UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Data Analytics Project

Students Names:
Katopodis Odysseas - 7115112400019,
Salmas Konstantinos - 7115112500014

Course: *Data Analytics [M161]*
Due: *January, 2026*

Contents

1 Part 1:Text Classification

Please note that for this segment, we submit Jupyter Notebooks and not simple python scripts. Because of the extensive experimentation we thought it is more readable and applicable that notebooks are submitted so that one can see our approach (and each run result per case). Parts two and three are simpler (in terms of code) and hence, simple python scripts are given. Source code is supplied in the equivalent folders (PartX) with appropriate names. Finally note that both scripts and notebooks require some modules to be installed so that they run without a problem (e.g., pandas, numpy, etc).

1.1 How to run

In order for one to recreate the experiments and run our code, they can either upload the notebook files to a well-known platform (e.g., Google Colab or Kaggle), or locally run them by utilizing Jupyter. All the experiments require the data be pre-processed and cleaned (as described in the next sections). As a result, firstly you should run the code that lies inside the "Data preparation and EDA".

1.2 Get to know the Data: WordCloud

Let us understand the data a bit more by analyzing what we are currently working with. For this purpose we construct wordclouds for each article category inside the dataset.

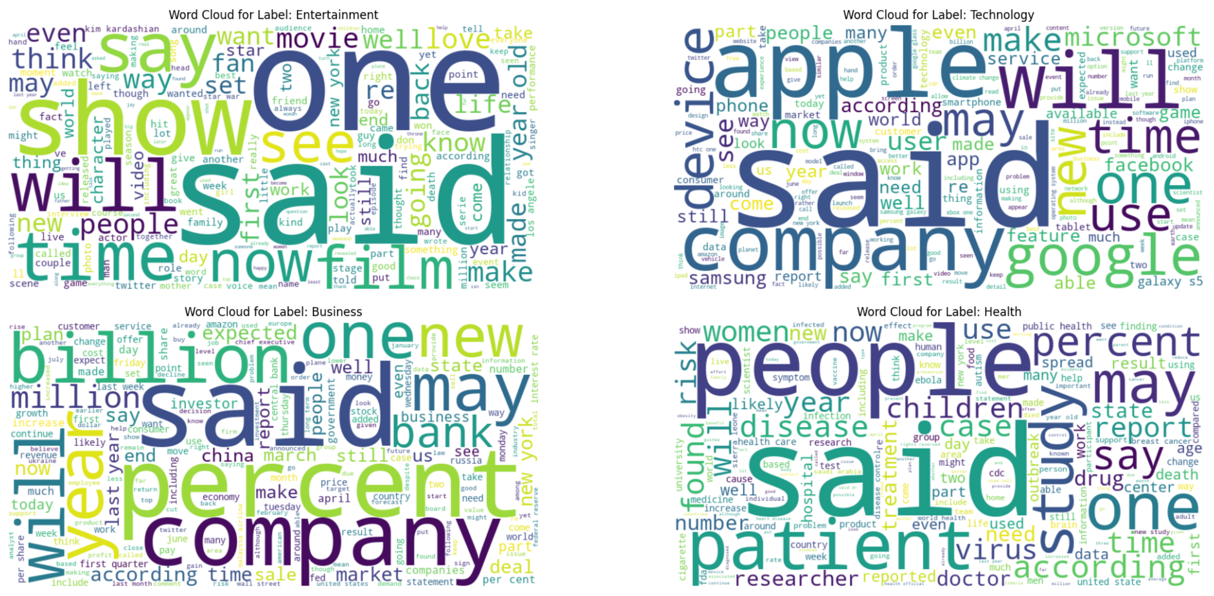

Figure 1: Word Clouds for each Article Category

1.2.1. Entertainment. In this category we can see that words like "show" and "film" dominate the wording. This is not a surprise, the film industry and more specifically movies and tv-series are a major business segment that generally govern the modern world. We can also see terms such as "year" and "will" which refer to articles that promote upcoming films and shows as many of the posts are published before the opening of a show so as to aid the market branding regarding the shows. Even though a lot of people are what we call *famous*, we can see that the only surname coming up is the "Kardashian"; clearly identifying the hegemonical influence of this particular family in the Entertainment Industry. Additionally we can see some Named Entities of specific locations such as "New York" and "Los Angeles". This is a classic hallmark of the geographic centralization regarding the film industry with United States being a global epicenter for the production of shows.

1.2.2. Technology. For this section we can see that the word cloud is dominated by major corporations names such as "Apple", "Google", "Samsung", "Microsoft", etc. We also see terms like "user", "device", "data" and "feature" which clearly refer to more technical articles about various applications. We can also see some specific product names such as "Galaxy S5" or "Google Glass" which show that there was (even at a specific time span) extensive coverage and publicity for those branded products.

1.2.3. Business. In this category, we can see that "percent" dominates the writing terminology which is no surprise as many articles refer to the Stock Exchange market or generally discuss about the financial aspects of various companies. We can also see important geo-locations coming up in the cloud such as "China" and "New York" which generally serve as primary sites for global capital aggregation housing the world's most liquid and robust financial markets. Finally we can see more specific terminology regarding the finance function such as "bank", "federal reserve", "interest rate" and so on signifying that most articles discuss the monetary and commercial opportunities of various business. In a more philosophical scale, one could say that these articles do not focus on the robustness or novelty of businesses but merely attend to their fiscal and investment nature.

1.2.4. Health. In this category we can understand that predominantly, most articles refer to academic and medical research on the field; this can be signified by terminology such as "researcher", "found", "scientist", "result" and so on. Interestingly, we can almost see no specific reference to well known diseases (with the exception of "ebola" or "cancer") which might mean that in the articles the diseases are not repeatedly used; the main corpus of an article is the symptoms, the mitigation strategies and maybe the research assumptions and results.

1.2.5. Journalistic Writing Style. What was conveniently left out of discussion was the appearance of specific words that are generally used by journalists, reporters and article writers. These words include terms such as "said" (which can be viewed as extensively used in all categories), "say", "think", "suggest" and so on. Generally, articles heavily rely on reported and indirect speech as such posts frequently quote sources or report opinions and reactions. However these words bear almost no semantic significance at all. That is to say, when trying to classify articles in categories or generally attempt natural language processing and understanding, such words are typically considered "stop-words". They can be removed without influencing the sentiment polarity of a textual segment or without affecting the primary category they might fall into.

1.3 Text pre processing

We start by checking both the types of features and the null values, in any tuple of our train data. The data set contains no null values and the types of its feature are correct.

Index reset. Data shape: (111795, 4)

#	Column	Non-Null Count	Dtype
0	Id	111795 non-null	int64
1	Title	111795 non-null	object
2	Content	111795 non-null	object
3	Label	111795 non-null	object

Table 1: Type and null values in each feature in the dataset

Then we search for duplicates based on the Title, the Content, and the combination of the two.

Duplicates	
Number of duplicate rows	0
Number of duplicate rows based on Title	1161
Number of duplicate rows based on Content	1811
Number of duplicate rows based on Title and Content	575

Table 2: Duplicates based on different features

We proceed cautiously and decide to eliminate duplicates based on both the title and the content feature (meaning that any tuple with the same title and content will be eliminated and only one instance will be kept). After removing 575 duplicates, 111220 instances remain. Index reset. Data shape: (111220, 4)

We create some data statistics based on the content feature (which will be practically the main text, the main contributor in our classifier later) like text length, word count, sentence count and average word length you get a better grasp of the data which will help in the proceeding steps.

	text_length	word_count	sentence_count	avg_word_length
count	111220	111220	111220	111220
mean	2540.216274	419.385021	24.36647	5.063333
std	2100.902069	348.882298	22.82903	0.44743
min	16	3	1	2.5
25%	1297	216	12	4.831776
50%	2022	335	19	5.051411
75%	3135	514	30	5.266425
max	78614	12562	1343	95.820513

Table 3: Content feature text statistics

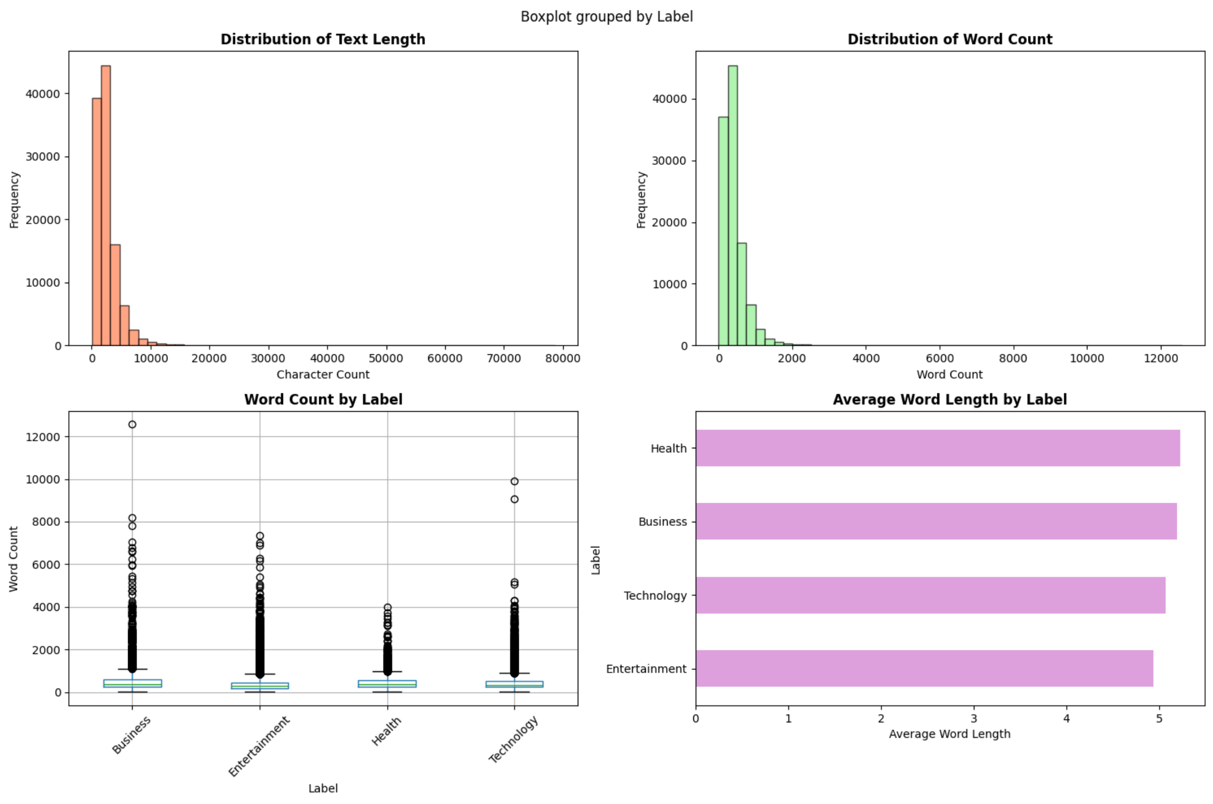

Figure 2: Content feature text statistics

Special mention should be given to the distributions of our data regarding their class.

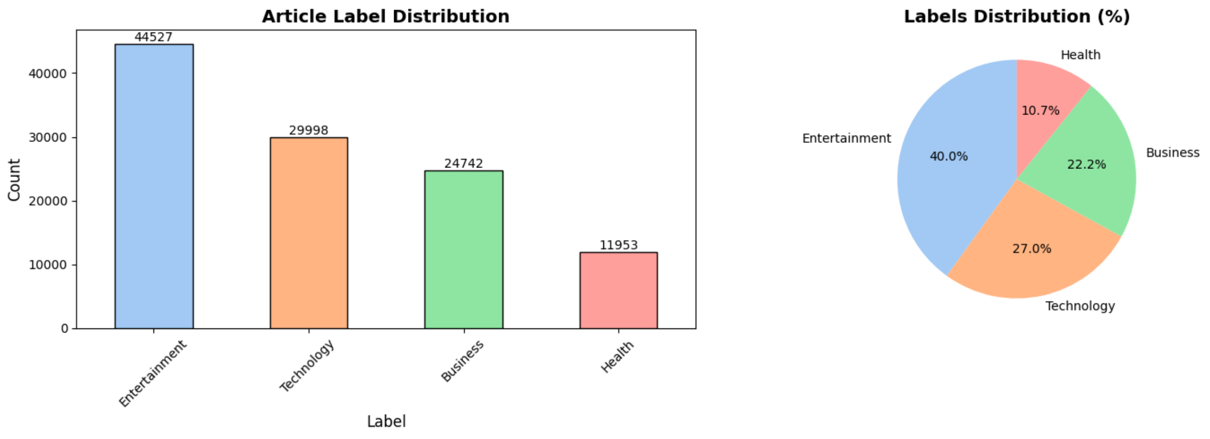

Figure 3: Distribution based on Label

As we see, there are big differences in instances per label with entertainment accounting for 40% of our data and health only accounting for 10.7%. Despite the class imbalance, the large amount of the overall data set (even for the label health we have 11,953 data instances) allows us to train a good classifier. In the event of keeping a smaller amount of data we should take care to stratify and choose pieces that have an equal distribution among the labels.

We proceed by removing words not found in the English Dictionary (nltk corpus was used[2])

Listing 1: Spellcheking words

```python
import re
import nltk
from nltk.corpus import words

# Download the words corpus if not already present
nltk.download('words')
english_words = set(words.words())

def clean_text(text):
    # Split text into words
    word_list = re.findall(r'\b\w+\b', str(text))
    cleaned_words = []
    for word in word_list:
        # Drop any word not in dictionary
        if word.lower() not in english_words:
            continue
        # Drop words with 2+ repeating chars not in dictionary (redundant now,
            but kept for clarity)
        if re.search(r'(.)\1{1,}', word):
            if word.lower() not in english_words:
                continue
        cleaned_words.append(word)
    return ' '.join(cleaned_words)

# Apply to both columns
dataTrain['Title'] = dataTrain['Title'].apply(clean_text)
dataTrain['Content'] = dataTrain['Content'].apply(clean_text)
```

This greatly helps us to later eliminate misspellings and many chat-like text and repeating character words (e.g aaaa, ahahahha), **which appear extensively in our data**.The first 20 examples in alphabetical order of word tokenization without spellchecking with dictionary:

’aa’ ’aaa’ ’aaaa’ ’aaaaaaaall’ ’aaaaaaand’ ’aaaaaahahaahahahaahahahaaha’ ’aaaaaahahahahahahaah’ ’aaaaaand’ ’aaaaah’ ’aaaaalmost’ ’aaaaand’ ’aaaai”aaaaiorg’ ’aaaall’ ’aaaalll’ ’aaaand’ ’aaaapprov’ ’aaaar’ ’aaaarizonaat”aaacom’

Now starts the main cleaning of the text in the title and content columns. There were 7 main steps as seen below which were executed in this specific order in order to achieve a consistent cleanup in preparation for inputting the text into classifier.

- expanding contractions
- Converting everything to lowercase removing special characters
- removing extra spaces
- Tokenization
- removing stop words
- Lemmatizing
- Stemming (the lemmatized words)

Listing 2: Text preprocessing

```python
import re
import nltk
import contractions
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer, PorterStemmer

# Download required NLTK data if not already present
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('omw-1.4')

stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()
stemmer = PorterStemmer()

def clean_text(text):
    # Expand contractions
    text = contractions.fix(text)
    # Convert to lowercase
    text = text.lower()
    # Remove special characters (keep only letters and spaces)
    text = re.sub(r'[^a-z\s]', '', text)
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()
    # Tokenize
    words = text.split()
    # Remove stopwords, lemmatize, and stem
    words = [stemmer.stem(lemmatizer.lemmatize(word)) for word in words if
        word not in stop_words]
    text = ' '.join(words)
    return text

for col in ['Title', 'Content']:
    dataTrain[col] = dataTrain[col].astype(str).apply(clean_text)
```

Those preprocessing steps are very crucial and greatly improve both the training speeds and the accuracy of any classifier. A quick definition for stemming and lemmatization is given on the basis that those two methods are not commonly known to computer science community with the exceptions of gorups who deal with natural language processing. Stemming is a Natural Language Processing (NLP) technique that reduces words to their base or root form (e.g., "running", "runner", and "ran" become "run") by chopping off affixes. Lemmatization is a Natural Language Processing (NLP) technique that reduces inflected or varied forms of a word (e.g., "running," "ran") to their base, dictionary form (e.g., "run"), known as a lemma. Unlike stemming, it uses vocabulary, morphological analysis, and part-of-speech context to ensure accuracy.

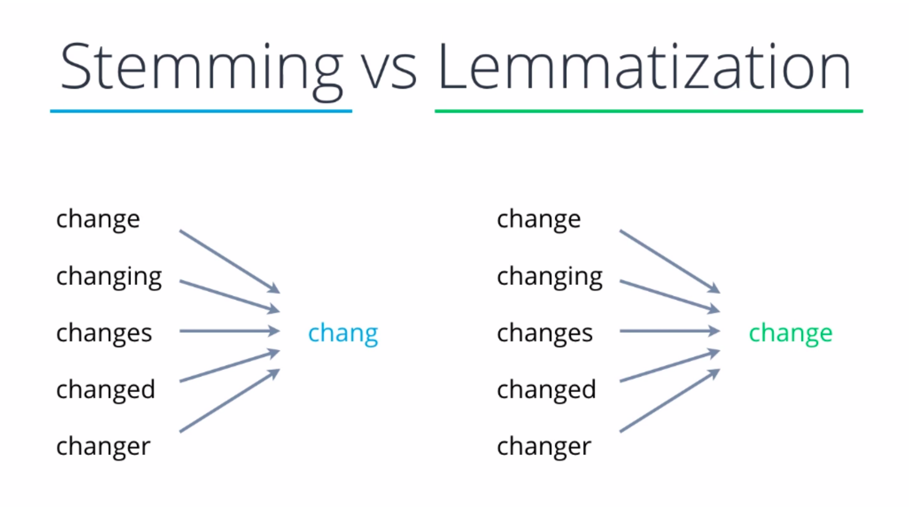

Figure 4: Stemminng vs Lemmatization

Between the two stemming is more suitable for large data sets because it's a simple algorithm and reduces the computational load. Lemmatization shines when the goal is more complex context aware outcomes.

Experiments using **stemming or lemmatization alone and even combining them** (firstly lemmatization and then the output fed into stemming function) were carried out. They resulted in minor improvements in the classifiers' accuracy when utilizing the combination, And despite the minor improvements we decided to go on and use this approach4.

	Accuracy %		
Classifier	Stemming	Lemmatization	Stem(lemmatize)
SVM	94.41	94.37	94.46
KNN (3000 instances)	86.61	86.41	86.68

Table 4: Stemming, lemmatizaton and ccombination trials

1.4 Utilities

Joblib python package was used as a convenience to save and improve the speed of execution while developing the algorithms of the following data:

1. The preprocessed train dataset
2. The test dataset Which has undergone the same exact preprocessing as the training data set.

All the data was saved before vectorization to allow different vectorization methods to be implemented.

1.5 Vectorizers

The starting vectorizer as specified by the project requirements is the "bag of words". A relative simple technique which in its default configuration separates the text word by word generating the vocabulary which appears in our train texts and then creates A sparse matrix with one row for his data instance and one column for each word in the vocabulary. In every row (data instance) it counts how many times the specific word of the vocabulary appears. Obviously, it results in a very high dimensional data structure which is not optimal for the calculations that will take place later when we try to classify things.

For best model namely logistic regression an alternative vectorizer was used named "Term frequency-inverse document frequency (TF-IDF)".

While Bag of Words treats every word as equally important, TF-IDF realizes that words like "the," "is," and "and" show up everywhere but don't tell you much about the topic. TF-IDF rewards words that are frequent in one document but rare across the rest of the collection.

TF-IDF is calculated by the equation[15]: $TF\text{-}IDF = TF \times IDF$

Where :

$$TF = \frac{\text{Number of occurrences of term } t \text{ in document } d}{\text{Total number of terms in the document } d}$$

$$IDF = \log_e \frac{\text{Total number of documents in the corpus}}{\text{Number of documents with term } t \text{ in them}}$$

1.6 Title and Content features handling

At first the title and content features were combined into one long string without any modification to give title more weight for vectorizing. We would evaluate our accuracy results and in the event that we were not satisfied by the numbers we would come back and try a different approach.

The accuracy results for all our classifiers we're above 93% so there was not motivation to create a more complex method to handle those 2 features.

2 Support vector machines (SVM)

Following the text preprocessing vectorization took place. Vectorization transforms text into numbers to enable classification methods calculations. The project required "bag of words" vectorizer to be utilized at least as a starting - comparison point for our models.

Because of the big size of a data set and also the extreme dimensionality produced by the bag of words vectorizing (vectorized matrix shape 111220 x Number of features parameter chosen) a method for using only part of the data set was utilized throughout the development of the first edition of the SVM classifier. Later on the ability of SVM to handle sparse data efficiently proved very helpful in reducing computation times..

2.1 Stratify and utilize part of dataset for development.

The intention was to keep the original distribution of data among the classes so the results wouldn't be skewed towards any class that was represented disproportionately in the number of data instances chosen. This allowed for quick runs so the algorithm, with the resulting accuracies improving step by step to the final version.

An example of this method keeping 50000 instances is given in the code bellow:

Listing 3: Keep 50000 stratified data instances implementation

```python
# Stratify and keep 50000 instances based on the 'Label' column
from sklearn.model_selection import train_test_split

# Stratify and sample 50000 instances
stratified_data, _ = train_test_split(
    dataTrain,
    train_size=50000,
    stratify=dataTrain['Label'],
    random_state=42
)
dataTrain = stratified_data.reset_index(drop=True)
print(f"Subset shape (stratified): {dataTrain.shape}")
```

2.2 Vectorizer Bag of words choices

The main adjustment in the vectorizer is the maximum number of features that we will allow it to keep. Keeping every feature it finds would result in a huge computational load. Practically when we say features we mean words or a combination of words (named ngrams).

Listing 4: BOW vectorizer implementation

```python
# Combine 'Title' and 'Content' columns into a single string
# and vectorize the result for classification
from sklearn.feature_extraction.text import CountVectorizer

# Create a new column that combines Title and Content
# (if either column is missing, fill with empty string)
dataTrain['Combined'] = dataTrain['Title'].fillna('') + ' ' +
    dataTrain['Content'].fillna('')

# Initialize CountVectorizer (Bag of Words)
bow_vectorizer = CountVectorizer(max_features=5000) # Use binary=True for
    presence/absence of words, limit to top 5000 features

# Fit and transform the combined column
dataTrain_bow = bow_vectorizer.fit_transform(dataTrain['Combined'])
```

2.2.1. Max features. The default configuration was used except setting the parameter max_features=5000 to decrease somewhat the computational load.

Practically this tells the program to build a vocabulary that only considers the top max_features ordered by term frequency across the corpus. Otherwise, all features are used[14].

There are other ways to achieve similar results like the parameters max_df and min_df which Ignore terms with strictly higher or lower frequencies than the threshold correspondingly. Trials with setting the max_df parameter to 0.9 in order to dismiss very common words like "and", etc did not show any improvement in accuracy when combined with the max features=5000 parameter.

2.2.2. *N-grams.* N-grams are contiguous sequences of n items from a given sample of text or speech. The items can be phonemes, syllables, letters, words, or base pairs according to the application. N-grams are used in various areas of computational linguistics and text analysis. They are a simple and effective method for text mining and (NLP) tasks, such as text prediction, spelling correction, language modeling, and text classification[4].The concept of an n-gram is straightforward: it is a sequence of ′n′ consecutive items. For instance, in the domain of text analysis, if ′n′ is 1, we call it a unigram; if ′n′ is 2, it is a bigram; if ′n′ is 3, it is a trigram, and so on. The larger the value of ′n′, the more context you have, but with diminishing returns on information gained versus computational expense and data sparsity.

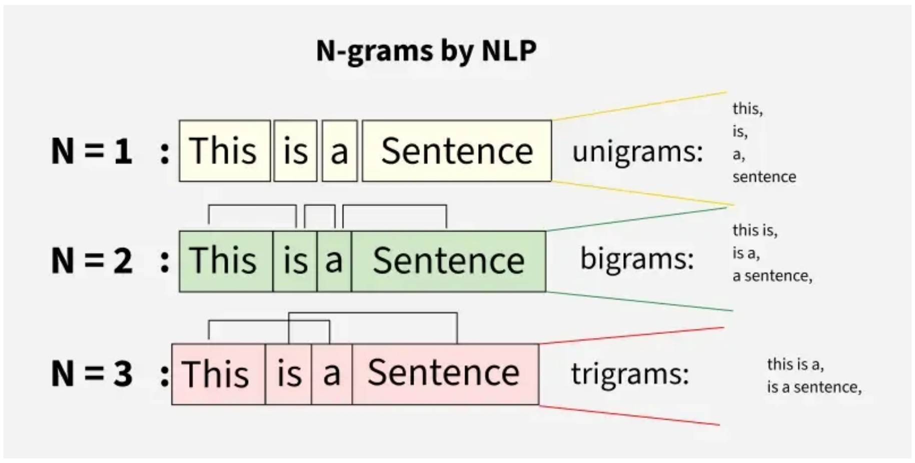
N-grams by NLP
N = 1 : This is a Sentence unigrams:
this,
is,
a,
sentence
N = 2 : This is a Sentence bigrams:
this is,
is a,
a sentence,
N = 3 : This is a Sentence trigrams:
this is a,
is a sentence,

Figure 5: Ngrams example[5]

Ngrams might be advatengeous in understanding meaning but the increase drastically the computing complexity. If you have a vocabulary of 10,000 words, you have 100 million possible bigrams and 1 trillion possible trigrams. This makes your data massive and very hard for a computer to process.

2.2.3. *SGDClassifier* . Owing to the large size of our data set typical computer hardware could not process the whole operation in one go. We experienced many reboots stemming from extreme load to the ram and the CPU when trying to do so.

Thankfully one solution is to split and process our data in chunks. The stochastic gradient descent Version of the SVM classifier from Scikit learn was chosen because it supports partial fit, and so scales computationally much better for very large data sets as expressed in scikit website[10]. Also SVM proved very nimble with sparse data matrices, produced by the vectorizer.

Gradient decent works much better and faster when data range in scaled. (without it, the gradient descent path will "bounce" back and forth between the steep changes on the loss function, the large-scale features. This makes convergence incredibly slow or impossible). Scikit learns Standard scaler was utilized.

Indeed, after testing compared to LinearSVC it was considerably faster to execute.

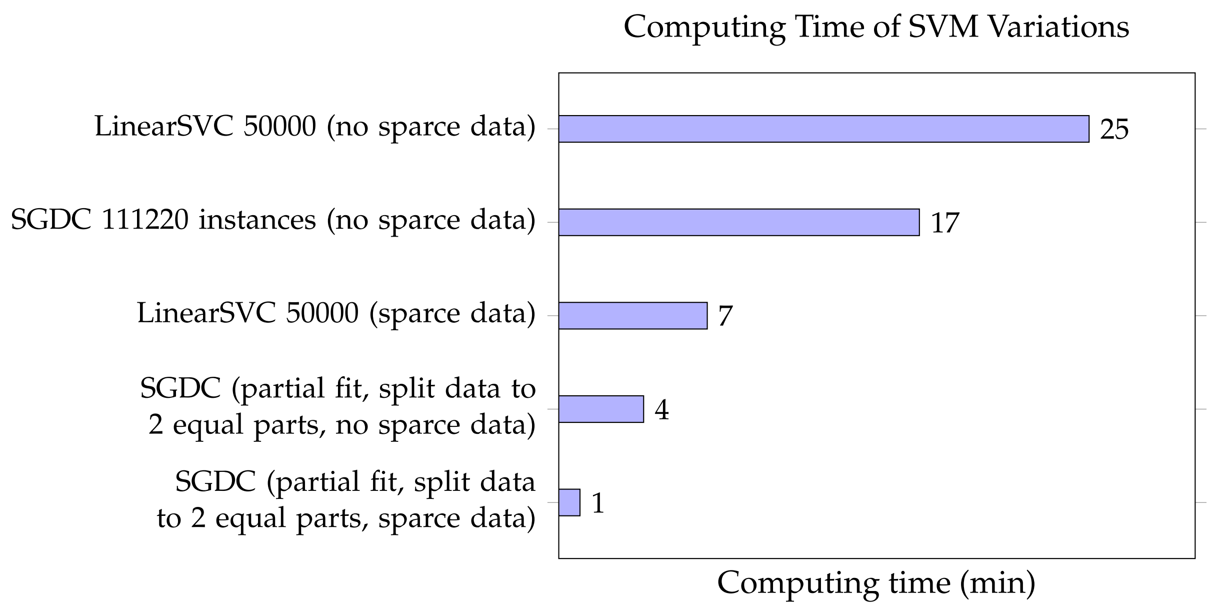

Figure 6: Computing time comparison for SVM variations.

2.2.4. SGDClassifier score. Before the advent of neural networks, support vector machines were considered some of the best classifiers out there. As such they did not disappoint achieving a fivefold cross validation accuracy of 93,15%.

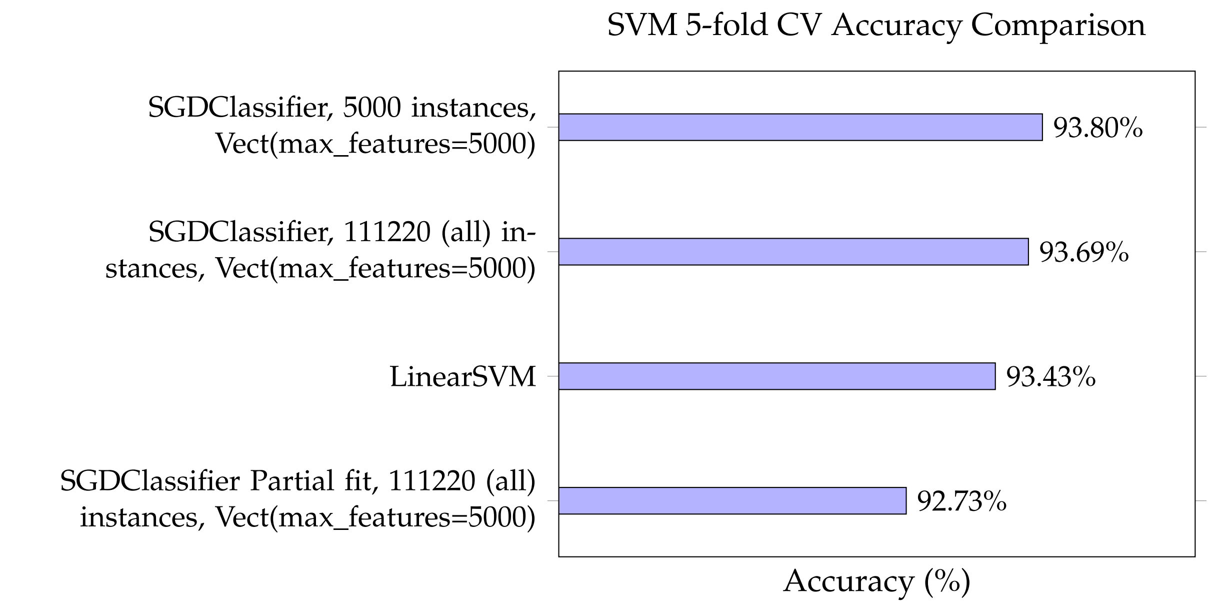

Figure 7: Performance comparison of SVM variations.

The classifiers utilizing only 50,000 of the instances obtain slightly higher accuracy scores due to less exposure to data. In reality they should be less capable in classification, with the best model being the partial fit SDGC one with exposure to all data instances.

2.2.5. SGDClassifier code. Bellow the partial fit SDGC classifier code:
Main parameters:

1. loss='hinge' , this actually signifies the SVM model in scikit learn
2. random_state=42, for code reproducibility
3. n_jobs=6, parallelizing utilizing multicore cpu

Listing 5: Partial fit SDGC classifier code

```python
# Partial fit with 5-fold cross-validation
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score

X = dataTrain_bow.toarray()
y = dataTrain['Label'].values

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_acc = []

for train_idx, test_idx in skf.split(X, y):
    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]

    # Divide train set into 2 parts
    split_idx = X_train.shape[0] // 2
    X1, X2 = X_train[:split_idx], X_train[split_idx:]
    y1, y2 = y_train[:split_idx], y_train[split_idx:]

    scaler = StandardScaler()
    X1_scaled = scaler.fit_transform(X1)
    X2_scaled = scaler.transform(X2)
    X_test_scaled = scaler.transform(X_test)

    sgd_partial = SGDClassifier(loss='hinge', random_state=42, n_jobs=6)
    classes = np.unique(y)
    sgd_partial.partial_fit(X1_scaled, y1, classes=classes)
    sgd_partial.partial_fit(X2_scaled, y2)

    pred = sgd_partial.predict(X_test_scaled)
    acc = accuracy_score(y_test, pred)
    cv_acc.append(acc)

print('Partial fit SGDClassifier 5-fold CV accuracy scores:', cv_acc)
print('Mean CV accuracy:', np.mean(cv_acc))
```

Note that using StandardScaler(with_mean=False) or MaxAbsScaler for scaling maintains the sparsity of the vector (leaving zeros unchanged), which the SVM leverages for faster computations.

3 K Nearest Neighbors with Jaccard Distance metric

3.1 Jaccard distance

The Jaccard distance is a metric used to measure how dissimilar two sets of data are. When used in a K-Nearest Neighbors (KNN) classifier, it helps the algorithm determine which data points are "closest" (most similar) to a new, unclassified point.To understand Jaccard distance, we first need to understand its counterpart: Jaccard similarity (often called the Jaccard Index).

Jaccard Similarity = (number of observations in both sets) / (number in either set) or mathematically,

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

$|A \cap B|$ is the number of elements present in both sets.$|A \cup B|$ is the total number of unique elements present in either set.The result is a score between 0 (no overlap) and 1 (identical sets).

Jaccard distance metrics need to represent how far apart things are, so a higher similarity must equal a lower distance. Therefore, the Jaccard distance $d_J(A,B)$ is simply 1 minus the Jaccard similarity:

$$d_J(A,B) = 1 - J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

A Jaccard distance of 0 means the sets are perfectly identical, while a distance of 1 means they share absolutely nothing in common.

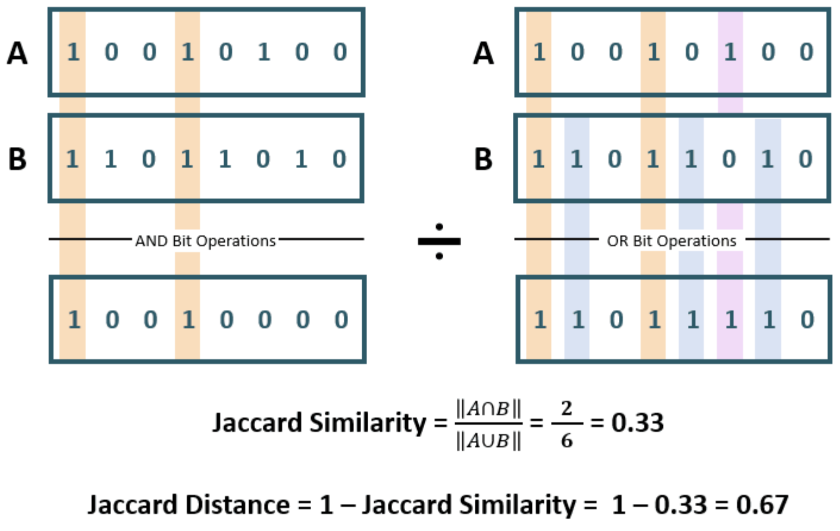

Figure 8: Example of Jaccard distance[11]

3.1.1. Jaccard advantage. Jaccard distance completely ignores mutual absences. It doesn't care about the 49,980 words that neither text has.Instead, it only looks at the Union—the pool of words actually used in text A or text B, and checks the Intersection, the words used in both.

In practice, two documents are similar because they share the same words, not because they both lack the same words.

The model benefits also computationally because it ignores all the zeros (absence of word) in the sparse matrix produced by the vectorizer for each text.

3.2 Stratify and utilize part of dataset for development.

The intention was to keep the original distribution of data among the classes so the results wouldn't be skewed towards any class that was represented disproportionately in the number of data instances chosen. This allowed for quick runs so the algorithm, with the resulting accuracies improving step by step to the final version.

The computational load with KNN with Jaccard distance classifier especially on memory (RAM) was so great but when we tried to run it on our personal hardware it would crash. Our hardware managed to execute with a maximum of 50.000 data instances, so around half off our data set.

This is really not surprising because KNN is a "lazy learner," meaning it doesn't actually "learn" a model with parameters (weights) that can be updated incrementally. Instead, it memorizes the entire training dataset.

Listing 6: Stratification code for keeping 50000 data instances

```python
 # Stratify and keep 50000 instances based on the 'Label' column
from sklearn.model_selection import train_test_split

# Stratify and sample 50000 instances
stratified_data, _ = train_test_split(
    dataTrain,
    train_size=50000,
    stratify=dataTrain['Label'],
    random_state=42
)
dataTrain = stratified_data.reset_index(drop=True)
print(f"Subset shape (stratified): {dataTrain.shape}")
```

3.3 BOW vectorizer

The input for the KNN With Jaccard distance classifier should be a sparce binary vector. Actually in scikit learn implementation even if we feed the algorithm with a vector with no binary values it automatically converts every non 0 value to 1. In our implementation which specify binary = true you need to count vectorizer parameters to get a sparce vector matrix with binary values, corresponding to the existence or absence of a specific word token.

3.3.1. Max_feautures. The same strategy as with the SVM classifier was used, so 5000 was the maximum number of top features ordered by term frequency allowed for the vectorization. Again, there are other strategies to achieve similar results, but owing to the simplicity of the max_features parameter, it was preferred.

3.3.2. Binary vector values output. As discussed earlier jaccard distance metric function with binary input vectors and that's precisely one of his strong points because it handles sparse matrices quite effectively. Consequently, vectorizer's binary parameter was set to true, producing a binary vector matrix. A 0 in the matrix means absence of this specific token (word or ngram, here ngrams were not used),and then 1 means they still can exist in that particular text.

3.3.3. Vectorizer code.

Listing 7: Count vectorizer with binary output

```python
# Combine 'Title' and 'Content' columns into a single string
# and vectorize the result for classification
from sklearn.feature_extraction.text import CountVectorizer

# Create a new column that combines Title and Content
# (if either column is missing, fill with empty string)
dataTrain['Combined'] = dataTrain['Title'].fillna('') + ' ' +
    dataTrain['Content'].fillna('')

# Initialize CountVectorizer (Bag of Words)
bow_vectorizer = CountVectorizer(binary=True, max_features=5000) # Use
    binary=True for presence/absence of words, limit to top 5000 features

# Fit and transform the combined column
dataTrain_bow = bow_vectorizer.fit_transform(dataTrain['Combined'])

# Show shape and a sample
print('Bag of Words matrix shape:', dataTrain_bow.shape)
print("dataTrain_bow sample (first 10 rows):", dataTrain_bow[:10].toarray())
print('Feature names (first 20):',
    bow_vectorizer.get_feature_names_out()[:20])
```

3.3.4. Vectorizer Output. Below that is a sample of the vectorizer output, in which we can see the "sparsity" of the binary vector matrix:

Count Vectorizer output

```
Bag of Words matrix shape:  (50000, 5000)
dataTrain_bow sample (first 10 rows):  [[0 0 0 ...  0 0 0]
[0 0 0 ...  0 0 0]
[0 0 0 ...  0 0 0]
...
[0 0 0 ...  0 0 0]
[0 0 0 ...  0 0 0]
[0 0 0 ...  0 0 0]]
Feature names (first 20):  ['aa' 'abandon' 'abdomin' 'abid' 'abil' 'abl'
'abnorm' 'aboard' 'abort''abroad' 'abrupt' 'absenc' 'absent' 'absolut'
'absorb' 'absurd' 'abu''abund' 'abus' 'academ']
```

3.4 KNN classifier

The only difference from the default configuration is the metric parameter was set to “jaccard”, with signals to the classifier to use jaccard distance. Other than that 5 nearest neighbors were used As specified by project requirements and five fold cross validation was used to obtain the mean accuracy score. The n_jobs parameter was set to 6 to parallelize the computation in available threads, and speed up the operation.

3.4.1. Code.

Listing 8: Partial fit SDGC classifier code

```python
# KNN classification with Jaccard distance and 5-fold cross-validation on Bag
    of Words features
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import jaccard_score, make_scorer
import numpy as np


# Use the same Bag of Words features as before
X = dataTrain_bow.toarray()
y = dataTrain['Label'].values

# Initialize KNN classifier with Jaccard distance (metric='jaccard')
knn_clf = KNeighborsClassifier(n_neighbors=5, metric='jaccard', n_jobs=6)

# Perform 5-fold cross-validation using accuracy
cv_scores_acc = cross_val_score(knn_clf, X, y, cv=5, scoring='accuracy',
    n_jobs=6)
print('KNN (Jaccard) 5-fold CV accuracy scores:', cv_scores_acc)
print('Mean CV accuracy:', np.mean(cv_scores_acc))
```

3.4.2. Score. Because of the computing cost not many variations of the algorithm when tested, and all of them pretty much changed only one parameter which was how many data instances were given as input for the training procedure. Even with only 10,000 data instances the accuracy reached over 90%. Subsequent trials with 50,000 data instances increased the accuracy up to 94.49%

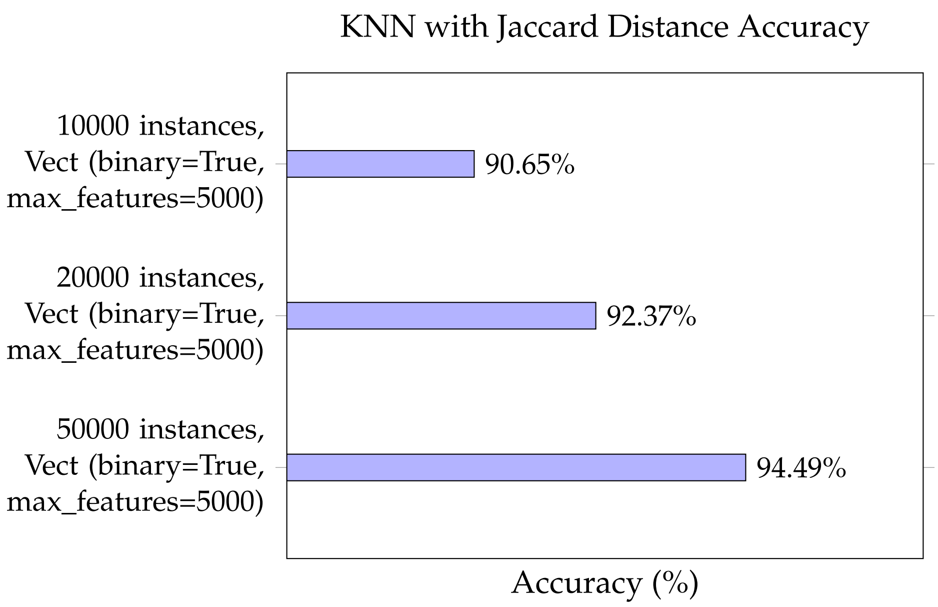

Figure 9: Accuracy comparison for KNN with Jaccard distance.

4 Best model: SGDC (SVM) with TFIDF Vectorization

4.1 TFIDF vectorizer

Bag of Words is a way to turn text into numbers. It treats a document like a "bag" full of words, ignoring grammar and word order but keeping track of frequency. TF-IDF stands for Term Frequency-Inverse Document Frequency. It fixes the BoW problem by penalizing common words and rewarding rare, meaningful ones. It is calculated by multiplying two metrics: TF: Term Frequency This is exactly like Bag of Words. It measures how often a word appears in a specific document.

$$TF(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

IDF: Inverse Document Frequency This is the "secret sauce." It measures how unique a word is across your entire collection of documents.

- If a word appears in every document (like "system" or "data"), its IDF score is very low.
- If a word appears in only one document (like "photosynthesis"), its IDF score is very high.

$$IDF(t,D) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents with term } t}\right)$$

Listing 9: TF-IDF implementation

```
from sklearn.feature_extraction.text import TfidfVectorizer
# Combine Title and Content
if 'Combined' not in dataTrain.columns:
```

```python
    dataTrain['Combined'] = dataTrain['Title'].fillna('') + ' ' +
        dataTrain['Content'].fillna('')

# Initialize TF-IDF Vectorizer
vectorizer = TfidfVectorizer(ngram_range=(1,2), max_features=100000)
dataTrain_tfidf = vectorizer.fit_transform(dataTrain['Combined'])

print('TF-IDF matrix shape:', dataTrain_tfidf.shape)
print('Feature names (first 20):', vectorizer.get_feature_names_out()[:20])
```

TFIDF Vectorizer output

```
TF-IDF matrix shape:  (111220, 100000)
Feature names (first 20):  ['aa' 'aa betty' 'aal' 'aam' 'aba' 'aback'
'abandon' 'abandoned' 'abandoned king' 'abandonment' 'abatement' 'abbas'
'abbey' 'abdomen' 'abdominal' 'abdominal obesity' 'abdominal pain'
'abduction' 'abed' 'aberration']
```

4.2 Classifier

4.1.1. Parameters tuned. Continuing our experimentation with SVM flash fires we choose the gradient descent version of the SVM from scikit learn's offerings. It proved both speedy and accurate, managing **98.94% accuracy score.**

4.2.1. Parameters.

- **alpha=0.00001**, the Regularization strength
- **loss='hinge'**, means SVM cladssifier
- **penalty='l2'**
- **max_iter=10000**
- **n_jobs=4**, utilizing multiple threads for computation speed
- **random_state=42**, reproducibility
- **learning_rate='adaptive'**, Keep the learning rate constant as long as we're making progress, but if we stall, slow down to find the optimal spot
- **tol=1e-3**, tolerance for decreasing loss impoovment, less than that and the model adjusts
- **warm_start=True**

4.2.2. Code.

Listing 10: Partial fit SDGC classifier code

```python
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import MaxAbsScaler
from sklearn.metrics import accuracy_score
import numpy as np

# 1. Keep data sparse! Avoid .toarray() to save memory
X = dataTrain_tfidf
```

```python
y = dataTrain['Label'].values

# 2. Scale the data globally before the CV loop
# MaxAbsScaler is ideal for TF-IDF as it scales features to [-1, 1]
# without removing the zeros (preserving sparsity).
scaler = MaxAbsScaler()
X_scaled = scaler.fit_transform(X)

# 3. Initialize CV and Model
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_acc = []

# Use a standard SGDClassifier with 'early_stopping' to prevent overfitting
sgd = SGDClassifier(
    alpha=0.00001,         # Regularization strength
    loss='hinge',          # Linear SVM
    penalty='l2',          # Standard regularization
    max_iter=10000,        # Maximum passes over the data
    tol=1e-3,              # Stopping criterion
    n_jobs=4,
    random_state=42,
    learning_rate='adaptive', # Adjust learning rate based on performance
    warm_start=True         # Reuse the solution of the previous fit as
        initialization
)

# 4. The CV Loop
print("Starting Cross-Validation...")
for i, (train_idx, test_idx) in enumerate(skf.split(X_scaled, y)):
    X_train, X_test = X_scaled[train_idx], X_scaled[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]

    # Standard fit (handles internal iterations automatically)
    sgd.fit(X_train, y_train)

    # Evaluation
    pred = sgd.predict(X_test)
    acc = accuracy_score(y_test, pred)
    cv_acc.append(acc)
    print(f"Fold {i+1} Accuracy: {acc:.4f}")

print("-" * 30)
print(f'Mean CV Accuracy: {np.mean(cv_acc):.4f}')
print(f'Standard Deviation: {np.std(cv_acc):.4f}')
```

4.3 Results summary and comparison with "Benchmark" models

Statistic measure	SVM(BoW)	KNN with Jaccard	OurMethod
Accuracy	94.44%	94.49%	98.94%

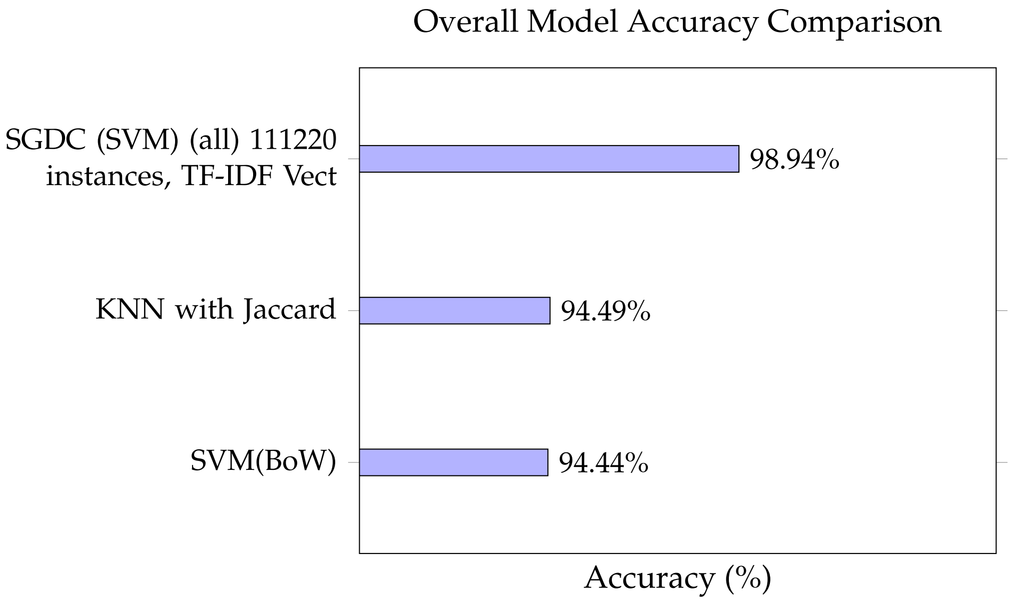

Figure 10: Overall accuracy comparison across the different primary models.

For the sakes of completeness we attempted to also experiment on this task with transformers. More specifically (eventhough it is out of this assignment's scope) we wanted to see how well a complex Neural Network is able to perform on this classification task. We tested various models and tampered with their parameters (especially of the BERT and RoBERTa families). The scores were decent but they were not able to overcome (in the testset) the performance of our other models. This along with the fact that they are much more complex and their training requires GPU and extensive runtimes makes one very careful when choosing the appropriate ML method for the task at hand.

5 Nearest Neighbor Search using Locality Sensitive Hashing

When datasets get massive in size, so does the computational load of finding its nearest neighbors. Comparing all pairs by utilizing the brute force method, produces a complexity of $O(DN^2)$, where $N = samples$ in D dimensions of the dataset [9].

This leads to a compulsory paradigm shift; methods that approximate nearest neighbor search (usually refer to as "ANN" in bibliography). Locality sensitive hashing is one of those techniques, which trades a percentage of accuracy for a massive performance gain. Many - top of the line - services we enjoy today, utilize this method for finding similarities almost instantly. For example Google, every time we type something into the search bar, performs a similarity search between the query and Google's indexed pages.

The idea is the following. Instead of comparing every pair, we can approximate and limit our search scope to only the most relevant ones - called candidate pairs. There are 3 steps in the method portrayed in Figure 11.

Firstly, the text is converted into sets by **splitting it to shingles or n-grams** as we know them from other NLP methods (for n=2 shingles the doc D_1 = abcab becomes S(D1) = {ab, bc, ca}). The next step is vectorization, more specificallym one hot encoding of the shingles into a sparse binary matrix. Now every column is a set (document, divided in shingles), and we would like to compare pairs of columns fast.

So we use a trick, we convert those columns into a small signature using a hash function which when the similarities between the columns is high it should also produce equal hash values with high probability, or to put it simply, when the hash values are equal the columns are (probably) equal. For the jacquard similarity which is used the suitable hash function is called min-hashing. **Min-hash** produces short integer vectors which represent the set name signatures. This is done by using permutations which is like shuffling the rows of our binary matrix and returning the index of the first row which has a value of 1. The number of permutations is one of the tunable parameters, more permutations, better accuracy but larger complexity. **The "magic" property of MinHash is that the probability of two MinHashed values colliding is equal to their Jaccard similarity of the original sets.** Permuting rows though is again computationally expensive so we substitute that with row hashing, and the ordering will provide us with a random row permutation.

Then **locality sensitive hashing** groups similar signatures into buckets. Signatures in the same buckets are **candidate pairs.** This is done by splitting the rows of our signature matrix now into b bands of r rows. Column pairs that hash to the same bucket >=1 band are candidate pairs. You can tune the parameters b and r in combination with the number of Minhash functions (rows in the signature table) to maximize the accuracy of the method .The MinHashLSH implementation includes a built-in optimizer will try to minimize the weighted sum of probabilities of false positive and false negative. The algorithm is a simple grid search over the space of possible parameters. So conveniently we don't need to do it manually. [3]

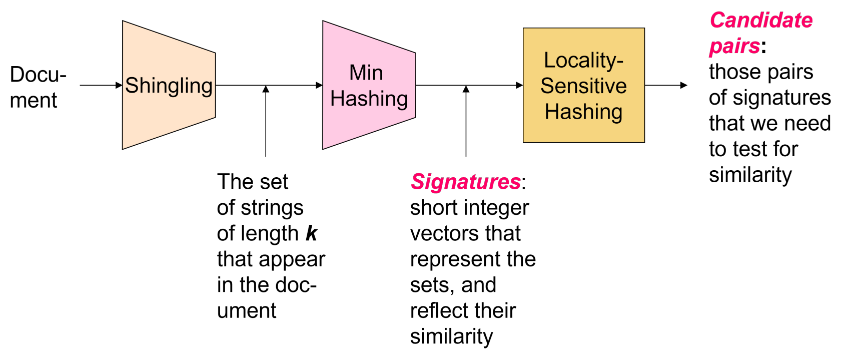
Docu-
ment
Shingling
Min Hashing
Locality-Sensitive Hashing
Candidate pairs: those pairs of signatures that we need to test for similarity
The set of strings of length k that appear in the document
Signatures: short integer vectors that represent the sets, and reflect their similarity

Figure 11: Mihash LSH steps[8]

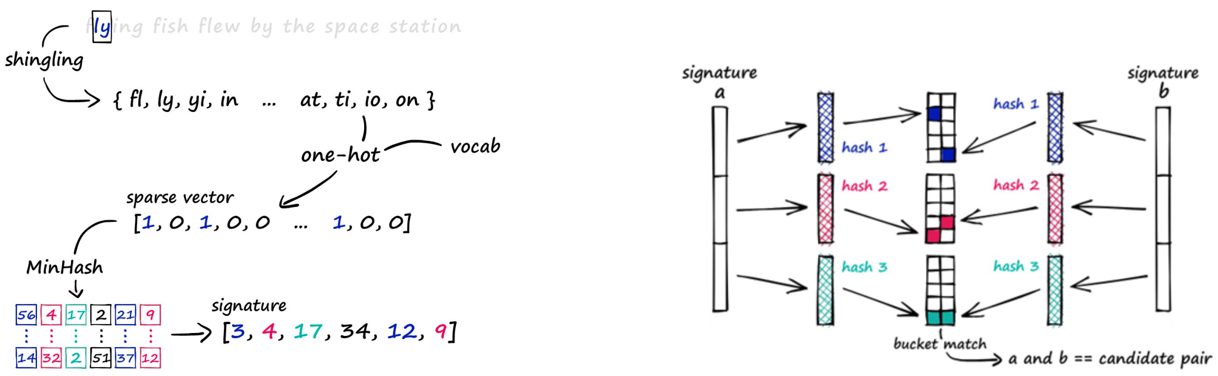

Figure 12: Shingling & MinHash (Left), and LSH buckets (Right)[12]

5.1 Our implementation

Initially in our approach, the intersection between a query vector and the candidate rows was computed using elementwise multiplication followed by a row-wise summation. This constructs an intermediate matrix representing the elementwise product before reducing it. Even in sparse form, this step introduces additional structural overhead (such as index alignment, temporary object creation, and memory allocation) which must occur before the result is performed. When this operation is repeated for thousands of queries, the accumulated cost becomes substantial.

By replacing that line with a sparse matrix–vector multiplication (@), the computation is carried out directly as a sequence of accumulations implemented in low-level compiled code. This avoids unnecessary allocations and reduces overhead, leading to a significantly lower constant factor in runtime while preserving exactly the same numerical result. As also discussed in part 3, python is by itself not a fast language. When simple loops or built-in functions are used, the computational cost might be high (comparing to a pre-compiled segment). When this is applied many times over big data, this overhead cost becomes exponentially larger.

Finally, we thought that during the Brute Force computation, there might be a window of opportunity to perform some *hardware* optimizations. Namely, by keeping the algorithm approach identical we split the set into chunks (docs) which are computed discretely by multiple CPU cores at the same time. Even though the approach is similar we were able to greatly speed up the process by using simple parallelization. Please note that the results in the next section and hence the runtimes were computed using said optimized version. Namely, the brute force method is performed in multiple threads simultaneously and the LSH methods are computed in a single-thread mode. Before we analyze the results, an important remark has to be stated; without parallelization (running the script in a single-threaded mode) and without scipy's help on the mathematical operations, the total time needed could even rech 2 or 3 hours.

5.2 Results

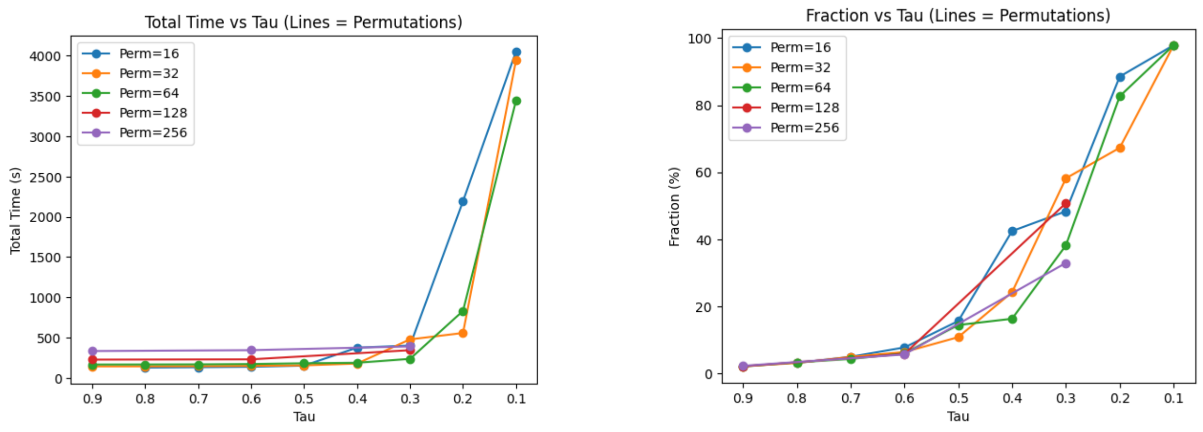

Figure 13: LSH Experiments

Our results are short of surprising; they show an expected and clear trade-off between accuracy (fraction) and efficiency (query time). When the threshold is high (e.g., $\tau = 0.9$) the fraction collapses to 2-3% for all permutation values. This indicates that the LSH method is too strict and few candidate pairs survive so most neighbors are not examined. However query time remains too low. Please note that in all experimental runs the LSH methods were computed in a single thread while Brute Force was parallelized. Remarkably, LSH was faster in almost all runs eventhough it only utilized one CPU core. Additionally, when we lower the threshold, fraction increases dramatically but simultaneously query time explodes to over 400 seconds in some cases. This is not unexpected, relacing the τ increases accuracy but pushes the method closer to brute force.

The effects of perturbing the number of permutations is also consistent. Higher values slightly improve stability. However this is not a guarantee, even at 64 permutations the $\tau = 0.9$. Finally, both k values exhibit similar behaviour indicating that performance is diven by the LSH configuration. Query times appear to be almost unchanged, as the basic cost lies in candidate generation and similarity computation. However for $k = 7$ the fractions are slightlry lower which is actually reasonable; a larger neighbourhood requires LSH to recover more neighbors. Generally, higher k values increase the difficulty.

Type	BuildTime	QueryTime	TotalTime	Fraction	Parameters
Brute-Force Jaccard	0	238.6	238.6	100%	k=7
LSH-Jaccard	90	49.14	139.15	7.7%	k=7, perm=16, tau=0.6
LSH-Jaccard	90.6	341.12	431.73	48%	k=7, perm=16, tau=0.3
LSH-Jaccard	99.45	43.78	143.23	2.2%	k=7, perm=32, tau=0.9
LSH-Jaccard	100.62	49.33	149.96	6.5%	k=7, perm=32, tau=0.6
LSH-Jaccard	100.24	414.95	515.2	58.1%	k=7, perm=32, tau=0.3
LSH-Jaccard	120.12	51.55	171.67	2.1%	k=7, perm=64, tau=0.9
LSH-Jaccard	118.53	55.41	173.94	6%	k=7, perm=64, tau=0.6
LSH-Jaccard	120.76	120.43	241.19	38%	k=7, perm=64, tau=0.3
LSH-Jaccard	157.58	68.35	225.92	2.15%	k=7, perm=128, tau=0.9
LSH-Jaccard	158.42	71.03	229.45	5.86%	k=7, perm=128, tau=0.6
LSH-Jaccard	163.40	179.87	343.27	50.8%	k=7, perm=128, tau=0.3
LSH-Jaccard	232.31	100.26	332.57	2.31%	k=7, perm=256, tau=0.9
LSH-Jaccard	238.67	104.68	343.35	5.71%	k=7, perm=256, tau=0.6
LSH-Jaccard	248.29	141.31	389.6	32.99%	k=7, perm=256, tau=0.3
Brute-Force Jaccard	0	273.12	273.12	100%	k=5
LSH-Jaccard	89.57	49.8	139.37	9.5%	k=5, perm=16, tau=0.6
LSH-Jaccard	90.01	341.59	431.61	50.8%	k=5, perm=16, tau=0.3
LSH-Jaccard	97.59	42.47	140.06	2.8%	k=5, perm=32, tau=0.9
LSH-Jaccard	99.06	49.76	148.82	8%	k=5, perm=32, tau=0.6
LSH-Jaccard	101.29	409.9	511.27	60.5%	k=5, perm=32, tau=0.3
LSH-Jaccard	115.48	50.32	165.81	2.7%	k=5, perm=64, tau=0.9
LSH-Jaccard	117.61	54.94	172.56	7.6%	k=5, perm=64, tau=0.6
LSH-Jaccard	119.59	120.12	239.72	41.2%	k=5, perm=64, tau=0.3

Table 5: Comparison of Brute-Force Jaccard and LSH-Jaccard for different numbers of permutations.

In table 5 we can see the most indicative of the experiments, one can also refer to the figure 13 for a graphical analysis of the results. There has been extensive experimentation with LSH and after carefully examining the data, we can see that when tau values become small the fractions grow larger and converge close to 100%. This happens because with very small tau values we approach the brute force approach and we need a big build and query time to do so. Generally, the candidate set size grows rapidly once the threshold drops below a certain point. What is more, increasing the number of permutations shifts the trade-off curve but does not fundamentally change the plot's behavior; tau predominantly controls how LSH operates.

6 Time Series Similarity

For this part of the assignment, we are tasked with implementing **Dynamic Time Warping**. Even though, it had already been extensively used in speech recognition research, it was introduced to the data mining and AI community by Berndt and Clifford (1994) [1]. The canonical roots of this algorithm predates the 1960s but it is still used in the Machine Learning field. The method's approach is fairly simple and mainly focuses on computing the similarity/difference between two sequences that may vary in speed or timing. A cost matrix is constructed by computing the Euclidean Distance between points of the sequences (hereinafter simply referred to as distance - in some cases other metrics might be used) with an ultimate goal of computing an optimal path which minimizes the cumulative cost. During the construction, we follow the rules of **monotonicity** (time flows forward) and **continuity** (there are no jumps - we traverse the matrix by moving to adjacent cells). The resulting path, effectively warps the time axis in the sense of computing a similarity metric even if one sequence is compressed or stretched relative to the other.

This method falls under the Dynamic Programming paradigm as the matrix is modularly created, without the need of a recursive implementation (which would almost certainly be impossible to run for large sequences). For our implementation we are meant to use the Euclidean Distance which is denoted as (for vectors **x** and **y**):

$$\text{EuclideanDistance} \equiv L_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

However, in the given test dataset, each time series is stored as a simple list containing real numbers. Our algorithm has to locally compute a distance between these single points, the Euclidean Distance is reduced to $\sqrt{(x-y)^2}$ which is mathematically equivalent to $|x-y|$, as we are talking about simple scalars living in $\mathbb{R}^1$. In the source code we submit there are three functions; appart from the main one which simply parses the arguments of input filename, output filename and the total rows (to use in a progressbar).

First of all, we have created the **getInput(filename)** function, which - given the path of dtw_test.csv (or any similar file), yields each ID and the series data dor said ID. Please note that this function is a generator. Namely, we do not load all the data into memory, rather each time a local distance is computed, we yield (using the function) the needed series points. That way we do not have to read the entire file before starting with the method and as a result we manage to avoid excessive memory usage which would probably also increase the runtime by multiple steps of swapping.

More over, we have created a **getDTW(X,Y)** function, which computes the cost matrix by calculating local distance of time series points. Finally, the **produceOutputCSV(ifpPath,ofpPath,rows=None)** function compiles all the aforementioned segments to calculate the DTW similarity for both series by iteratively yielding each line and calling the **getDTW(X,Y)** function and ultimately printing out a csv file. Please note that the optional **rows** argument is used for **tqdm** to easily print a progressbar that showcases how long the procedure will take. Let us discuss the following run:

```
python3 main.py dtw_test.csv output.csv 1001
```

It was tested on Ubuntu with a 12-threaded CPU and 16GB of RAM. The total runtime was approximately 7 minutes (439.47 seconds). Running so on Kaggle or other platforms needs more time as the resources given are stricter. Even though the runtime is not excessive for our task at hand, it feels imperative to explore ways through which we can quicken the process. Generally, there are two important caveats; the language we use and the algorithm itself.

Firstly, we should address the elephant in the room; python is not fast. As a language it was build on simplicity and not speed. Lists for example (even if of simple numbers) are not like C-arrays, rather like pointer lists and traversing through those require great resources. More over, loops take much more time than those of compiled languages (like C) as python is interpreter-based. A *hacky* way with which we can greatly speed up the process is if we use the Numba Module [6] which is a JIT compiler (based on LLVM) that pre-compiles specific functions. As a result adding "**@nb.njit**" above a function allows the compiler to *pack* said function and gravely speed up the process. Running the algorithm with this trick drops the runtime to a whoping **3.26** seconds. The results are identical.

Another *greatly different* way one could probably speed up the DTW computation is via introducing Constraints. As stated in Sakoe and Chiba (1978) [13], we can enforce a restriction to only compute cells close to the diagonal. So, with this relation:

$$|i - j| < R$$

We are able to calculate a path by avoiding multiple calculations. Note however that if the optimal path actually lies inside this band, the optimization Sakoe and Chiba introduced computes the path algorithmically (not mathematically) faster. However the restriction could skew our results by not calculating some path segments which migh belong to the optimal path. Interestingly, applying the Sakoe optimization by staying close to the diagonal returned identical results to the original DTW only for $R \leq 298$ and this happened because our series are mismatched (length-wise). So we were effectively forcing the path toward a diagonal that doesn't represent the natural alignment of unequal-length sequences. When we changed the optimization to center based on a scaled diagonal we were able to yield identical results to DTW with a much smaller $R = 40$. Because of the Numba library the runtime speedup was not actually quite visible. It went from 3.01 seconds to 2.99 seconds. In order for the optimization to actually display its true power we had to temporarily remove the Numba pre-compiled flag. The Sakoe variation was able to produce identical results to the original DTW method in **1 minute** while the vanilla-version of DTW needed **7 minutes** (this is a huge difference). In our source code one can run the Sakoe variation by calling:

```
python3 main.py dtw_test.csv output_sakoe.csv 1001 --sakoe --R 40
```

Lastly, as per the instructor's slides, appart from Sakoe-Chiba we also wanted to explore the Itakura Parallelogram Optimization [7]. Interestingly, Itakura was not able (with various slope values) to actually produce identical results to vanilla-DTW; it is generally more strict. The differences were manually inspected and most of them were not greatly appart. However, the final point remains; it was not able to compute the actual optimal path. One can run the Itakura variation by:

```
python3 main.py dtw_test.csv output_itakura.csv 1001 --itakura --s 2.5
```

As already mentioned, we proceeded with comparing the output results. The Sakoe-Chiba optimization achieved a speed-up whilst producing identical distances to the original DTW algorithm. This clearly shows that (for our task at hand) the optimal path lies within the Sakoe band (after twiching the way this band is computed - the path was not inside the diagonal). Au contraire, the Itakura Parallellogram optimization (for a slope of 2.5) was not able to produce identical results. However, the deviations that were introduced were small. Statistical comparison shows that Mean Absolute Difference (between the Itakura output and the original DTW) was only **0.0463**. This is a classic trade-off. COmparing the Itakura and the original DTW runtimes (with the Numba optimization turned off) the Itakura is completed after 2 minutes and 44 seconds while the original requires approximately 7 minutes. For large scale datasets and for specific tasks such a deviation may be bearable for a much needed speed up.

The implementation lies within the file Part3/dtw_compute.py, in there one can find both the original approach and the two optimizations. In the aforementioned paragraphs there exist examples of what arguments are needed; use accordingly. Results are also in the same file with appropriate names, saved in csv format as per the assignment's specifications.

References

[1] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In KDD Workshop, 1994. URL: `https://api.semanticscholar.org/CorpusID:929893`.

[2] Edward Loper Bird, Steven and Ewan Klein. Natural Language Processing with Python. O'Reilly Media Inc., 2009.

[3] datasketch. Minhashlsh,lsh bands and rows optimizer. `https://ekzhu.com/datasketch/documentation.html#minhash-lsh`.

[4] Deepai. `https://deepai.org/machine-learning-glossary-and-terms/n-gram`. 2026.

[5] geeksforgeeks. `https://www.geeksforgeeks.org/nlp/n-gram-in-nlp/`. 2026.

[6] Anacoda Inc. Numba: A high performance python compiler. `https://numba.pydata.org/`.

[7] F. Itakura. Minimum prediction residual principle applied to speech recognition. IEEE Transactions on Acoustics, Speech, and Signal Processing, 23(1):67–72, 1975. `doi:10.1109/TASSP.1975.1162641`.

[8] Jeff Ullman Jure Leskovec, Anand Rajaraman. Mining of massive datasets, Slides, chapter 3. Stanford University, 2014.

[9] Scikit learn. Nearest neighbor algorithms, brute force. `https://scikit-learn.org/stable/modules/neighbors.html`.

[10] Scikit learn. `https://scikit-learn.org/stable/modules/sgd.html`. 2026.

[11] Oracle. Jaccard Similarity, 2026.

[12] pinecone. Locality sensitive hashing (lsh): The illustrated guide. `https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/`.

[13] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1):43–49, 1978. `doi:10.1109/TASSP.1978.1163055`.

[14] Scikit-learn Developers. CountVectorizer Documentation, 2026.

[15] Sowmya Vajjala, Bodhisattwa Majumder, Anuj Gupta, and Harshit Surana. Practical Natural Language Processing, chapter 3. O'Reilly Media, 2020.