Module 4 - Final Report

Course: Capstone - I

Course Code: AIDI1003

Course Facilitator: Dr. Uzair Ahmad

Team Members

Terence Yu (#100827101)
Jaspreet Singh Marwah (#100847939)
Sherap Gyaltsen (#100831790)
Shail Patel (#100846377)
Lawrence Wanderi Mwangi (#100836815)

Date: 10th December, 2021

Code: https://github.com/Zantorym/Aidi-capstone-I

Executive Summary

This report covers what we did for our capstone project. Our project was able to successfully match an input document, either a job description or a resume document, to similar documents of either type. We were able to improve upon the accuracy of the original RecSys system.

A similarity rating is used as a mathematical way of measuring how similar one document is to another. It is expressed as a decimal value from 0 to 1 with a value of 0 meaning totally different texts and a value of 1 representing the same text. This project uses similarity ratings to provide the top N documents that match the contents of the input document.

We have implemented TF-IDF word embedding, BoW word embedding, cosine similarity, and euclidean distance.

Introduction

Our group is looking to enhance the functionality of the current Recommendation System (RecSys), specifically by working on the pre-processing, word embedding and similarity detection portions of the algorithm. This is to improve the accuracy of the results returned by the recommendation system. An improvement of the results is determined by comparing the results delivered by the modified system and the current system against a manually prepared target results document, and comparing the accuracies of the results.

Rationale Statement

With the increasing number of jobs and job applicants being posted and submitted on a day to day basis, human resources personnel are having difficulty keeping up with figuring out which applicants are most suitable for the positions offered. AI is used as a tool to help them in filtering out those applicants who are deemed not a match to the role. This is done by finding matching resumes to the job description of the job post. By providing an algorithm that measures the similarity rating of resumes to the job description and only providing the ones that match a criteria set by the person, the number of actual resumes that they need to manually review are reduced to a manageable level.

Problem

The main challenge encountered at this stage is the conversion of text into numerical values which machine learning algorithms can use for processing. While in the previous stage of the project, the text has successfully been converted into tokens, this is only the start as these tokens are still text (alphanumeric). Natural Language Processing (NLP) requires these texts to be converted into vectors for processing. Initial attempts to manually code the algorithm to convert these texts into a bag of word vectors and further into TF-IDF vectors proved ineffective. The manually created algorithms performed sufficiently with small datasets. However, they were unable to process the large input corpus in a satisfactory time period. In fact, the long processing time caused the Google Colaboratory environment to time out during processing. Instead, we relied on scikit-learn’s implementations of BoW and TF-IDF.

Data Requirements

While the same original documents are used as the source corpus, to save on overall processing time, the pickled versions are used instead. The pickled versions are two python dictionaries where the filename is the key and the contents of the file are the corresponding value in the dictionary. This saves processing time as it is very fast to load the pickled files into memory.

Data

The original source corpus as provided at the start of the project is still used as the main input where job descriptions and resume texts are obtained. These are obtained from the recsys virtual machine stored in approximately 200 thousand files in two separate directories (one for job descriptions and another for resumes). For our testing dataset, we have used the definitive ranking originally provided to us and have manually appended rankings for an additional 81 files.

Processing Environment

Due to the large number of files to be processed, the cloud was determined to be the optimum environment to use. Processing will be done using Google Colaboratory (or Colab) for cost considerations. Google Colab provides powerful machines and storage combined with a Jupyter notebook environment which makes documentation and coding simpler and more expressive using text blocks.

Even though Google Colab provides a temporary storage to be used for input datasets, this is not sufficient for the purposes of this project due to its temporary nature and limited size. To address this, Google Drive is used to host all input, intermediary, and output files. Google Drive was chosen due to its ease of interoperability with Google Colab.

Exploratory Data Analysis

General Information

The dataset consists of 201,243 text documents. Of those, 151,211 files are job descriptions (JDs), and 50,032 files are resumes. Each file is structured as follows: the title of the file is the job that it relates to, and the content of the file is the raw text from a job description/resume.

Noise in the Dataset

There is evidence that the JDs were scraped from webpages as the files include text such as “apply now” and “Send me jobs like this” which would be buttons on a webpage. As this text is not relevant to the actual job position, it may contribute to noise in the dataset.

The dataset seems to be pre-processed to some extent, as all sorts of punctuation marks are missing and have been replaced with empty space. For instance, wherever you would expect the word “I’m”, you would instead see “I m”. As a result, when tokenizing the dataset, “I’m” would be treated as two separate words (“I” and “m”). This may negatively influence the accuracy with which we are able to retrieve relevant documents.

Some documents were empty. These were removed during the cleaning phase.

Unique Words in the Dataset

After standardizing the capitalization of all the words inside the dataset, we are left with 149,961 unique words in the resume corpus and 586,718 words in the JD corpus. After filtering out stop words, these numbers drop down to 149,843 and 586,599 for resumes and JDs, respectively.

Since the English language only has 171,476 words according to the Oxford English Dictionary, this means that our dataset contains a large amount of noise in the form of words that are not actually words. This, in-part, is due to 1-letter “words” formed due to the pre-processing mentioned in the “Noise in Dataset” section of this report. This is also due to “words” such as links to webpages or phone numbers

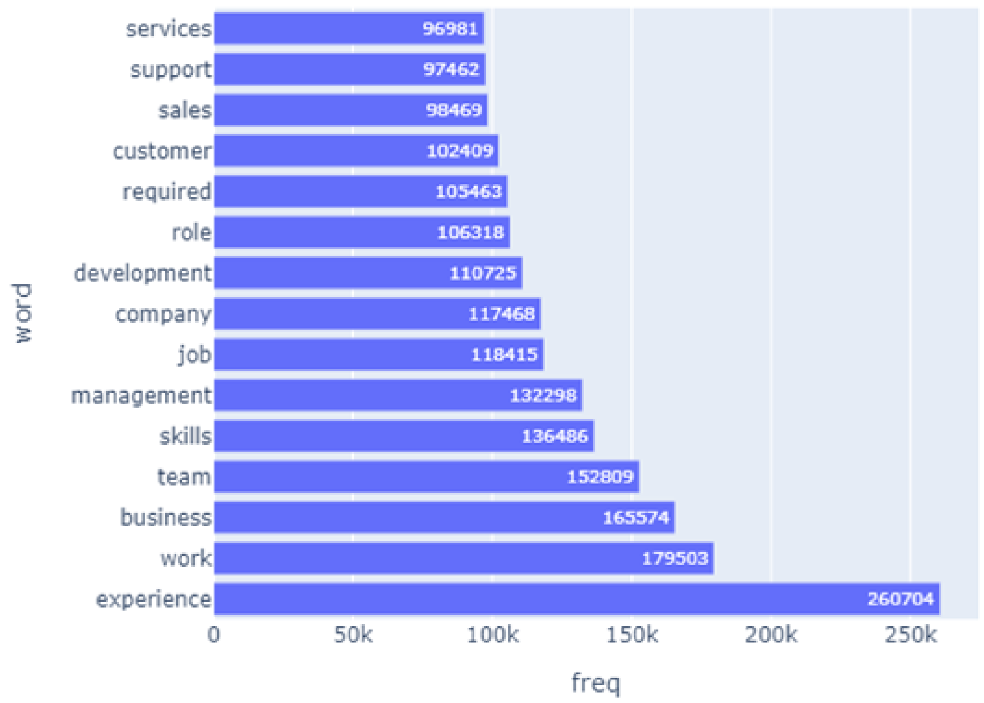
To demonstrate this point, we can find the number of unique words in the dataset that occur more than once or twice. Since “words” such as links and phone numbers are unique to a job description/resume, they would not appear across multiple documents. We can also remove all 1-letter “words” and any “word” that is not strictly alphabetic from consideration. On doing so, we get 121,953 unique words in the JD corpus and 56,320 unique words in the resume corpus, which is much more reasonable.

Overall, the dataset contains 146,350 unique words.

Most Popular Words

Job Description Corpus

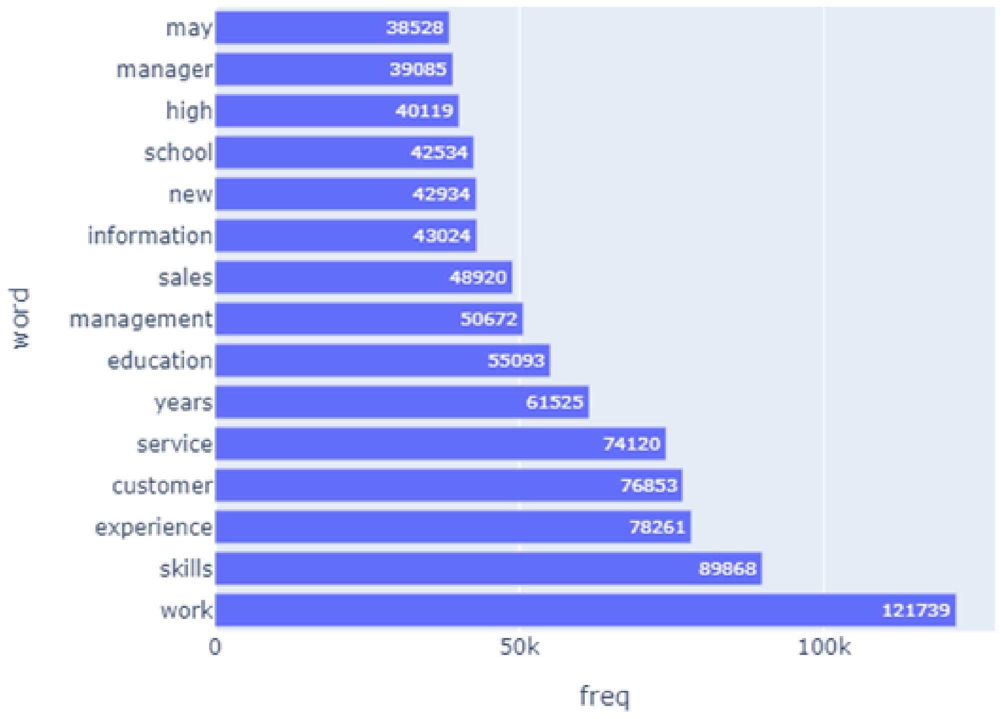
experience
business
work
skills
team
management
development
company
sales
services
customer
required
job
support
role
knowledge
including
manager
software
data
years
looking
industry
apply
please
position
service
ability
client
systems
project
working
design
requirements
new
strong
within
candidate
technical
excellent
opportunity
recruitment
must
solutions
environment
information
customers



The word that occurs the most in the JD corpus is “experience”, with a frequency of 260,704. The mean frequency of words within the JD corpus is 202.19 and the median frequency is 6.

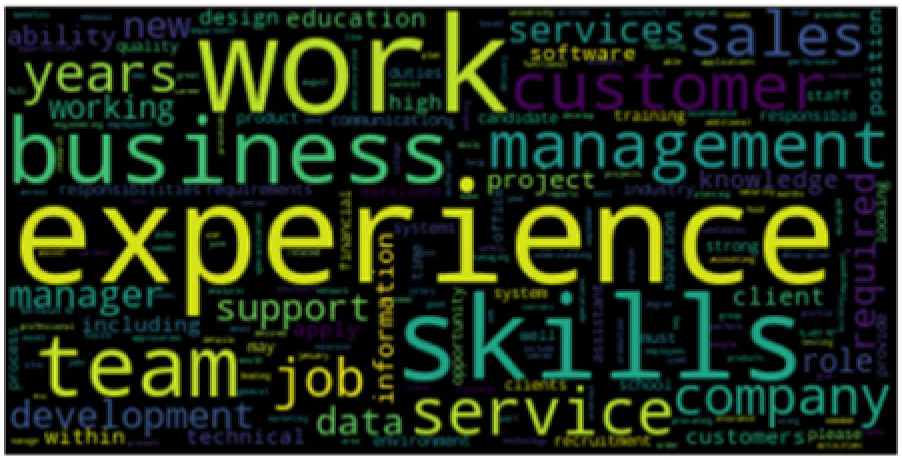
Resume Corpus

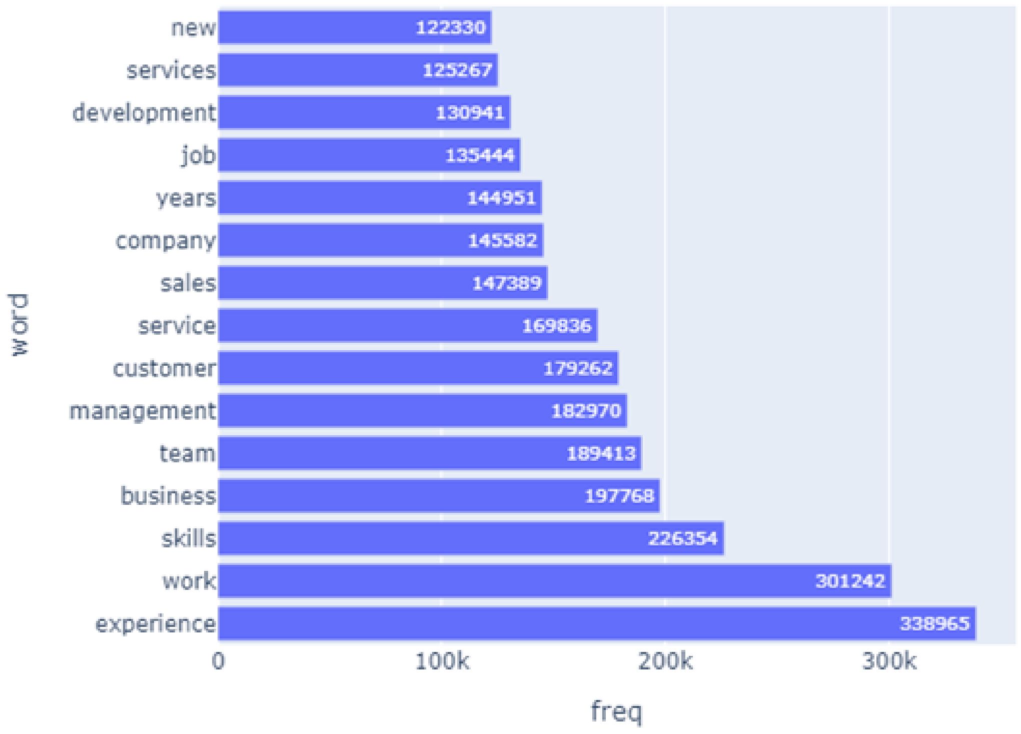
work
skills
experience
customer
service
years
education
management
high
school
office
may
sales
present
manager
customers
information
new
august
june
team



The word that occurs the most in the resume corpus is “work”, with a frequency of 121,739. The mean frequency of words within the resume corpus is 198.28 and the median frequency is 9.

Overall



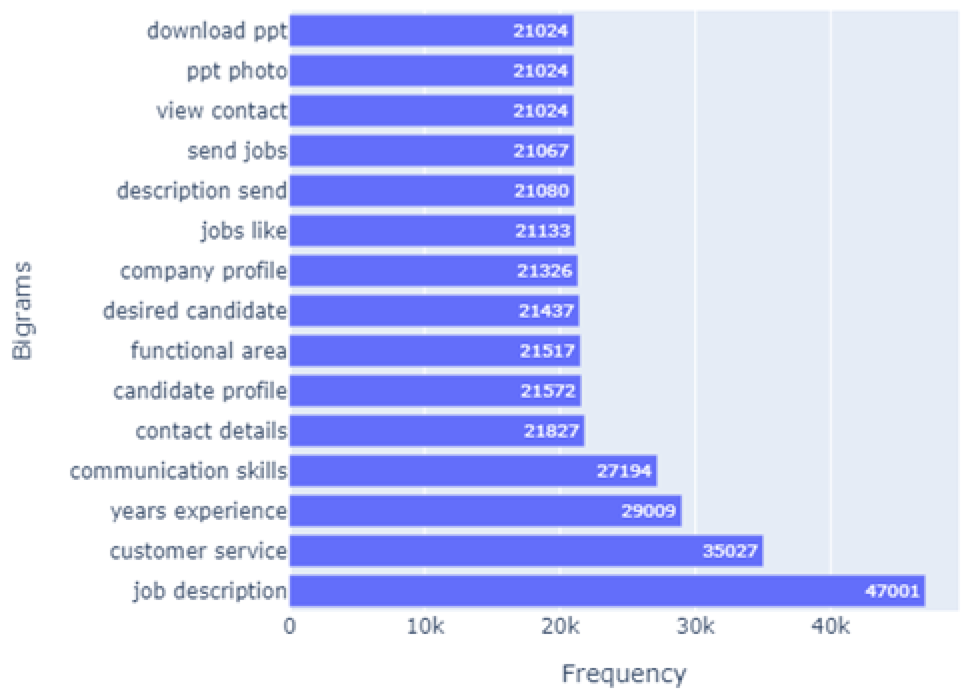


The word that occurs the most in the dataset is “experience”, with a frequency of 338,965. The mean frequency of words within the dataset is 244.79 and the median frequency is 6.

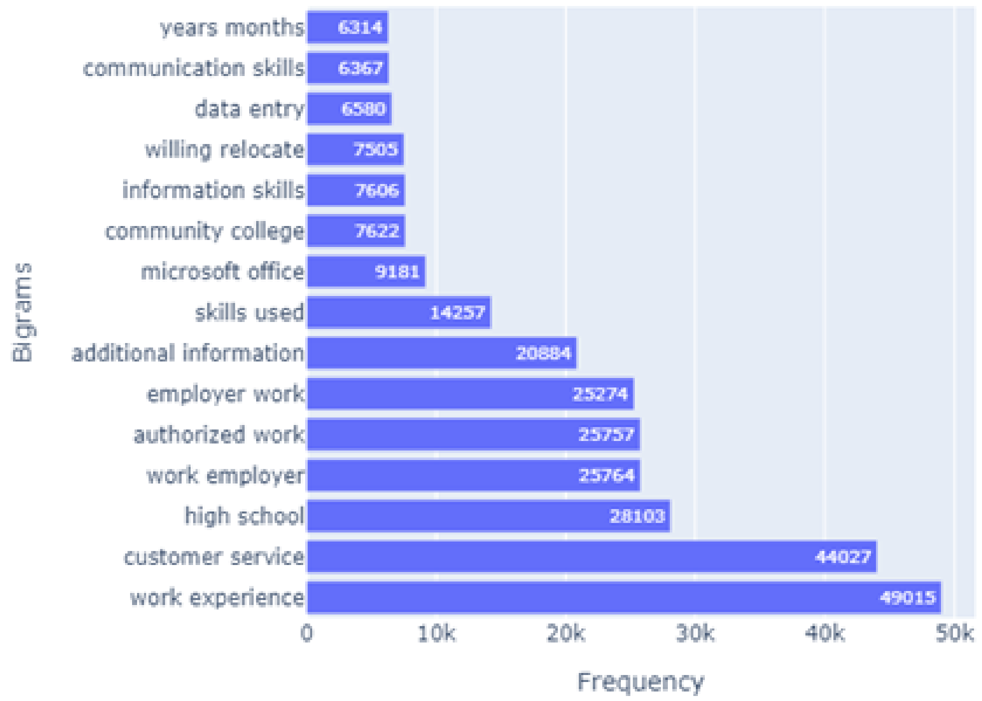
For both corpuses, “business”, “skill” and “experience” are one of the most common words.

Most Popular Bigrams

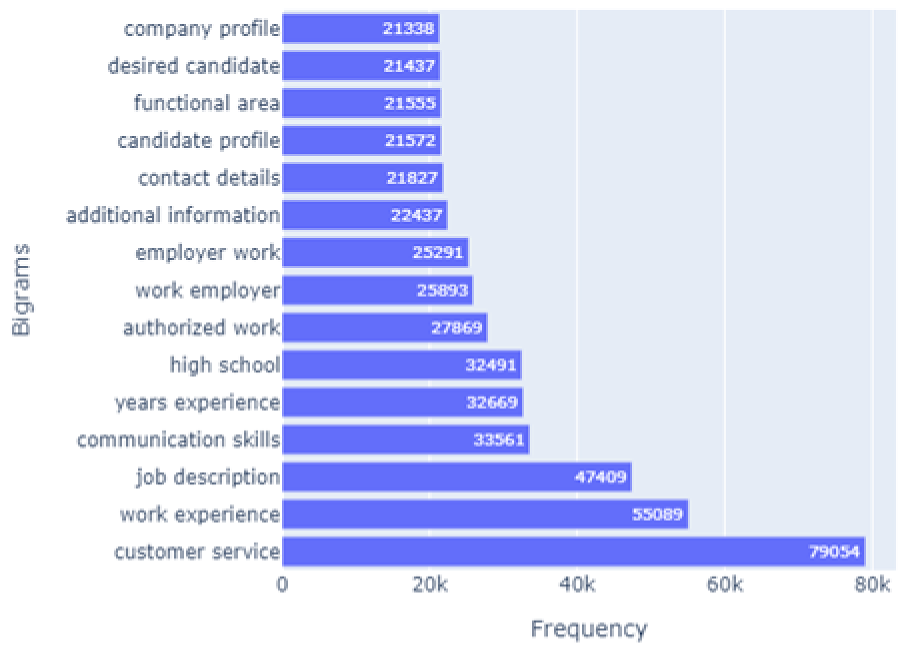
Job Description Corpus



Resume Corpus



Overall



Bigrams
company profile
21338
desired candidate
21437
functional area
21555
candidate profile
21572
contact details
21827
additional information
22437
employer work
25291
work employer
25893
authorized work
27869
high school
32491
years experience
32669
communication skills
33561
job description
47409
work experience
55089
customer service
79054
0
20k
40k
60k
80k
Frequency

Pre-Processing, Cleaning, and Preparation

Corpus

Project input is composed of thousands of files separated into two groups: Job Descriptions and Resumes. Each file represents a single document. All the files together comprise the whole corpus that will be matched for similarity against each other.

The number of files found are summarized in the table below:

Type	**Count**
Job Description	151,210
Resumes	50,023

Collection

All the input files are stored in a directory in the provided virtual machine. The files were downloaded from the virtual machine using the SFTP protocol and eventually transferred to the cloud for processing.

Challenges

As the files are broken down into hundreds of thousands of files, loading and reading them one at a time in Google Collab was extremely slow. This may be due to restrictions that Google included to prevent abuse and therefore rate limited loading of multiple files.

Files Preprocessing

To get over the challenge previously mentioned, all the files (per type) were merged into a single document which contains a colon separated value containing the filename as the first field and the file contents as the second field. Using a colon to separate the fields is possible as an initial investigation of the files showed that none contains colons.

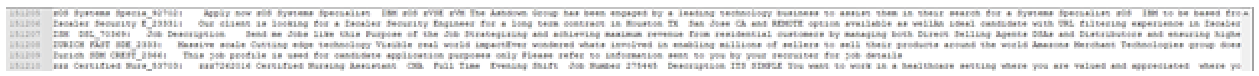
To merge the files, the UNIX system of the virtual machine containing the files were used to generate the content and concatenate all of them together. The resulting files were again downloaded using SFTP protocol and uploaded into Google Drive for later processing.

The resulting files are summarized in the table below:

Filename	Number of Lines	Size in KiloBytes

jd.list	151,210	272,486
resume.list	50,023	115,744

Example contents are shown in the image below:



```
z06 Systems Specia_92702:    Apply now z06 Systems Specialist  IBM z06 zVSE zVM The Ashdown Group has been engaged by a leading technology business to assist them in their search for a Systems Specialist z06  IBM to be based fro
Zscaler Security E_23331:   Our client is looking for a Zscaler Security Engineer for a long term contract in Houston TX  San Jose CA and REMOTE option available as wellAn ideal candidate with URL filtering experience in Zscaler
ZSE  DSL_70369:   Job Description    Send me Jobs like this Purpose of the Job Strategizing and achieving maximum revenue from residential customers by managing both Direct Selling Agents DSAs and Distributors and ensuring highe
ZURICH FAST SDE_2331:   Massive scale Cutting edge technology Visible real world impactEver wondered whats involved in enabling millions of sellers to sell their products around the world Amazons Merchant Technologies group does
Zurich SDM CREST_2346:   This job profile is used for candidate application purposes only Please refer to information sent to you by your recruiter for job details
zzz Certified Nurs_53703:    zzz7262016 Certified Nursing Assistant  CNA  Full Time  Evening Shift  Job Number 275465  Description ITS SIMPLE You want to work in a healthcare setting where you are valued and appreciated  where yo
```

Through this process, since Google Colab is only loading and reading in two very large files, the loading of the data into processing RAM took only seconds instead of the hours it originally took when loading in the thousands of files.

The data is read in from the two files and stored into a python dictionary. The dictionary keys are the filename (first field) and the dictionary data are the file contents (second field). A python dictionary is used for quick reference and retrieval of content for both processing and investigation purposes.

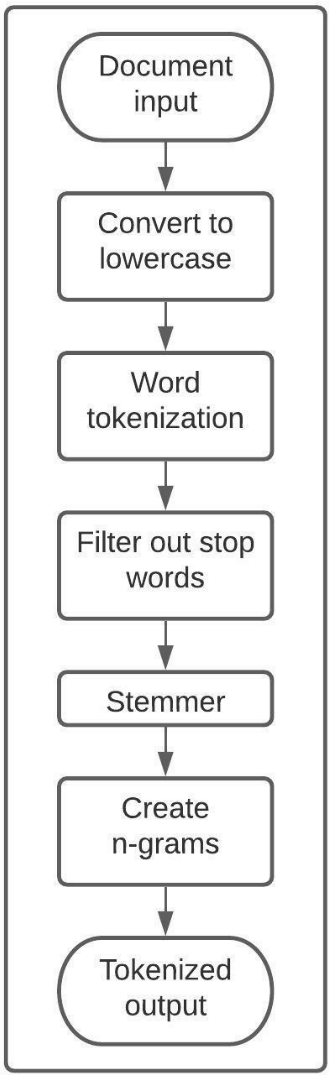
The resulting dictionaries are also saved into external files using the Python Pickle library. The resulting files are shared to Microsoft Teams, General Channel files section for sharing to other teams.

Cleaning

During processing, it was found that certain files contain only spaces and no actual text (alphanumeric characters). For those files, they are excluded from processing.

Preparation

The preparation stage of the project involves tokenizing the documents which comprise the corpus. A flexible pipeline is used to process the documents to generate the corpus. The tokenization pipeline is shown in the figure below.



To allow for flexibility in preprocessing, multiple possible options are allowed for various stages such as the word tokenization, filter stop words, stemmer, and create n-grams stages.

All the contents of the documents are converted to lowercase to avoid creating different tokens based on capitalization. Since capitalization is primarily important for differentiating between proper nouns and common nouns such as Baker (as a name) and baker (as a profession), it is less relevant in finding similar documents between job descriptions and resumes. As such it was deemed as a necessary step to reduce the number of tokens generated.

Two options are provided when choosing the word tokenization algorithm to use. One can choose to simply use Python's String.split() method to tokenize the words or to use NLTK TreebankWordTokenizer instead. The String.split() method is used as the default method for processing time considerations.

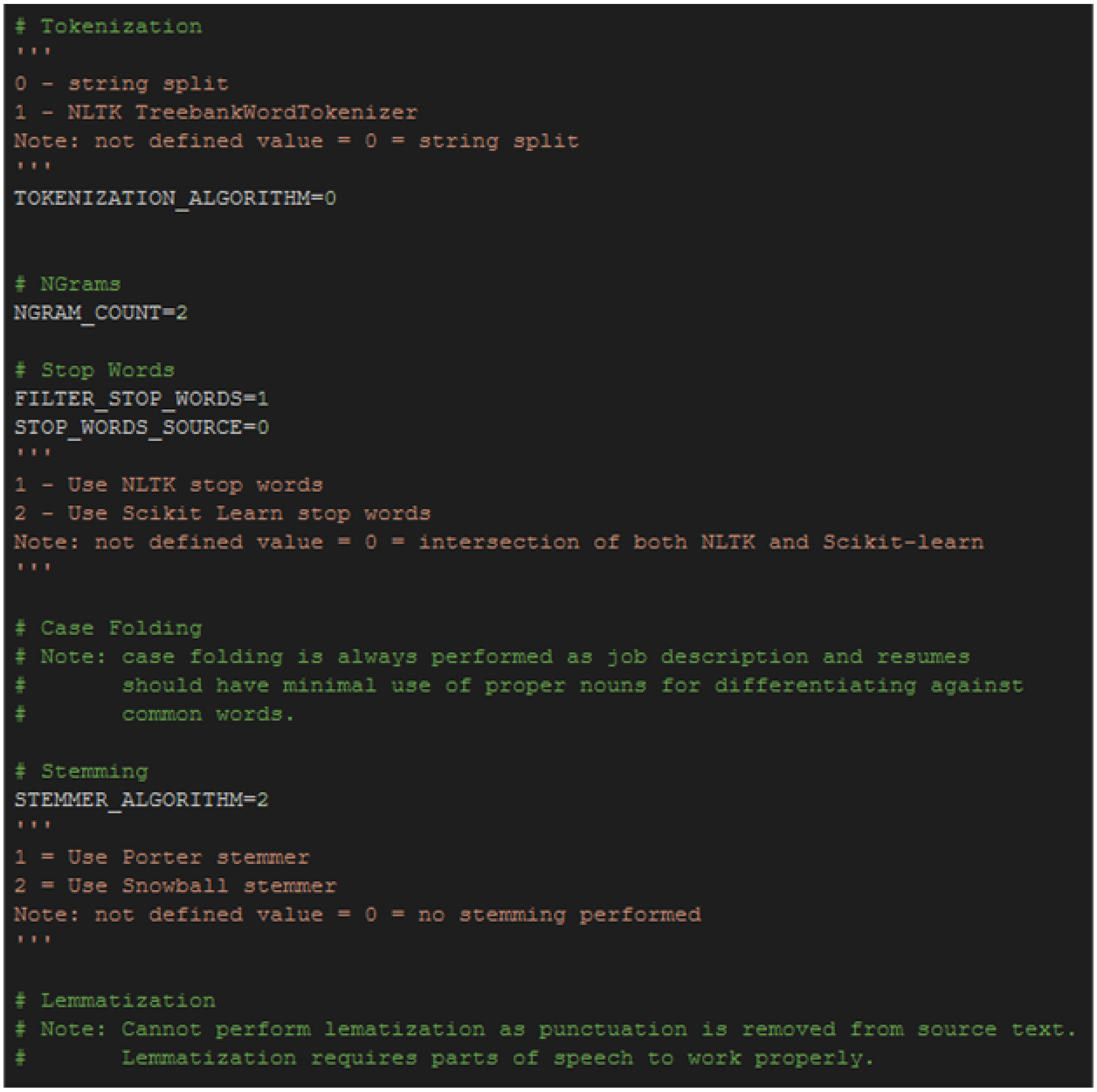
Filtering of stop words can be enabled or disabled. When enabled there are three choices for the source of the stop words to be used. One can choose to use the stop words as defined in the Scikit-Learn Library, the NLTK stop words, or the intersection of the stop words found in both libraries. By default, stop words are filtered out and the intersection of stop words found in both Scikit-Learn and NLTK are used as the list of stop words.

Three options are also provided for the stemmer stage. One can choose to use the Porter Stemmer, the Snowball stemmer, or not to stem the tokens. By default, the Snowball Stemmer is used due to its benefits in reducing the number of unique tokens produced.

The last stage involves creating n-grams if deemed necessary. In this stage the user can simply enter the number to be used as “n” in n-grams. Entering a value of 1 or 0 will provide the same result as single word tokenization. By default, 2-grams are used as two-word descriptions that are important for job descriptions or resumes, such as “artificial intelligence”, “uber driver”, “project manager”, etc.

Note that there is no lemmatization stage involved in the pipeline. This is because upon checking the contents of the files, all punctuation marks have been removed from the contents. Lemmatization can perform best when the parts of speech are provided as input. As such lemmatization takes as input sentences instead of individual words. Since there are no punctuation marks in the data, the individual sentences cannot be obtained. Passing individual words to be lemmatized will result in the same word as the output. Thus, no lemmatization is done.

All configuration for the pipeline stages is done through modification of constants defined near the top of the code. The constants and corresponding values are summarized in the image below.



```
# Tokenization
'''
0 - string split
1 - NLTK TreebankWordTokenizer
Note: not defined value = 0 = string split
'''
TOKENIZATION_ALGORITHM=0


# NGrams
NGRAM_COUNT=2

# Stop Words
FILTER_STOP_WORDS=1
STOP_WORDS_SOURCE=0
'''
1 - Use NLTK stop words
2 - Use Scikit Learn stop words
Note: not defined value = 0 = intersection of both NLTK and Scikit-learn
'''

# Case Folding
# Note: case folding is always performed as job description and resumes
#       should have minimal use of proper nouns for differentiating against
#       common words.

# Stemming
STEMMER_ALGORITHM=2
'''
1 = Use Porter stemmer
2 = Use Snowball stemmer
Note: not defined value = 0 = no stemming performed
'''

# Lemmatization
# Note: Cannot perform lematization as punctuation is removed from source text.
#       Lemmatization requires parts of speech to work properly.
```

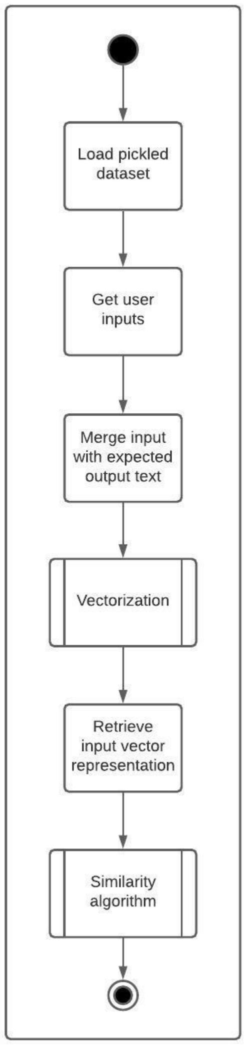
Processing

Model / Architecture Approach

In order to resolve the problem of slow processing time for custom-made vectorisation algorithms (owing to the large size of the corpus), the project team decided to utilize an established and performant TF-IDF vectorization algorithm - Scikit Learn's TfidfVectorizer, as well as a Bag-of-Words vectorization algorithm - Scikit Learn's CountVectorizer. During implementation, they were found to be much faster in processing the corpus and converting it into vectors. Furthermore, these vectorizers can take in parameters which allow us to customize the vectorization process according to our requirements.

Data Processing Pipeline

There are some assumptions and observations that the team has made regarding the preprocessing of the text file to optimize vectorization. This has led to the processing pipeline performed and shown in the flowchart below.


Load pickled dataset
Get user inputs
Merge input with expected output text
Vectorization
Retrieve input vector representation
Similarity algorithm

The code initially retrieves the corpus from the pickled file. Once loaded, the result are two Python dictionaries containing the text for each type of file, job description or resume respectively.

Then user input is required to specify three inputs: the type of file used for the input, the filename of the file which will be the input document, and the type of the files to be compared for similarity against the input document. The types of the files for both input and output are used to determine from which dictionary to retrieve the necessary information. Since the keys of the dictionaries are the actual filenames, it is easy to retrieve the text of the file for processing. Another benefit is that it is easily verified if the filename provided as input exists in the corpus.

The next step is to combine the text of the input into the dataset comprising the text of the selected output format. This is required because for proper vectorization to be performed both the input and the output texts must be vectorized with the same columns. Since the Inverse Document Frequency portion of the TF-IDF algorithm normalizes the vectors against all the words in the full dataset processed, both input and output must be processed together.

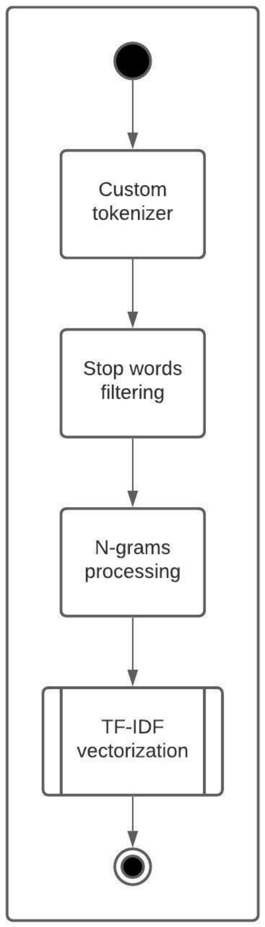
Once merged, vectorization of the combined dataset is then done. Details on the vectorization process are explained in the corresponding section of this document.

After vectorization, the next step is to retrieve the vector representation of the input text document. This is important as it will be used to find the similarity against other documents. With this vector value, a similarity measure algorithm is performed to obtain a similarity value.

The final step is just to arrange the similarity values of the output and retrieve the top N texts which provided the highest values.

Vectorization Process

As previously mentioned, Scikit-Learn's TfidfVectorizer and CountVectorizer are used; but are modified through passed-in arguments. A flow chart of the vectorization process is described in the flowchart below and explained afterwards.



The default tokenization method of TfidfVectorizer/CountVectorizer has been customized to return individual words as tokens using string's split method. Then filtering is performed to ensure that only alphanumeric tokens are returned.

For stop words filtering, the stop words used is the intersection of the stop words defined in both NLTK and Scikit Learn's stop words modules. Only words that occur in both modules are in the final list of stop words to be filtered out. If a stop word is found in only one module, it is dropped from the list.

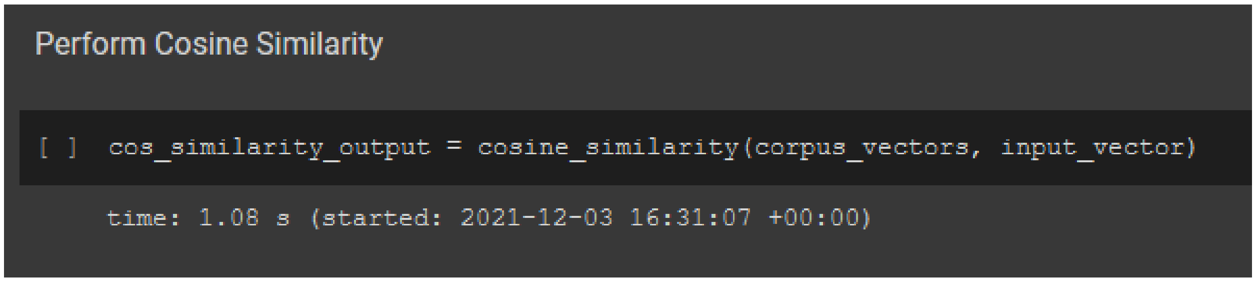
In the context of job descriptions and resumes, there are many phrases used to describe work, technology, or process that are composed of multiple words. For example the term "machine learning" is composed of two words and taking each individual word alone without the other changes the context. It is for this reason that 1-gram and 2-grams are used for the vectorization. Although there are potential phrases which are composed of 3 or more words, this is left out as the additional processing time is deemed to be not worth the smaller chance of these phrases occurring.

Once these steps are performed, the actual vectorization is performed by the library. The resulting output is a numpy sparse matrix comprising the vectors for the dataset processed.

Algorithm Evaluation

Cosine Similarity

To obtain the cosine similarity values for the output, Scikit-Learn's cosine_similarity function was used. The complete vectorized representation of the complete processed dataset is passed in along with the input text vector representation.

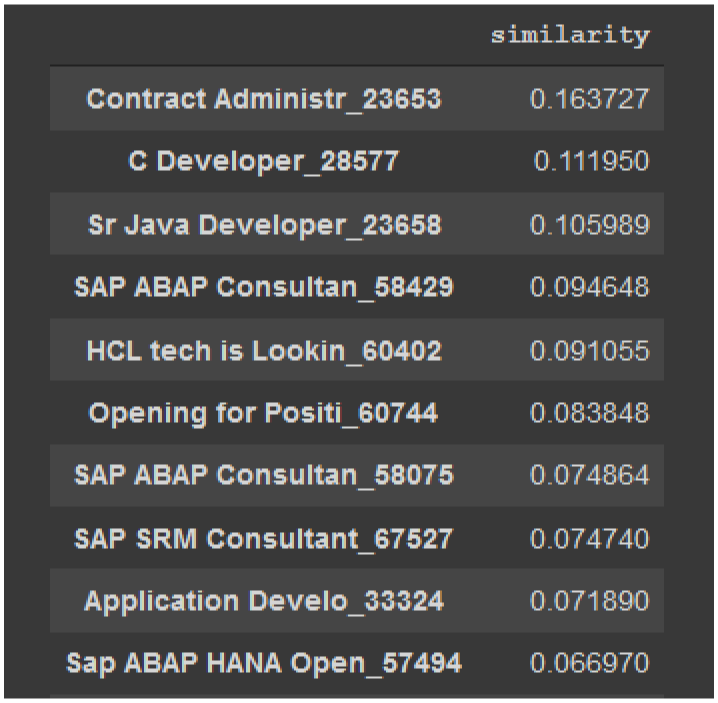


```
cos_similarity_output = cosine_similarity(corpus_vectors, input_vector)
```

```
time: 1.08 s (started: 2021-12-03 16:31:07 +00:00)
```

Since the input text vector is also included in the vectorized dataset passed to cosine similarity, when calculating the similarity values, it will return a result of 1. This is because the cosine similarity of an item with itself is 1. Therefore before arranging the results for decreasing similarity result values, the cosine similarity of the input with itself is first removed.

The result of this process returns the filenames of the output text type documents that are most similar to the input as well as their cosine similarity value. An example is provided in the screenshot below.



	similarity
Contract Administr_23653	0.163727
C Developer_28577	0.111950
Sr Java Developer_23658	0.105989
SAP ABAP Consultan_58429	0.094648
HCL tech is Lookin_60402	0.091055
Opening for Positi_60744	0.083848
SAP ABAP Consultan_58075	0.074864
SAP SRM Consultant_67527	0.074740
Application Develo_33324	0.071890
Sap ABAP HANA Open_57494	0.066970

Euclidean Distance

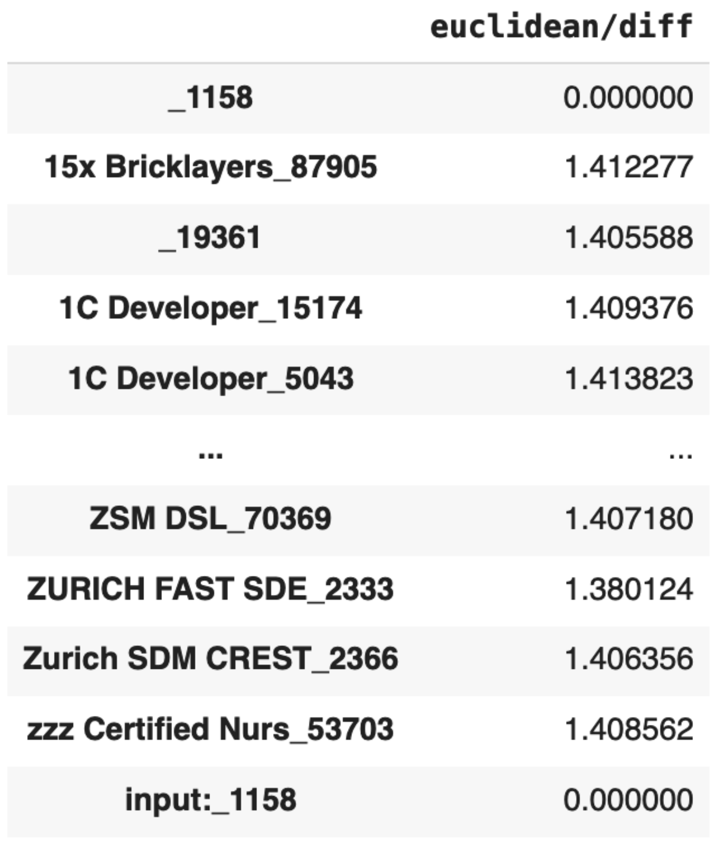
Euclidean distance method is for measuring the proximity between vectors in a vector space. Sklearn.metrics.pairwise has been used to import euclidean_distance.

```
from sklearn.metrics.pairwise import euclidean_distances
```

The entire corpus vector representation is being passed along with the input text vector.

```
jd_test_output = euclidean_distances(corpus_vectors,input_vector)
```

Resultant output will be a dataframe consisting of filenames as index and the euclidean distance between the input text vector and the entire vectorised dataset. In the example below the input filename is “_1158” which is present in the dataset. When estimating the euclidean distance between two vector points turns out to be 0 since it is taking the difference of an item with itself.



	euclidean/diff
_1158	0.000000
15x Bricklayers_87905	1.412277
_19361	1.405588
1C Developer_15174	1.409376
1C Developer_5043	1.413823
...	...
ZSM DSL_70369	1.407180
ZURICH FAST SDE_2333	1.380124
Zurich SDM CREST_2366	1.406356
zzz Certified Nurs_53703	1.408562
input:_1158	0.000000

Candidate Algorithm Selection and Rational

Word Embedding

In natural language processing (NLP), word embedding is a term used for the representation of words for text analysis, typically in the form of a real-valued vector that encodes the meaning of the word such that the words that are closer in the vector space are expected to be similar in meaning. Word embeddings can be obtained using a set of language modeling and feature learning techniques where words or phrases from the vocabulary are mapped to vectors of real numbers. Conceptually it involves the mathematical embedding from space with many dimensions per word to a continuous vector space with a much lower dimension.

TF-IDF

A popular word embedding technique for extracting features from corpus or vocabulary. This is a statistical method to find how important a word is to a document all over other documents.

TF (Term Frequency)

In TF, we are giving some scoring for each word or token based on the frequency of that word. The frequency of a word is dependent on the length of the document. In a large document a word occurs more than a small or medium sized document. So, to overcome this problem we will divide the frequency of words with the length of the document (total number of words) to normalize. By using this technique also, we are creating a sparse matrix with frequency of every word.

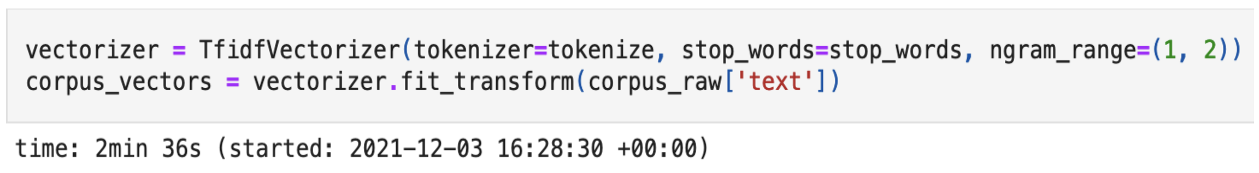
TF = no. of times term occurrences in a document / total number of words in a document.

IDF (Inverse Document Frequency)

Here we are assigning a score value to a word. This score value indicates how rare a word is across all documents. Rarer words have more IDF scores.

IDF = $\log_e$ (total number of documents / number of documents which are having term)

TF IDF = TF * IDF



```
vectorizer = TfidfVectorizer(tokenizer=tokenize, stop_words=stop_words, ngram_range=(1, 2))
corpus_vectors = vectorizer.fit_transform(corpus_raw['text'])
```

```
time: 2min 36s (started: 2021-12-03 16:28:30 +00:00)
```

Bag-of-Words

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity.

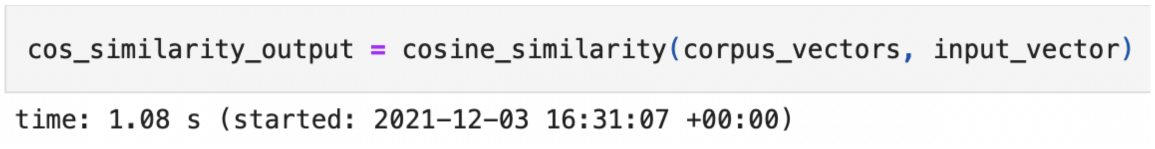
The bag-of-words model is commonly used in methods of document classification where the (frequency of) occurrence of each word is used as a feature for training a classifier. This is the case with Scikit Learn's CountVectorizer, which converts each document into a vector indicating the frequency of each word within the corpus.

Similarity

Cosine Similarity

Cosine similarity measures the similarity between two vectors. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction. It is often used to measure document similarity in text analysis. A document can be represented by thousands of attributes, each recording the frequency of a particular word (such as a keyword) or phrase in the document. Thus, each document is an object represented by what is called a *term-frequency vector.*

Term-frequency vectors are typically very long and **sparse** (i.e., they have many 0 values). Applications using such structures include information retrieval, text document clustering , biological taxonomy, and gene feature mapping. The traditional distance measures do not work well for such sparse numeric data. For example, two term-frequency vectors may have many 0 values in common, meaning that the corresponding documents do not share many words, but this does not make them similar. We need a measure that will focus on the words that the two documents *do* have in common, and the occurrence frequency of such words. In other words, we need a measure for numeric data that ignores zero-matches.



```
cos_similarity_output = cosine_similarity(corpus_vectors, input_vector)
```

```
time: 1.08 s (started: 2021-12-03 16:31:07 +00:00)
```

Cosine similarity is a measure of similarity that can be used to compare documents or, say, give a ranking of documents with respect to a given vector of query words. Let ***x*** and ***y*** be two vectors for comparison. Using the cosine measure as a similarity function, we have

$\cos(\theta) = x \cdot y/(||x|| \cdot ||y||)$,

where ||***x***|| is the Euclidean Norm of vector x=(x1,x2,…,xp), defined as x12+x22+⋯+xp2. Conceptually, it is the length of the vector. Similarly, ||***y***|| is the Euclidean norm of vector ***y***. The measure computes the cosine of the angle between vectors ***x*** and ***y***. A cosine value of 0 means that the two vectors are at 90 degrees to each other (orthogonal) and have no match. The closer the cosine value to 1, the smaller the angle and the greater the match between vectors.

Euclidean Distance

Euclidean distance calculates the distance between two real-valued vectors.

You are most likely to use Euclidean distance when calculating the distance between two rows of data that have numerical values, such as a floating point or integer values.

If columns have values with differing scales, it is common to normalize or standardize the numerical values across all columns prior to calculating the Euclidean distance. Otherwise, columns that have large values will dominate the distance measure.

Euclidean distance is calculated as the square root of the sum of the squared differences between the two vectors.

EuclideanDistance = sqrt(sum for i to N (v1[i] – v2[i])^2)

If the distance calculation is to be performed thousands or millions of times, it is common to remove the square root operation in an effort to speed up the calculation. The resulting scores will have the same relative proportions after this modification and can still be used effectively within a machine learning algorithm for finding the most similar examples.

EuclideanDistance = sum for i to N (v1[i] – v2[i])^2

This calculation is related to the L2 vector norm and is equivalent to the sum squared error and the root sum squared error if the square root is added.

Modelling Results

We tested the data against the ranking available in our test set. There are 4 possible kinds of predictions:

1. JD to JD: We are given a job description as input and provide a list of similar job descriptions
2. Resume to Resume: We are given a resume as input and provide a list of similar resumes
3. Resume to JD: We are given a resume as input and provide a list of similar job descriptions
4. JD to Resume: We are given a job description as input and provide a list of similar resumes

We used MAP (Mean Average Precision) score as the basis of our evaluation for the performance of the models.

MAP Score				
Method\Corpus	JD to JD	Resume to Resume	JD to Resume	Resume to JD
No pre-processing, TF-IDF word embedding, cosine similarity	4.82342%	2.18031%	3.84615%	2.56410%
Pre-processing, TF-IDF word embedding, cosine similarity	5.05767%	2.00374%	0%	0.32051%
No pre-processing, TF-IDF word embedding, euclidean distance	4.36046%	1.59176%	0%	0.32051%
Pre-processing, TF-IDF word embedding, euclidean distance	4.00793%	1.81647%	0%	0.32051%
No pre-processing,	4.15213%	2.80898%	3.84615%	0%

BoW word embedding, cosine similarity				
Pre-processing, BoW word embedding, cosine similarity	5.44465%	2.17228%	0%	0%
No pre-processing, BoW word embedding, euclidean distance	4.37661%	1.19382%	0%	1.28205%
Pre-processing, BoW word embedding, euclidean distance	3.63049%	2.85580%	0%	1.92307%

From these results, it is clear that there is no “one size fits all” solution to our problem. Depending on what our input file is and what type of files are being requested, the optimal approach varies.

When recommending JDs given a JD input file, the best approach is to apply pre-processing, use bag-of-words word embeddings, and implement cosine similarity as our evaluation metric.

However, when recommending resumes given a resume input file, the best approach is to apply pre-processing and use bag-of-words, but implement euclidean distance instead as our evaluation metric.

When recommending resumes given a JD file, most approaches fail to get any accuracy. However, two approaches were fruitful. In the first approach, we applied no pre-processing, used TF-IDF word embedding, and cosine similarity as our evaluation metric. In the second approach, we did the same thing but used bag-of-words word embedding instead. Both approaches performed equally well.

When recommending JDs given a resume file, by far the best approach was to apply no pre-processing, use TF-IDF word embedding, and implement cosine similarity as the evaluation metric.

Therefore, when our input file and recommended files are of the same type, bag-of-words is the better word embedding choice. But when the input and recommended files are of different types, TF-IDF works better. Also, pre-processing improves the accuracy when the input and recommended files are of the same type, but not if they’re of different types. Finally, cosine similarity is generally the better metric, except for when we need to recommend resumes from a resume input file.

Here are the best scores based on the file types:

MAP score			
Corpus\Model	Baseline Model	Best Model	Best Model Features
JD to JD	2.683528%	5.44465%	Pre-processing, BoW word embedding, cosine similarity
Resume to Resume	3.20339%	2.85580%	Pre-processing, BoW word embedding, euclidean distance
JD to Resume	0%	3.84615%	No pre-processing, TF-IDF word embedding, cosine similarity
Resume to JD	0.324675%	2.56410%	

As we can see, our current models outperform the baseline model in every scenario besides Resume to Resume.

Threats to validity

Bag-of-Words:

- It does not capture position in text, semantics, co-occurrences in different documents, etc.
- It is only useful as a lexical level feature.

TF-IDF :

- Cannot capture semantics(e.g as compared to topic models, word embeddings). With that being said, the highest TF-IDF words of a document may not make sense with the topic of the document.
- TF-IDF computes document similarity directly in the word-count space, which may be slow for large vocabularies.

Cosine similarity :

- It works best when there are a great many(likely sparsely populated) features to choose from which makes it less optimal under spaces of lower dimension.
- the magnitude of vectors is not taken into account which means that the differences in values are not fully taken into account.

Euclidean distance :

- Euclidean distance is fine for lower dimensions, however does not perform well in sparse matrices.
- If two data vectors have no attribute values in common, they may have a smaller distance than the other pair of data vectors containing the same attribute values.