions.  
The code for creating the elbow graph is as follows:

| distortions = [] K = range(2, 100) for k in K:  k\_means = KMeans(n\_clusters=k).fit(n)  k\_means.fit(n)  distortions.append(sum(np.min(cdist(n, k\_means.cluster\_centers\_, 'euclidean'), axis=1)) / tfidf.shape[0]) |
| --- |

Now that we know the distortions of every cluster between 2 and 100 we will create a seaborn plot that holds the elbow graph of the corelation between the distortions and the number of clusters (K):

| X\_line = [K[0], K[-1]] Y\_line = [distortions[0], distortions[-1]]  # Plot the elbow plt.plot(K, distortions, 'b-') plt.xlabel('k') plt.ylabel('Distortion') plt.title('The best k') plt.show() |
| --- |

The elbow graph ended up looking like this:



I decided to run the clustering process on 30 clusters because it looks as though that is the value of K at which the incline of the graph started to really moderate.

Once we know the optimal number of clusters for our data, we can run the K-means function imported from sklearn. Using the following code:

| kmeans = KMeans(n\_clusters=30, random\_state=42) y\_pred = kmeans.fit\_predict(n) parsedclean['y'] = y\_pred |
| --- |

In the code above, we apply the trained k-means module on our data and save it inside the text data frame in the y column.

Using PCA to obtain a 2-dimensional plot:

While PCA was used before to decrease the dimensionality of the Embedding space while keeping a 95 percent variance. While this step was crucial to make sure there aren’t any unwanted vectors in our K-means function, it still isn’t enough. While dimensionality reduction helped us improve the accuracy of the model by removing minimal variance, plotting the vectors into a 2-dimensional plot is done by reducing the dimensions of the Tf-Idf data frame to 2. To do this we only need to change the n\_components parameter in the PCA model.

The code for minimizing the embedding-space to 2 using the PCA model is:

myPCA = PCA(n\_components=2)  
nda = myPCA.fit\_transform(df)  
ndf = pd.DataFrame(nda)  
return ndf

In the code above I set the n\_components parameter of the PCA component to 2. And while this parameter can be set to a float and will keep the float amount of variance, it can also be set to an integer and will transform the data into an embedding space of the given integer.

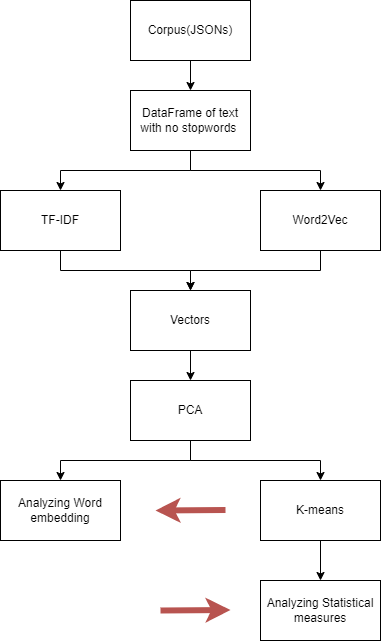
Finally we have a data frame of two columns, each representing one of the most valuable vectors and the two axis of the final plot.

How this is used in Word-Embedding and Statistical measures:

In both Statistical measures and word embedding we want to lower a high dimensional embedding space into a 2-dimensional plot by using PCA and other machine learning techniques. Unlike Statistical measures, Word-Embedding only needs to go through a dimension reduction process. On the other hand, in tf-idf we have many vectors that represent different documents and terms in those documents, meaning there will be more processes needed.

That is why for the statistical measures part of the project we will run PCA for reducing dimensions under 95% of variance, cluster the data using K-means, and finally run another PCA function to lower the data to a 2-dimensional embedding space.  
And for Word-embedding we only need to run the PCA process to lower the vectors into a 2-dimensional embedding space.

# Chapter 4 - analyzing the data



How I Analyzed the data:

Using seaborn for plotting the data we can create a visual representation of both the Tf-Idf dataframe and the Word2Vec model. The two plots are simply a representation of the two dimensions found via PCA in the final dimensionality reduction of each embedding space.  
The code for visualizing the Word2Vec model was simple:

| x\_vals, y\_vals, labels = Word2Vec.reduce\_dimensions(w2vmodel)  def plot(x\_vals, y\_vals, labels):  plt.figure(figsize=(12, 12))  plt.scatter(x\_vals, y\_vals)   indices = list(range(len(labels)))  for i in indices:  plt.annotate(labels[i], (x\_vals[i], y\_vals[i]))   plot(x\_vals, y\_vals, labels)  plt.show() |
| --- |

In the code above I used the Matplotlib plt class to plot the x values and y values of the 2-dimensional word embeddings and the list of words labels for understanding what data point we are looking at.

The code for visualizing the Tf-Idf model was a bit more complex, we first had to show the plot with the datapoints that were not clustered yet. This was simply done with the following code:

| pca = PCA(n\_components = 2)  X\_embedded = pca.fit\_transform(X\_std)   import seaborn as sns  # sns settings  sns.set(rc={'figure.figsize': (15, 15)})   # colors  palette = sns.color\_palette("bright", 1)   # plot  sns.scatterplot(X\_embedded[:, 0], X\_embedded[:, 1], palette=palette)  plt.title('PCA with no Labels')  plt.show() |
| --- |

In the code above I plotted the datapoints without clustering them using the sns.scatterplot function.

Finally, after running this code, we show the final result of the plot with clusters. Using the following code:

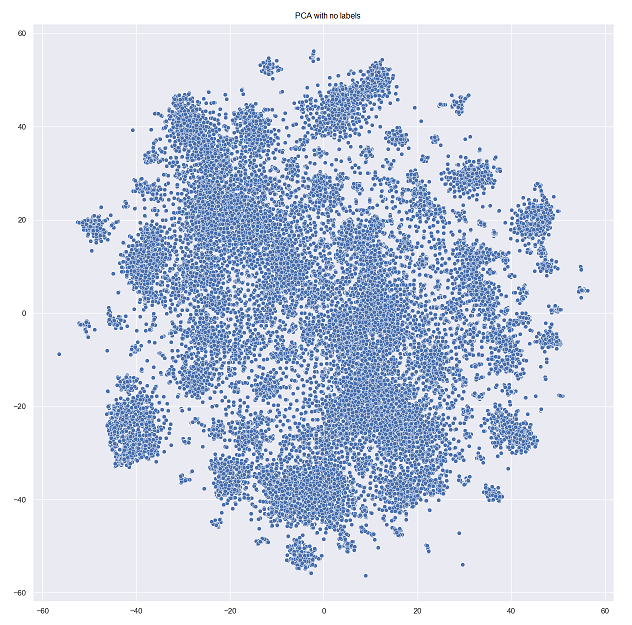
| from matplotlib import pyplot as plt  import seaborn as sns   # sns settings  sns.set(rc={'figure.figsize': (13, 9)})   # colors  palette = sns.hls\_palette(30, l=.4, s=.9)   # plot  sns.scatterplot(X\_embedded[:, 0], X\_embedded[:, 1], hue=y\_pred, legend='full', palette=palette)  plt.title('PCA with Kmeans Labels')  plt.show() |
| --- |

In the code above I used the same sns function as before plotting the two principal component axise. The main difference here is the added hue and palette parameter. The hue and palette parameters are the clustering and colors of the said clusters in the plot. This is what determined the color and cluster of each datapoint.

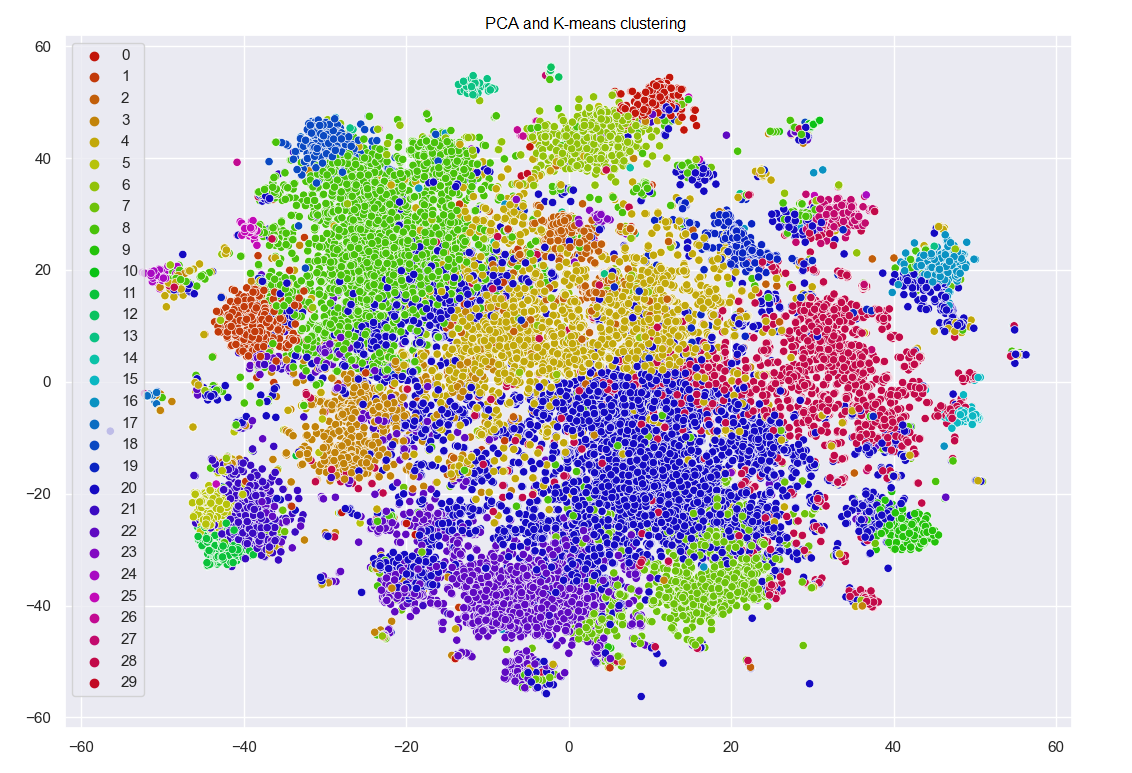
Finally, after many hours of coding and understanding the machine learning algorithms, we can visualize the plots and come up with interesting findings about the data.

Tf-Idf analyzation:

The fist plot I created was a simple plot that visualized every document as a different data point:



In the following plot we will see a variety of clusters in 30 different colors. The legend on the side is the guide of color for each cluster:

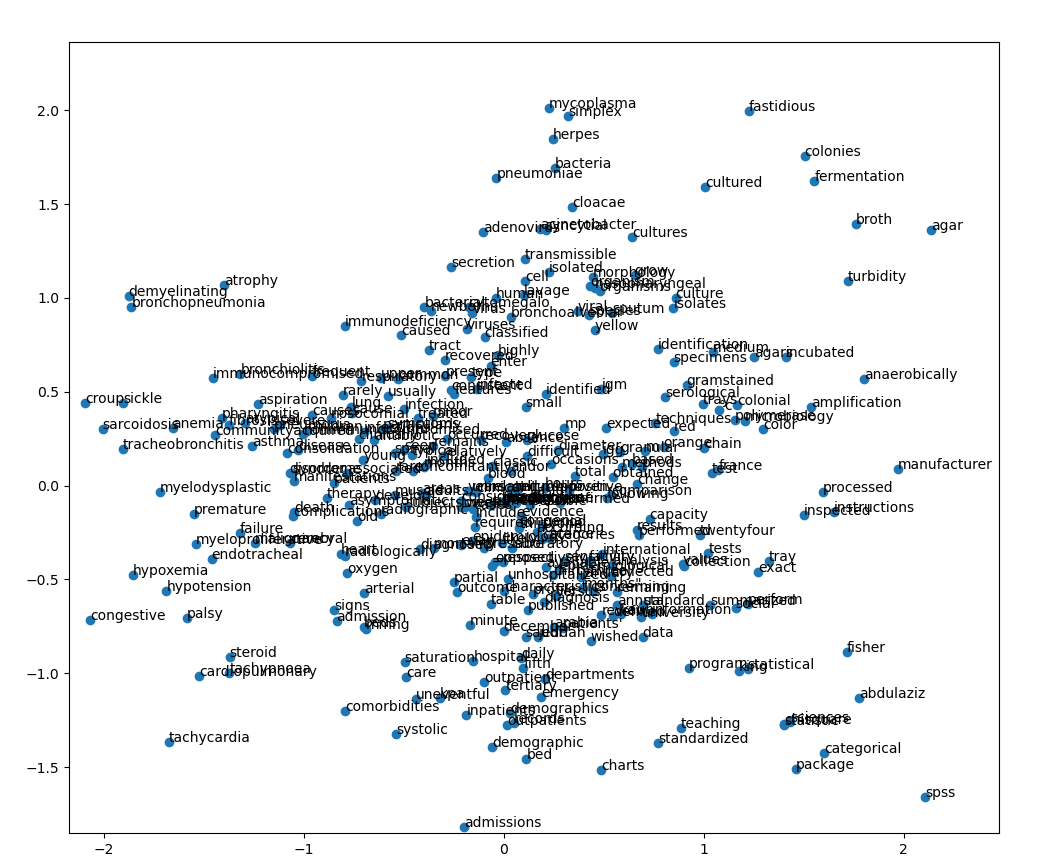


What we can learn:

As can be seen from the above image the TF-IDF dataframe processing resulted in clusters which were found to represent the intricate relations within the Corpus and the relevance of each document as represented within the clusters. The clusters of documents processed here represent the connection between the term frequency and its relation with the inverse document frequency. This graphic representation aptly shows the clusters of documents and how clusters can be different in size, shapes, and even range. When all is said and done, showing that the data is clusterable is very important and in fact proves that the process was complete fairly well.

Word2Vec analyzation:

While the Tf-Idf plot contains clusters of data points and measures the documents. The Word2Vec plot is the 2-dimensional embedding space that represents similarities between words and terms:



While the plot above is very dense in text and terms, we can still see that some words in the graph make sense being near one another. To understand the connections between words better I used the gensim function wv.most\_similar() to find the pairs of words that are most similar to each other:

| def presenthigher(num):  for i in labels:  sims = w2vmodel.wv.most\_similar(str(i), topn=1)  if sims[0][1]>=num:  print(f'{i}: {sims}') |
| --- |

In the code above, I used the wv.most\_similar() function to find the similarities that are 50 percent similar at least. This was done by finding the percentage of similarities between a term and it’s closest relative and then comparing the similarity percentage to 0.5.

Pairs of words that had a higher similarity percentage than 0.5 and are interesting to analyze:

* ‘Human’ and ‘virus’ for 0.548 is an interesting pair. The word embedding code found that Human and Virus are similar which means they are connected in some way. As a human who lived through 2021 I can definitely say there is a connection between these words.
* ‘syncytial’ and ‘adenovirus’ for 0.575. According to the CDC Syncytial is a respiratory virus that can cause mild, cold like symptoms. On the other hand, according to the CDC, Adenoviruses are common viruses that cause a range of illness. This pair is very interesting, according to the results they are either similar in meaning or in usage. By the definitions of the CDC they are quite similar signifying the that the program diagnosed the similar meanings as pairs.
* ‘Anaerobically’ and ‘agars’ for 0.666. These words tend to come in a pair and have a combined meaning of a medical term. Anaerobic Agar is a medium for the cultivation of anaerobic bacteria. This finding signifies that the definitions of these words are close, meaning that there are some pairs of words that are not similar in meaning but are usually found around the same part of the sentence.
* ‘Syndrome’ and ‘severe’ for 0.527. Much like Anaerobically and agars, Syndrome and severe are two words that were simply used frequently and commonly together. We can make an assumption and say that in the pandemic there were many severe syndromes and diseases.
* ‘Instructions’ and ‘manufacturer’ for 0.691. This pairing shows that not only the meaning of words were understood by the model but also words that are associated with one another like manufacturer and instructions.

Looking at the long list of pairings, i’m happy it all makes sense. Finding similarities are fun and that is why I added a full list of the similarities above 50 percent below.

A full list of the similarities that are above 50 percent:

* pneumoniae: [('mycoplasma', 0.6056060791015625)]
* virus: [('human', 0.5482456684112549)]
* bronchiolitis: [('croup', 0.5012776851654053)]
* agar: [('broth', 0.6728885769844055)]
* syndrome: [('severe', 0.5270587801933289)]
* cultures: [('culture', 0.5192334055900574)]
* kauh: [('abdulaziz', 0.8110352158546448)]
* culture: [('cultures', 0.5192334651947021)]
* arabia: [('saudi', 0.8121945261955261)]
* saudi: [('arabia', 0.8121945261955261)]
* manufacturer: [('instructions', 0.6918728351593018)]
* mycoplasma: [('pneumoniae', 0.6056060791015625)]
* tray: [('trays', 0.5971413254737854)]
* complications: [('failure', 0.5085536241531372)]
* broth: [('agar', 0.6728885769844055)]
* oxygen: [('saturation', 0.5203846096992493)]
* cultured: [('cultures', 0.5147014260292053)]
* minute: [('mmhg', 0.5074703693389893)]
* instructions: [('manufacturer', 0.6918726563453674)]
* disease: [('children', 0.5152879357337952)]
* aslo: [('microbio', 0.7241846323013306)]
* igm: [('igg', 0.5515386462211609)]
* igg: [('igm', 0.5515386462211609)]
* gramstained: [('anaerobically', 0.5791090726852417)]
* tracheobronchitis: [('croup', 0.5492058992385864)]
* croup: [('tracheobronchitis', 0.5492058992385864)]
* university: [('abdulaziz', 0.5129455924034119)]
* signes: [('aslo', 0.6701815724372864)]
* microbio: [('aslo', 0.7241847515106201)]
* lavage: [('bronchoalveolar', 0.8041881918907166)]
* polymerase: [('chain', 0.5919760465621948)]
* chain: [('polymerase', 0.5919759273529053)]
* king: [('abdulaziz', 0.6774079203605652)]
* jeddah: [('abdulaziz', 0.5609074831008911)]
* abdulaziz: [('kauh', 0.8110352754592896)]
* bronchoalveolar: [('lavage', 0.8041882514953613)]
* nasopharyngeal: [('sputum', 0.5157318115234375)]
* sputum: [('nasopharyngeal', 0.5157317519187927)]
* months": [('microbio', 0.5719894170761108)]
* fastidious: [('gramstained', 0.5164048671722412)]
* agars: [('anaerobically', 0.6663967967033386)]
* failure: [('complications', 0.5085536241531372)]
* congenial: [('aslo', 0.5184872150421143)]
* human: [('virus', 0.5482456088066101)]
* children: [('disease', 0.5152878761291504)]
* fisher: [('chisquare', 0.6323066353797913)]
* adenovirus: [('syncytial', 0.5756772756576538)]
* syncytial: [('herpes', 0.5865219235420227)]
* acinetobacter: [('cloacae', 0.5271605849266052)]
* herpes: [('simplex', 0.8063705563545227)]
* cloacae: [('acinetobacter', 0.5271605849266052)]
* simplex: [('herpes', 0.8063704967498779)]
* myelodysplastic: [('myeloproliferative', 0.6093348264694214)]
* myeloproliferative: [('myelodysplastic', 0.6093347668647766)]
* exact: [('fisher', 0.6180357336997986)]
* trays: [('tray', 0.5971412658691406)]
* incubated: [('°c', 0.586394190788269)]
* anaerobically: [('agars', 0.6663967967033386)]
* °c: [('incubated', 0.586394190788269)]
* chisquare: [('fisher', 0.6323065757751465)]
* package: [('spss', 0.6394980549812317)]
* statistical: [('spss', 0.6407709121704102)]
* sciences: [('spss', 0.5036577582359314)]
* spss: [('statistical', 0.6407709121704102)]
* categorical: [('chisquare', 0.6226192116737366)]
* hypotension: [('tachycardia', 0.5544597506523132)]
* severe: [('syndrome', 0.5270587801933289)]
* tachycardia: [('tachypnoea', 0.5871831774711609)]
* tachypnoea: [('tachycardia', 0.5871832370758057)]
* systolic: [('mmhg', 0.6460161209106445)]
* saturation: [('mmhg', 0.5950636267662048)]
* mmhg: [('systolic', 0.6460161209106445)]
* arterial: [('mmhg', 0.5143253803253174)]
* kpa: [('systolic', 0.5029351115226746)]

# Conclusions and what I learned

In this project I proved and explained some of thebasic text-mining algorithms using a covid-19 literature data set. The main focus was to prove that I can learn and write a python script that is capable of understanding human language using the text-mining algorithms.

This was a difficult process and took me some time to understand, I had a large amount of fun writing the code and revising what I have learned in the Machine learning course I took at the start of the project.

I also had Ilan my mentor as a large amount of help when writing the project. He gave me feedback on some of my mistakes and overall helped me understand the things I learned to a deeper level.

Other than Ilan I had some help from my father who is a medical doctor. He was part of the inspiration for the Covid-19 corpus and i’m sure I would have had a totally different experience without him.

In this project I learned how to preprocess data and transform text into a data frame capable of being vectorized, I learned about different types of text-mining techniques and methods to use data and applicate a processed corpus, I understood how dimensionality reduction works and how different cluster numbers can greatly affect the overall outcome of the machine learning algorithm. I learned that working on a deadline can be stressful and can damage the learning process if done incorrectly, and most importantly, I learned more about my interest and the profession I would like to take part in the future.

The experience and knowledge I obtained in this project can not come second hand to any other source. Working on such a big project can be over whelming, but I can definitely say that it was an experience unlike any other.

# Bibliography:

The majority of equations and understandings in this project were found by taking part in the machine learning course of stanford university Taught by Andrew Ng:

<https://www.coursera.org/learn/machine-learning/home/welcome>

For creating my Tf-Idf functions used the following article that had an indepth explanation for the equations:

TF-IDF from scratch in python on a real-world dataset. By William Scott MSc of IIITD:

<https://towardsdatascience.com/tf-idf-for-document-ranking-from-scratch-in-python-on-real-world-dataset-796d339a4089>

Stemming vs Lemattization by Aditya Beri:

<https://towardsdatascience.com/stemming-vs-lemmatization-2daddabcb221>

Json documentation:

<https://www.json.org/json-en.html>

Word2Vec demo:

<https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html>

Word2Vec gensim documentation:

<https://radimrehurek.com/gensim/models/word2vec.html>

Seaborn documentations:

<https://seaborn.pydata.org/>

# CODE

Main module:

| import re import gensim.models.keyedvectors import pandas as pd import tqdm.autonotebook import matplotlib.pyplot as plt import random import PCAandKmeans import Parse from tqdm import tqdm import Json\_reader import Word2Vec import TfIdf import matplotlib.pyplot as plt from sklearn.decomposition import PCA from sklearn.preprocessing import StandardScaler import numpy as np  if \_\_name\_\_ == '\_\_main\_\_':  # setting the settings for pandas  pd.set\_option('display.max\_rows', 1000)  pd.set\_option('display.max\_columns', 1000)  pd.set\_option('display.width', 2000)   # loading the data and creating a dataframe out of the csv file in the folder:  path = 'C:/Users/User/Desktop/finalProj/metadata/'  json\_path = path + 'document\_parses/pdf\_json/'  csv\_path = path + 'metadata.csv'  metaDf = pd.read\_csv(csv\_path)  metaDf.dropna(axis=0, how='all', subset=['pdf\_json\_files'], inplace=True)  a = f'{json\_path}d1aafb70c066a2068b02786f8929fd9c900897fb.json'  creating a list of all filepaths:  Jp = []  for i in range(0, len(metaDf)):  json\_id = metaDf['pdf\_json\_files'].iloc[i]  Jp.append(f'{path}{json\_id}')   def containswords(body):  if re.search('[a-zA-Z]', body):  return True  else:  return False  # using the read class we created for a new dataframe  j = 0  t = 100000  a = []  for i in tqdm(Jp, total=t):  if (t == j):  break  try:  data = Json\_reader.Read(i, metaDf)  except Exception:  continue  if len(data.body) == 0:  continue  if not containswords(data.body):  continue  id = data.id  abstract = data.abstract  body = data.body  doi = data.doi  url = data.url  author = data.author  title = data.title  row = [id, abstract, body, doi, url, author, title]  a.append(row)  j = j + 1  clean = pd.DataFrame(columns=['id', 'abstract', 'body', 'doi', 'url', 'author', 'title'], data=a)  clean.to\_csv(path\_or\_buf='D:\FinalProj/clean100000.csv')  # clean=pd.read\_csv(filepath\_or\_buffer='D:\FinalProj/clean.csv')  print(clean['body'].iloc[21278])  print(clean['body'].iloc[21277])  parsedclean = Parse.parsedata(clean)  parsedclean.to\_csv(path\_or\_buf='D:\FinalProj/parsedclean100000.csv')  parsedclean=pd.read\_csv(filepath\_or\_buffer='D:\FinalProj/parsedclean100000.csv')  print(len(parsedclean))  print(parsedclean['processed\_body'].iloc[0])   unique = TfIdf.uniquewords(parsedclean)  w2vmodel = Word2Vec.createmodel(parsedclean)  w2vmodel = gensim.models.Word2Vec.load('D:\FinalProj/word2vec100000.model')  labels = np.asarray(w2vmodel.wv.index\_to\_key)  def presenthigher(num):  for i in labels:  sims = w2vmodel.wv.most\_similar(str(i), topn=1)  if sims[0][1]>=num:  print(f'{i}: {sims}')   presenthigher(0.5)    x\_vals, y\_vals, labels = Word2Vec.reduce\_dimensions(w2vmodel)  def plot(x\_vals, y\_vals, labels):  plt.figure(figsize=(12, 12))  plt.scatter(x\_vals, y\_vals)   indices = list(range(len(labels)))  for i in indices:  plt.annotate(labels[i], (x\_vals[i], y\_vals[i]))   plot(x\_vals, y\_vals, labels)  plt.show()  w2vmodel.save("D:\FinalProj/word2vec100000.model")   #The tfidf part  tfidf = TfIdf.TfIdftransform(parsedclean)  tfidf = tfidf.T  tfidf.to\_csv(path\_or\_buf='D:\FinalProj/tfidf100000.csv')  tfidf = pd.read\_csv(filepath\_or\_buffer='D:\FinalProj/tfidf100000.csv')  print(tfidf.head(10))  print(tfidf.info())  #  X\_std = StandardScaler().fit\_transform(tfidf)  pca = PCA(n\_components=20)  principalComponents = pca.fit\_transform(X\_std)  features = range(pca.n\_components\_)   from sklearn.cluster import KMeans  from scipy.spatial.distance import cdist   # run kmeans with many different k  distortions = []  K = range(1, 50)  for k in K:  k\_means = KMeans(n\_clusters=k).fit(principalComponents)  k\_means.fit(principalComponents)  distortions.append(sum(np.min(cdist(principalComponents, k\_means.cluster\_centers\_, 'euclidean'), axis=1)) / tfidf.shape[0])   X\_line = [K[0], K[-1]]  Y\_line = [distortions[0], distortions[-1]]   # Plot the elbow  plt.plot(K, distortions, 'b-')  plt.xlabel('k')  plt.ylabel('Distortion')  plt.title('The best k for the k-means operation')  plt.show()  #  k = 30  kmeans = KMeans(n\_clusters=k, random\_state=42)  y\_pred = kmeans.fit\_predict(principalComponents)  parsedclean['y'] = y\_pred  pca = PCA(n\_components = 2)  X\_embedded = pca.fit\_transform(X\_std)   import seaborn as sns  # sns settings  sns.set(rc={'figure.figsize': (15, 15)})   # colors  palette = sns.color\_palette("bright", 1)   # plot  sns.scatterplot(X\_embedded[:, 0], X\_embedded[:, 1], palette=palette)  plt.title('PCA with no Labels')  plt.show()   from matplotlib import pyplot as plt  import seaborn as sns   # sns settings  sns.set(rc={'figure.figsize': (13, 9)})   # colors  palette = sns.hls\_palette(30, l=.4, s=.9)   # plot  sns.scatterplot(X\_embedded[:, 0], X\_embedded[:, 1], hue=y\_pred, legend='full', palette=palette)  plt.title('PCA with Kmeans Labels')  plt.show() |
| --- |

Parse module:

| import re from langdetect import detect from tqdm import tqdm   def parsedata(df):  tempdf = df  checkLanguage(tempdf)  a = []  for i in tqdm(tempdf['body']):  nsw = stopwordreduce(i)  a.append(nsw)  tempdf['processed\_body'] = a  tempdf = tempdf[tempdf['language'] == 'en']  return tempdf   def checkLanguage(df):  langs = []  for i in tqdm(df['body']):  a = i.split(' ')  if len(a) >= 50:  try:  lang = detect(" ".join(a[:50]))  except Exception:  lang = 'null'  elif len(a) > 0:  try:  lang = detect(" ".join(a))  except Exception:  lang = 'null'  langs.append(lang)  df['language'] = langs  def stopwordreduce(body):  # the spacy stop-word list:  # list of stop-words that will be taken out of the body  # the list can be found on https://machinelearningknowledge.ai/tutorial-for-stopwords-in-spacy/  stopwords = ['really', 'sometimes', 'go', 'since', 'whither', 'they', 'its', 'them', 'well',  'meanwhile', 'seems', 'and', 'latterly', 'regarding', 'somehow', 'sixty', 'whole', 'anyway', 'else',  'few', 'beside', 'to', 'namely', 'someone', 'see', 'moreover', 'wherein', 'for', 'former', 'bottom',  'it',  'next', 'six', 'along', 'once', 'might', 'whenever', 'below', 'another', 'yourself', 'each', 'just',  'ourselves',  'everyone', 'any', 'across', 'get', 'that', 'eight', 'we', 'which', 'therefore', 'may', 'keep',  'among', 'give',  'such', 'are', 'indeed', 'everywhere', 'same', 'herself', 'yourselves', 'alone', 'were', 'was', 'take',  'seem',  'say', 'why', 'show', 'between', 'during', 'elsewhere', 'or', 'though', 'forty', 'made', 'others',  'whereafter', 'formerly', 'several', 'via', 'does', 'please', 'three', 'also', 'fifty', 'afterwards',  'noone',  'no one', 'do', 'perhaps', 'further', 'I', 'beforehand', 'myself', 'empty', 'yet', 'thereby', 'been',  'both', 'never',  'put', 'without', 'him', 'a', 'nothing', 'thereafter', 'make', 'then', 'whom', 'must', 'sometime',  'against', 'through', 'being', 'four', 'back', 'become', 'our', 'himself', 'because', 'anything',  'nor',  'therein', 'due', 'until', 'own', 'most', 'now', 'while', 'of', 'only', 'am', 'itself', 'too',  'nobody', 'if',  'one', 'whereas', 'twelve', 'together', 'can', 'who', 'even', 'be', 'she', 'besides', 'herein', 'off',  'last', 'no', 'whereupon', 'the', 'thru', 'out', 'hereupon', 'by', 'us', 'already', 'became', 'here',  'hers',  'onto', 'beyond', 'down', 'enough', 'did', 'some', 'over', 'serious', 'quite', 'move', 'around',  'nowhere',  'amongst', 'but', 'so', 'wherever', 'twenty', 'often', 'part', 'again', 'where', 're', 'within', 'at',  'yours',  'front', 'unless', 'could', 'anyone', 'third', 'whatever', 'doing', 'nevertheless', 'before', 'rather',  'fifteen', 'her', 'me', 'thereupon', 'mostly', 'throughout', 'hence', 'mine', 'ten', 'hundred', 'nine',  'call',  'when', 'about', 'will', 'whereby', 'this', 'upon', 'you', 'should', 'always', 'themselves', 'not',  'has',  'behind', 'on', 'anywhere', 'side', 'their', 'hereby', 'latter', 'after', 'none', 'these', 'name',  'every',  'although', 'however', 'he', 'becoming', 'how', 'whose', 'still', 'hereafter', 'whether', 'towards',  'more', 'everything', 'whoever', 'seemed', 'cannot', 'up', 'otherwise', 'in', 'would', 'under', 'done',  'thence', 'whence', 'seeming', 'either', 'other', 'with', 'into', 'amount', 'five', 'much', 'except',  'his', 'thus',  'what', 'almost', 'becomes', 'least', 'ever', 'above', 'is', 'first', 'there', 'somewhere', 'top',  'than', 'have',  'toward', 'per', 'all', 'ours', 'full', 'anyhow', 'as', 'many', 'various', 'your', 'had', 'eleven',  'from', 'something',  'less', 'those', 'an', 'two', 'my', 'very', 'neither']  # list of strings that will be taken out of words  parts\_stopwords = ["'d", "'m", "n't", "'ve", "'re", "'s", "'ll", ",", "(", ")", "[", "]", "-", "/", ":", "%", '•',  ';', '(r)', '>', '∝', '<']  # my custom word list:  custom = ['doi', 'preprint', 'copyright', 'peer', 'reviewed', 'org', 'https', 'author', 'figure',  'rights', 'reserved', 'permission', 'used', 'using', 'biorxiv', 'medrxiv', 'license', 'fig', 'fig.',  'al.', 'Elsevier', 'PMC', 'CZI', 'ago', 'de', 'com']  nums = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']  parts = parts\_stopwords + nums  stopwords = stopwords + custom  no\_stop\_words = body.lower()   # Reduction of parts of other words  for i in parts:  no\_stop\_words = no\_stop\_words.replace(str(i), '')   # Removing periods from the middle of words  nswlist = no\_stop\_words.split()  for i in range(0, len(nswlist)):  if '.' in nswlist[i]:  if (nswlist[i].count('.') > 1) and (nswlist[i][-1] == '.'):  nswlist[i].replace('.', '', (nswlist[i].count('.') - 1))  elif (nswlist[i].count('.') == 1) and (nswlist[i][-1] == '.'):  continue  else:  nswlist[i].replace('.', '')   # Adding a period to the last word of the list if needed  if nswlist[-1][-1] != '.':  nswlist[-1] = nswlist[-1] + '.'  no\_stop\_words = ' '.join(nswlist)   # Full words reduction:  for i in stopwords:  no\_stop\_words = re.sub(r'\s\*\b' + i + r'\b\s\*', ' ', no\_stop\_words)  return no\_stop\_words |
| --- |

Json\_reader module:

| import json import pandas as pd   class Read:  def \_\_init\_\_(self, path, df):  file = open(path)  data = json.load(file)  self.id = data['paper\_id']  self.abstract = []  self.body = []  self.doi = []  self.url = []  self.author = []  self.title = []  self.journal = []  abst = []  bod = []  for i in data['abstract']:  abst.append(i['text'])  for i in data['body\_text']:  bod.append(i['text'])  self.abstract = ' '.join(abst)  self.body = ' '.join(bod)  self.doi = df['doi'].loc[df['sha'] == self.id]  self.url = df['url'].loc[df['sha'] == self.id]  self.author = df['authors'].loc[df['sha'] == self.id]  self.title = df['title'].loc[df['sha'] == self.id]  self.journal = df['journal'].loc[df['sha'] == self.id]   def \_\_repr\_\_(self):  return f'id-{self.id}: doi-{self.doi.to\_string()} url-{self.url.to\_string()} author-{self.author.to\_string()} title-{self.title.to\_string()} journal-{self.journal.to\_string()} abstract-{self.abstract[:100]}... body-{self.body[:100]}' |
| --- |

TfIdf module:

| import pandas as pd from tqdm import tqdm import math  # creating the Tf-Idf table using only a dataframe def TfIdftransform(df):  unique = uniquewords(df)  tfidf = pd.DataFrame(index=unique)  tfdf = tf(df, unique)  idfdf = idfl(df, unique)  print('tfidf:')  for i in tqdm(range(0, len(df))):  ii = str(i)  col = []  for word in range(0, len(unique)):  tfi = tfdf[ii].iloc[word]  idfi = idfdf['idf'].iloc[word]  tfidfi = tfi \* idfi  col.append(tfidfi)  tfidf[ii] = col  return tfidf   # defining a function to get the list of all the unique words in the corpus def uniquewords(df):  print('unique:')  for i in tqdm(range(0, len(df))):  body = df['processed\_body'].iloc[i]  bodyl = body.split()  unique = []  for word in bodyl:  if unique.count(word) == 0:  unique.append(word)  print(len(unique))  return unique   # defining the tf(t,d) function def tf(df, uni):  unique=uni  tfdf = pd.DataFrame(index=unique)  print('tf:')  for i in tqdm(range(0, len(df))):  body = df['processed\_body'].iloc[i]  bodyl = body.split()  column = []  for word in unique:  count = bodyl.count(word)  column.append(count / len(bodyl))  index = str(i)  tfdf[index] = column  return tfdf   # defining the idf(t) def idfl(df, uni):  unique=uni.copy()  idfdf = pd.DataFrame(index=unique)  col = []  print('idf:')  for word in tqdm(unique):  num = 0  for i in range(0, len(df)):  body = df['processed\_body'].iloc[i]  bodyl = body.split()  num = num + bodyl.count(word)  col.append(num)  idfdf['docf'] = col  idf = []  for i in idfdf['docf']:  n = len(df)  iidf = math.log10(n / (i + 1))  iidf = abs(iidf)  idf.append(iidf)  idfdf['idf'] = idf  return idfdf |
| --- |

Word2Vec model:

| import numpy as np from gensim import models from tqdm import tqdm import matplotlib.pyplot as plt from sklearn.decomposition import PCA   def createmodel(df):  print('create model:')  docs = df['processed\_body']  asentences = createsentences(docs.iloc[0])  model = models.Word2Vec(sentences=asentences, min\_count=0, vector\_size=200, workers=5, window=7, sg=1)  for i in tqdm(range(1, len(df))):  Asentences = createsentences(df['processed\_body'].iloc[i])  token\_count = len(Asentences)  model.train(corpus\_iterable=Asentences, word\_count=0, total\_examples=token\_count, epochs=10)  return model   def createsentences(doc):  sentences = []  where = [0]  docl = doc.split()   # Create a list of indexes of words with periods  for i in range(0, len(docl)):  if '.' in docl[i]:  where.append(i)  where.append(i + 1)   # Create a list of sentences according to the indexes of words with periods  for i in range(0, int(len(where) / 2) + 1, 2):  sentence = []  for j in range(where[i], where[i + 1] + 1):  sentence.append(docl[j])  sentences.append(sentence)   # Remove periods from final sentences list  for i in range(0, len(sentences)):  sent = sentences[i]  for j in range(0, len(sent)):  sent[j] = sent[j].replace('.', '', )   return sentences   def plot\_with\_matplotlib(x\_vals, y\_vals, labels):  plt.figure(figsize=(12, 12))  plt.scatter(x\_vals, y\_vals)  labels = list(range(len(labels)))  for i in labels:  plt.annotate(labels[i], (x\_vals[i], y\_vals[i]))  return plt  def reduce\_dimensions(model):  vectors = np.asarray(model.wv.vectors)  labels = np.asarray(model.wv.index\_to\_key)   pca = PCA(n\_components=2)  vectors = pca.fit\_transform(vectors)   x\_vals = [v[0] for v in vectors]  y\_vals = [v[1] for v in vectors]  return x\_vals, y\_vals, labels |
| --- |