



澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Faculty of Science and Technology

CISC3025 – Natural Language Processing

**Project Task 2: Implementation of a Naïve-Bayes Text Classification Model and its
Performance Evaluation in Reuters Dataset**

Group Members:

Huang Yanzhen DC126732

Table of Contents

1 INTRODUCTION	3
2 BACKGROUND	3
2.1 CLASSIFICATION	3
2.2 CLASSIFICATION MODEL	3
3 DESIGNS AND APPROACHES.....	4
3.1 TOOLS & PACKAGES	4
3.2 MODEL PREPARATION	4
3.2.1 <i>The Beginning Principles: Naïve Bayes Basics</i>	4
3.2.2 <i>Json Parsing & Prior Probability Calculation</i>	6
3.2.3 <i>Pre-Processing: Tokenization, Stemming and Lowercase</i>	7
3.2.4 <i>Dictionary Construction</i>	9
3.2.5 <i>Vocabulary Construction</i>	10
3.3 MODEL CONSTRUCTION	10
3.3.1 <i>Posterior Probability of Word Type</i>	10
3.3.2 <i>Token Frequency Matrix Construction, Sorting and Feature Selection</i>	11
3.3.3 <i>Token Probability Matrix Construction</i>	12
3.3.4 <i>Model Storage Strategies and Adaption of Logarithm Space</i>	13
3.4 MODEL TESTING	14
3.4.1 <i>Decapsulate Testing Data</i>	14
3.4.2 <i>Perform Classification</i>	14
3.5 MODEL EVALUATION	15
3.5.1 <i>Precision and Recall Matrix</i>	15
3.5.2 <i>Precision and Recall Calculation</i>	16
3.5.3 <i>F₁-Score Calculation using Macro & micro-Averaging</i>	17
4 RESULTS	18
4.1 OUTPUT FILES AND EXPLANATIONS	18
4.1.1 <i>word_count.txt</i>	18
4.1.2 <i>word_dict.txt</i>	19
4.1.3 <i>word_probability.txt</i>	20
4.1.4 <i>classification_result.txt</i>	21
4.1.5 <i>Additional: f_scores.txt</i>	22
4.2 WORD PROBABILITY FLUCTUATION PHENOMENON	22
4.2.1 <i>Description of Fluctuation</i>	22
4.2.2 <i>Analysis of Phenomenon</i>	23
5 CONCLUSION.....	24

1 Introduction

In numerous domains of application, such as email filtering to combat spam and malware, and sentiment analysis of user comments on entertainment platforms, the demand for efficient and accurate text classification has become crucial in the era of the Internet.

One highly effective approach to text classification is the Naïve Bayes model. This model employs conditional probability to assign a predictive class to a given document, leveraging a substantial amount of training data. The main objective of this project is to implement, test, and evaluate a Naïve Bayes model using the Reuters test set, which classifies documents into five classes: *crude*, *grain*, *money-fx*, *acq*, and *earn*.

The Naïve Bayes classification model consists of two essential components: the prior probabilities of all classes, and the posterior probabilities of the unique word types present in the training data, considering all the classes. During the model construction phase, the prior probabilities are directly derived. However, the posterior probabilities are derived using various techniques such as string tokenization, word stemming, dictionary construction, and vocabulary construction. Subsequently, the top 10,000 words are selected from the dataset as feature words to further build a probability matrix. As a result, the model is stored as a combination of an array of prior probabilities and a posterior probability matrix.

Once constructed, the model is utilized to classify the documents in the test set, which are pre-labelled manually to a class, called the actual class. After testing, each document in the test set is assigned another class called the prediction class by the classification model, placing prediction and actuality side by side. The test results are then retrieved to evaluate the model.

The evaluation of the model involves calculating precision and recall. Precision measures the accuracy of the model's predictions while disregarding the depth of the predictions. On the contrary, recall measures the model's ability to identify all relevant instances, regardless of the accuracy of recognition. These measures are combined and balanced using the F_1 -Score. A higher F_1 -Score indicates a more accurate performance of the model on the test set.

2 Background

2.1 Classification

Classification is a supervised machine learning method where the model attempts to predict the correct label of a given test data. The model is trained using training data, and then evaluated on test data.

2.2 Classification Model

A classification model is a structured representation of the training data. This structure enables the model to establish implicit connections between patterns in the training data and the assigned classification results, which are determined manually during the training process. Training a model involves generating a program using existing data, in contrast to the traditional approach of generating data using a program.

3 Designs and Approaches

3.1 Tools & Packages

There are some python packages that are useful for fulfilling this project. Below shows a list of packages imported and the corresponding tasks they are used to perform.

Table 1: Python packages used in this project	
Package Name	Usage
json	Compile the <i>.json</i> source file into structural objects.
re	String Tokenization
nltk	Tokenizer construction; Word stemming.
pandas	Multiple matrix construction.
numpy	Some basic math tasks, like initialize an array filled with 0.
collections	Dictionary Construction.

3.2 Model Preparation

3.2.1 The Beginning Principles: Naïve Bayes Basics

The essential goal of implementing the Naïve Bayes algorithm is to predict a class for a sentence based on the current datastore.

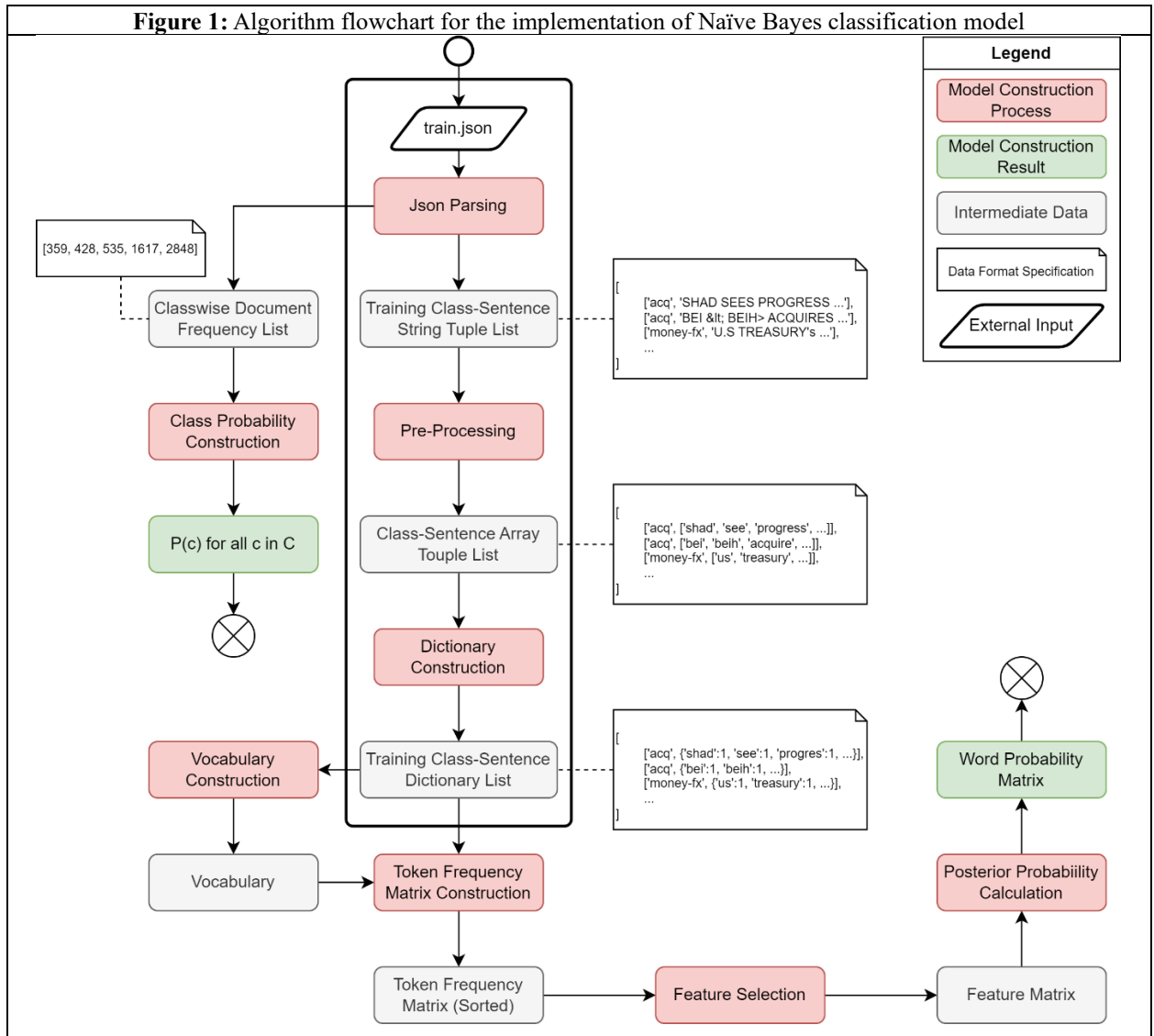
Abstractly speaking, the datastore is a corpus storing representative documents that's been labelled manually, implying some hidden patterns dictating that the abundant occurrences of some specific words in the language may somehow certainly leads to it being sorted into a specific class. To clarify this idea, the below table demonstrates the inputs and outputs of Naïve Bayes Classification Algorithm, showing how the algorithm summarizes the language features.

Table 2: Inputs and outputs of Naïve Bayes Classification Algorithm
Input:
1. A test document $d = \{x_1, x_2, \dots, x_n\}$, where x_1, x_2, \dots, x_n are words of the document.
2. A finite set of class $C = \{c_1, c_2, \dots, c_n\}$, where c_1, c_2, \dots, c_n are classes stored in the set.
3. A finite set of documents $D = \{d_1, d_2, \dots, d_n\}$, where d_1, d_2, \dots, d_n are documents stored in the document set.
4. A mapping relation $D \rightarrow C$, i.e. a training data set $T = \{(d_1, c_1), (d_2, c_2), \dots, (d_n, c_n)\}$, where $d_i \in D$ and $c_j \in C$ for all $i \in [1, D], j \in [1, C]$.
Compute:
Target Class:
$c_t(d, C, D, T) = \operatorname{argmax}_{c \in C} P(c d)$ $= \operatorname{argmax}_{c \in C} \frac{P(d c)P(c)}{P(d)}$ $= \operatorname{argmax}_{c \in C} P(d c)P(c)$ $= \operatorname{argmax}_{c \in C} P(x_1 x_2 \dots x_n c) P(c)$ $\approx \operatorname{argmax}_{c \in C} [P(x_1 c) P(x_2 c) \dots P(x_n c)] P(c)$ $= \operatorname{argmax}_{c \in C} \left[\prod_{i=1}^n P(x_i c) \right] P(c)$

Therefore, we can break the task of sentence classification into two subtasks: First, calculate the prior probability $P(c)$ of each class. Second, calculate $P(d|c)$ for each document in the training dataset by calculating the posterior probabilities of each word $P(x|c)$ in a specific document respectively and then multiplying them together. The second subtask can only hold if the assumption of independence, that is,

$$P(x_1 x_2 \dots x_n | c) = P(x_1 | c) \cdot P(x_2 | c) \cdot \dots \cdot P(x_n | c) = \prod_{i=1}^n P(x_i | c)$$

The following process will show how the model is prepared and constructed based on the idea demonstrated above. To make a long story short, using the given *json* data, this model calculates both $P(c)$ for each class $c \in C$ and $P(x|c)$ for each word type x in each class $c \in C$. The below algorithm flowchart demonstrates the model preparation process (later shown in 3.2) and the model construction process (later shown in 3.3).



3.2.2 Json Parsing & Prior Probability Calculation

Python package *json* is used to parse the training data into a structural format. Raw training data called *Training Class-Sentence String Tuple List* is retrieved after parsing. It is a list whose elements are tuples of three attributes. The first attribute is the training id, the second attribute is the class label assigned to the sentence, and the third attribute is the sentence itself. Below shows the data format of *Training Class-Sentence String Tuple List*:

Table 3: Data format of training class-sentence string tuple list	
[
['training id', 'class', 'w1 w2 w3 w4 w5 ...'],	
['training id', 'class', 'w1 w2 w3 w4 w5 ...'],	
['training id', 'class', 'w1 w2 w3 w4 w5 ...'],	
['training id', 'class', 'w1 w2 w3 w4 w5 ...'],	
...	
]	

It is observable that, within each record, we can already retrieve the class labels for calculating document frequencies for each class. There are five classes given in the training set, which in order are: *crude*, *grain*, *mondey-fx*, *acq* and *earn*. The below algorithm shows the process of calculating the document frequencies for each class.

Algorithm 1: Calculate document frequency for each class	
1	Procedure CalcClassDocFreq(trainClassSentenceStr);
2	Let classFreqArr \leftarrow zeros(1,5);
3	for instance in trainClassSentenceStr do
4	curClass \leftarrow instance.class;
5	classFreqArr[curClass] += 1;
6	Output classFreqArr.

Having document frequencies for each class, it is possible to calculate the prior probabilities of each class. The prior probability of a specific class c is the portion of documents that is labelled as class c in the training set compared to all the documents. It is equal to the number of documents in labelled in class c divide by $|D|$, the number of documents.

$$P(c) = \frac{\text{num}(\text{doc}, \text{doc.label} = c)}{\text{num}(\text{doc})}$$

Implementing this idea, the below algorithm calculates the prior probabilities of each class given as the input of the algorithm.

Algorithm 2: Calculate prior probabilities for each class	
1	Procedure CalcClassPriorProbs(classFreqArr);
2	Let classPriorProbs \leftarrow zeros(1,5);
3	Let sum \leftarrow sum(classFreqArr);
4	for index = 0...classFreqArr.length()-1 do
5	classPriorProbs[index] \leftarrow classFreqArr[index] / sum;
6	Output classPriorProbs.

At this point, we have fulfilled the first task of constructing the Naïve Bayes language classification model. The rest of the work will be focusing on calculating posterior probabilities of each unique word type extracted from the training set.

3.2.3 Pre-Processing: Tokenization, Stemming and Lowercase

Referring to 3.2.2, we have investigated the second attributes of all instances in *Training Class-Sentence String Tuple List*, i.e., the class labels of all documents in the training set. Now, we move our focus onto the documents itself. Remark that in each instance of the raw training data, the third attribute is the document in the format of plain string.

The goal of 3.2.3 is to convert each document string into a list of tokens. We want to parse each document string according to all kinds of delimiters into lists of word tokens, with each word normalized. This involves string tokenization, word stemming and lowercasing.

3.2.2.1 String Tokenizing

The key task of this phase is to determine whether a string segment is a valid word. By design, the following table shows the abbreviated version of the disambiguating rules that drives the idea of tokenization.

Table 4: Disambiguating rules to define a word				
Pattern	Examples	Validity	Interpretation	Correction
In-segment dots	U.S.	Yes	Country name abbreviation;	
	Co.	Yes	Abbreviation for “cooperation”	
	13.10	Yes	Floating point numbers	
In- segment Slashes	autumn/winter	No	Selections, i.e. autumn or winter.	‘autumn’, ‘winter’
	13-3/2	Yes	Hybrid number.	
In-segment Hyphens	government-to-government	Yes	Connected subwords forming a long word.	
In-segment Commas	2,365,000	Yes	A large number separated by a comma.	
“<” and “>”	<Banca>	No	Escape sign for “<”.	Banca
Ending with a punctuation mark	nations'	Yes	Possessive pronoun of the original noun plural.	
	policy.	No	Extra period.	policy
Brackets surrounded	(Bracket)	No		Bracket
Pure punctuation marks	...	No	Abbreviation mark	Empty Char
	--	No	Extension mark	Empty Char
...				

The tokenization process is performed in sequence by two roles considering the ideas shown above. The first rule is made to avoid in-segment symbols that connects two disjoint words. For instance, the segment “said...Taiwan” should be separated into two disjoint words. Below shows the regular expression for replacing these symbols by a space.

Table 5: Regular expression for the first rule	
Rule	<code>r'\\" \.\.+ \(\) \s--+ s (?<=[A-Za-z])/ &[a-z]+; >'</code>
Patterns	Interpretation
<code>\"</code>	Any double-quotation mark won't be a part of a valid word.
<code>\.\.+</code>	Any continuous periods always separates two disjoint words.
<code>\(\)</code>	Any brackets won't be a part of a valid word.
<code>\s--+ s</code>	Any continuous hyphens always separates two disjoint words.
<code>(?<=[A-Za-z])/</code>	A slash only separates two disjoint words but doesn't separate two numbers.
<code>&[a-z]+; ></code>	Remove the escape sign of "<", as well as the actual sign of ">".

The second rule is made to normalize different delimiters. There are all kinds of delimiters in the natural language, like a comma followed by a space, or a period followed by a space. The second rule replaces these patterns by a single space, realizing a unification.

Table 6: Regular expression for the second rule	
Rule	<code>r'(?<![A-Z])([.,?!"]\s+)'</code>
Interpretation	This rule listed many possible situations for a delimiter. Any punctuation mark followed by any number of spaces is likely to be a delimiter. However, this rule doesn't apply for abbreviations like U.S. or U.K., so these cases are included using the <code>?<![A-Z]</code> signs.

Since all delimiters are unified to be arbitrary (mostly only one) number of spaces, it is easy for a tokenizer to parse a sentence using the delimiter of `"\s"`. However, there are still space to improve: There are still some words that contain some invalid symbols at the end. For instance, there is a word like "berry,", having an extra comma. There were also invalid symbols like "--" presenting as a word string in the tokenization. Therefore, it is necessary to use the built-in `rstrip()` function to filter these invalid tokens.

Code 1: Python code for filtering invalid tokens	
<pre>for word in _cur_token_array: word = word.rstrip(' ,?!"-') # Remove extra punc marks at the end cur_token_array.append(word) if word != "-" or "" else None</pre>	

Below demonstrates the algorithm for string tokenization (where test id is abbreviated).

Algorithm 3: String tokenization	
1	Procedure tokenize(trainClassSentenceStr);
2	Let trainClassSentenceArr \leftarrow zeros(size(trainClassSentenceStr));
3	for instance in trainClassSentenceStr do
4	curClass \leftarrow instance.class;
5	curDocStr \leftarrow instance.doc;
6	curDocStr \leftarrow curDocStr. regexReplace ('\" \.\.+ \(\) \s--+ s (?<=[A-Za-z])/ &[a-z]+; >', '\s');
7	curDocStr \leftarrow curDocStr. regexReplace ('(?<![A-Z])([.,?!"]\s+)', '\s');
8	curDocArr \leftarrow curDocStr. tokenizeWithDelimiter (' ');
9	curInstance \leftarrow (curClass, curDocArr);
10	trainClassSentenceArr.append(curInstance);
11	Output trainClassSentenceArr.

Even though these rules and methods can't guarantee a 100-percent accuracy of tokenization, it is completely accurate enough for parsing the training data given. After information retrieval and tokenizing, we retrieve *Training Class-Sentence Array Tuple List*, whose data should be of the following form (training id abbreviated).

Table 7: Data format of training class-sentence array tuple list
[
['class', ['w1', 'w2', 'w3', 'w4', 'w5']],
['class', ['w1', 'w2', 'w3', 'w4', 'w5']],
['class', ['w1', 'w2', 'w3', 'w4', 'w5']],
['class', ['w1', 'w2', 'w3', 'w4', 'w5']],
...
]

3.2.2.2 Word Stemming and Lowercase

Both word stemming and lowercase are used to remove redundant differences between words, making it easier to count during 3.2.4 Dictionary Construction. Realized by using python package *PorterStemmer*, we unify tokens with a same base word but multiple different formats. For example, unifying “play” and “playing” as “play”. We then translate sequences into lowercases, which further merges multiple identical words among whom the only differences are the case of the letters.

The training data *Training Class-Sentence Array Tuple List* is now mature enough for dictionary construction.

3.2.4 Dictionary Construction

The goal of dictionary construction is to convert each document token array in the instances of *Training Class-Sentence Array Tuple List* into a dictionary, where the keys are word types in the document array (which is unique), and the values are the corresponding frequencies of occurrence of the word type. This is done using the python package *Counter*.

It has never been unreasonable to do so for Naïve Bayes classification model construction since we've assumed the independence of words in each document (illustrated in 3.2.1), which indicates that order of words doesn't matter.

After dictionary construction using *Training Class-Sentence Array Tuple List*, we gained *Training Class-Sentence Dictionary Tuple List*. Below shows the data format of this training data.

Table 8: Data format of training class-sentence dictionary tuple list
[
['class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],
['class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],
['class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],
['class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],
...
]

Note that the original token array is now replaced by the dictionary. From now on, we regard *Training Class-Sentence Dictionary Tuple List* as the formal training data.

3.2.5 Vocabulary Construction

The goal of constructing a vocabulary is to preserve a unique index for all words disregarding their frequencies, making it easier to store statistic values into the correct place.

Unlike dictionary construction, the vocabulary construction regards the scope of all instances of the training data. The vocabulary embraces every word that had ever been present in any of the documents, and only store them once. Hence, it is intuitive to consider the vocabulary as a set. The built-in *set()* function is used to perform the construction. By traversing all the instances, the vocabulary only inserts words it had never seen before into its storage. Below shows the algorithm for vocabulary construction using *Training Class-Sentence Dictionary*.

Algorithm 4: Vocabulary construction	
1	Procedure ConstructVocab(trainClassDictionaries);
2	Let vocab $\leftarrow \emptyset$;
3	for instance in trainClassDictionaries do
4	for word, frequency in instance.dictionary do
5	vocab \leftarrow vocab \cup {word};
6	Output vocab.

It's not difficult to comprehend that this process yields a better performance when the instances are processed to merge identical words into one, saving the time for traversing. By constructing the vocabulary over the training data, we again realized the advantage of dictionary construction.

3.3 Model Construction

3.3.1 Posterior Probability of Word Type

Having both the dictionary-formatted training data *Training Class-Sentence Dictionary Tuple List* and the vocabulary, we can finally work on the second task, which is to calculate $P(x|c)$ for all word types in all classes.

The posterior probability of a word x , i.e., $P(x|c)$ is a conditional probability stating that, within the scope of all documents with label class $c \in C$, the portion of word x compared to all the words in the documents. The below formula demonstrates the calculation of the probability. This formula is smoothed using Add-One method in case there are words in test set that doesn't exist in the training set.

$$P(x_i|c) = \frac{\text{num}(x_i, x_i \in d \wedge d.\text{label} = c) + 1}{\sum_{x \in V} \text{num}(x, x \in d \wedge d.\text{label} = c) + |V|}$$

To calculate $P(x|c)$, we need to find out both the numerator and the denominator. The numerator is the number of a specific word in the document labelled with the constrained class. The denominator is the number of words that belongs to the document labelled with this class.

It is obvious that we need to unify multiple tokens sparsely located in different dictionaries in the training data, summing their occurrences. Therefore, we need to construct a Token Frequency Matrix.

3.3.2 Token Frequency Matrix Construction, Sorting and Feature Selection

3.3.2.1 Token Frequency Matrix Construction and Sorting

The desired Token Frequency Matrix is of size $|V| \times 5$.

The row indexes of the matrix are the words in the vocabulary, while the column indexes of the matrix are five different classes. Traversing every instance of the training data, the program first gets the class of the instance. It then adds the frequency values corresponding to every unique word in the dictionary into the matrix. Notice that the matrix cell is located by the word-class pair. The insertion operation is performed on only one column of the matrix since there's only one class for an instance.

Below shows the abbreviated version of one possible Token Frequency Matrix. The matrix is realized using the *DataFrame* object from the *pandas* package, which by default generates index in a random order.

Table 9: The token frequency matrix (Abbreviated)					
Index\Class	crude	grain	money-fx	acq	earn
prior	4	7	1	27	501
30	33	47	34	133	300
board	6	24	12	343	192
six	32	22	106	81	286
year	0	0	0	0	526
loss	0	0	2	0	470
may	11	47	11	54	333
...					

This matrix is sorted using the sum of each column, i.e. the overall occurrences of each word token in the vocabulary. Unfortunately, due to lack of functionalities of *DataFrame*, it is required to calculate the row sum and store them in the last column during matrix construction. After sorting, the last column storing the sum of each row is sliced out. The algorithm below shows the full construction process.

Algorithm 5: Token Frequency Matrix construction	
1	Procedure ConstructTFMatrix(trainClassDictionaries, vocab, classes);
2	Let TFM \leftarrow new Matrix(row=vocab, col=classes);
3	for instance in trainClassDictionaries do
4	Let curClass \leftarrow instance.class;
5	Let curWordDict \leftarrow instance.wordDictionary;
6	for word, frequency in curWordDict do
7	TFM[word, curClass] += frequency;
8	TFM \leftarrow sortByRowSum(TFM, order=descending);
9	Output TFMMatrix.

3.3.2.2 Feature Selection

When the vocabulary is too large and full of redundant words whose time of occurrence is very few, it is necessary to select a part of it as feature words. Since the Token Frequency Matrix is already sorted descending along the index of the vocabulary by the overall occurrence of the word type, we just need to cut the matrix at the row of the threshold, in this case, 10,000.

After that, we get the abbreviated version of the Token-Frequency Matrix, called *Feature Matrix*. Below shows the static table demonstrating the feature selection process.

Table 10: Feature selection using the sorted Token Frequency Matrix							
Rank	Index\Class	crude	grain	money-fx	acq	earn	F.S.
1	the	4099	4467	6642	9867	6306	Selected
2	of	1896	2115	2656	6566	5098	
3	to	2257	2400	2946	5845	3528	
...	
10,000	62,000	0	0	0	3	0	Disregarded
10,001	anixt	0	0	0	3	0	
...	
30275	3,007,000	0	0	0	0	1	

One thing important to notice is that the feature selection doesn't affect the document frequency of each class since it only operates in the class-wise token frequency domain.

3.3.3 Token Probability Matrix Construction

The denominator for calculating $P(x|c)$ represents the word frequencies for each class. This is calculated by iterating rows of the feature matrix.

Algorithm 6: Calculate word frequencies for each class	
1	Procedure CalcClassWordFreq(featureMatrix);
2	Let classWordFreq \leftarrow zeros(1,5);
3	for wordType, frequencies in featureMatrix.rows() do
4	for index = 0...frequencies.length() - 1 do
5	classWordFreq[index] += frequencies[index];
6	Output classWordFreq.

The numerator for calculating $P(x|c)$ represents the conditional word probability regarding to a specific class. Therefore, to calculate $P(x|c)$ for all word type for all classes, we simply divide the value in the *Feature Matrix* by the corresponding value of *Word Frequencies for each Class*, given in the following algorithm, forming the *Token Probability Matrix*.

Algorithm 7: Token Probability Matrix construction	
1	Procedure ConstructTPMatrix(featureMatrix, classWordFreq);
2	Let TPMMatrix \leftarrow zeros(featureMatrix.size());
3	Let vocabSize \leftarrow featureMatrix.height();
4	for wordType, frequencies in featureMatrix.rows() do
5	for index = 0...frequencies.length() - 1 do
6	TPMMatrix[wordType, index] \leftarrow (featureMatrix[wordType, index] + 1) / (classWordFreq[index] + vocabSize)
7	Output TPMMatrix.

3.3.4 Model Storage Strategies and Adaption of Logarithm Space

3.3.4.1 Store the Model

As is mentioned, the Naïve Bayes classification model consists of two parts. In 3.2.2, we completed the first part of calculating prior probabilities:

$$P(c) \text{ for } \forall c \in C.$$

In 3.3.3, we completed the second part of calculating the posterior probabilities:

$$P(x_i|c_j) \text{ for } \forall x_i \in d, \forall c_j \in C, \text{ for } \forall d \in D.$$

The prior probabilities are stored in the form of an array (*Word Frequencies for each Class*), the posterior probabilities are stored in the form of a $|V| \times 5$ matrix (*Token Probability Matrix*). In this project, they are stored separately: The former is stored in the first line of *word_count.txt*, while the latter is stored in *word_probability.txt*.

3.3.4.2 Precision Issue

It is essential to mention that the precision of the posterior probabilities is not enough to perform the actual classification tasks. Below is a practical example.

According to the storage of *Token Probability Matrix* in *word_dict.txt*,

$$P(\text{the}|\text{money-fx}) \approx 0.0630708467045174$$

It is one of the largest probabilities at a glance, yet it is still lower than 0.1. Consider testing this model using a test sentence of 30 words, where the probabilities of all the words in the class “money-fx” are about this value. By the assumption of independence, the conditional probability of the test document would be:

$$0.0630708467045174^{30} \approx 9.8823411 \times 10^{-37}$$

This number is even less than the machine epsilon of 64-bit floating point storage, whose value is $2^{-52} \approx 2.22 \times 10^{-16}$. What’s worse is that most of the test sentences are over 30 words, meaning that if we directly use the actual value for model testing, only zeros will be assigned to the test sentence. Therefore, the *Token Probability Matrix* for model testing is stored in logarithmic space instead.

To sum up, to cope with precision issues, the posterior probability (*Token Probability Matrix*) part is stored simultaneously in two parts: *output/word_probability.txt*, stored in plain values for visualization; and *temp_output/word_probability_log.txt*, stored in negative logged manner for model testing. The relationships of the corresponding values in the two files are:

$$P_{\log}(x|c) = -\log_e[P_{\text{plain}}(x|c)]$$

3.4 Model Testing

3.4.1 Decapsulate Testing Data

Testing data is stored in `/data/test.json`. To decapsulate data, simply apply the same technique illustrated from 3.2.1 to 3.2.4. After decapsulation, we retrieved the test data similar to *Training Class-Sentence Dictionary Tuple List*, called *Testing Class-Sentence Dictionary Tuple List*. It is of form:

Table 11: Data format of testing class-sentence dictionary tuple list	
[
['test id', 'class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],	
['test id', 'class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],	
['test id', 'class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],	
['test id', 'class', {'w1':3, 'w2':4, 'w3':2, 'w4':1, 'w5':1}],	
...	
]	

3.4.2 Perform Classification

As illustrated in 3.2.1, to predict the class for document:

$$d = w_1 w_2 w_3 \dots w_n$$

Target class would be:

$$c_t(d, C, D, T) = \operatorname{argmax}_{c \in C} [P(w_1|c) \times P(w_2|c) \times \dots \times P(w_n|c)] \times P(c)$$

The below algorithm demonstrates the classification process.

Algorithm 8: Naïve Bayes classification	
1	Procedure NBClassify(classPriorProbs, TPMatrix, testClassSentenceDictionaries);
2	Let trainVocabSize \leftarrow TPMatrix.height();
3	Let testRecords \leftarrow []
4	for instance in testClassSentenceDictionaries do
5	Let joinedProbs \leftarrow zeros(1,5);
6	curTestID \leftarrow instance.testID;
7	curActualClass \leftarrow instance.class;
8	curWordDict \leftarrow instance.wordDictionary;
9	for index = 0...classPriorProbs.length() - 1 do
10	Let curJoinedProb \leftarrow classPriorProbs[index];
11	for word, frequency in curWordDict do
12	if word \in TPMatrix.indexes() do
13	curJoinedProb *= TPMatrix[word, index] ^{frequency} ;
14	else do
15	curJoinedProb *= $\frac{1}{\text{classPriorProbs}[\text{index}] + (\text{trainVocabSize} + 1)}$;
16	joinedProbs.append(curJoinedProb);
17	curPredictClass \leftarrow argmax(joinedProbs);
18	curTestRecord.append(curTestID, (predictClass, curActualClass));
19	Output testRecords.

Again, all the multiplication is done in log space. That is,

$$\begin{aligned}
P(w_1 w_2 \dots w_n | c) P(c) &\approx [P(w_1 | c) \times P(w_2 | c) \times \dots \times P(w_n | c)] P(c) \\
&\Leftrightarrow -\log_e [P(w_1 w_2 \dots w_n | c) P(c)] \\
&\approx [-\log_e P(w_1 | c) - \log_e P(w_2 | c) - \dots - \log_e P(w_n | c)] - \log_e P(c)
\end{aligned}$$

Correspondingly,

$$c_t(d, C, D, T) = \operatorname{argmin}_{c \in C} [-\log_e [P(w_1 w_2 \dots w_n | c) P(c)]]$$

This model finishes in $O(|\text{test set}| \times |C| \times |d|)$ time. The algorithm outputs a comparable predict-actual class pairs, indexed by the test id embedded in the test set. Below demonstrates the data format of test results.

Table 12: Data format of test results	
[
['test id', ['predict class', 'actual class']],	
['test id', ['predict class', 'actual class']],	
['test id', ['predict class', 'actual class']],	
['test id', ['predict class', 'actual class']],	
...	
]	

The test results will be further forwarded for model evaluation.

3.5 Model Evaluation

F_1 -Score is going to be the evaluation metric for this Naïve Bayes model. To implement the F_1 -Score measurement, we need to first calculate the overall precision and recall. This project implements both macro-average and micro-average methods for overall precision and recall calculations.

3.5.1 Precision and Recall Matrix

The goal of this step is to build a matrix to record that, for each document in the test set, the number of correct predictions. To be more specific, the row indexes of the matrix are the predicted classes; The column indexes of the matrix are the actual classes. Each cell records "The number of predictions on document with label <column index> that's been classified into <row index>."

Below shows a sample of *Precision and Recall Matrix*.

Table 13: Precision and recall matrix					
prediction\actual	crude	grain	money- fx	acq	earn
crude	172	0	0	5	4
grain	1	147	0	1	0
money-fx	1	0	176	3	2
acq	3	0	0	698	28
earn	3	1	1	3	1049

The algorithm for generating this matrix is shown below.

Algorithm 9: Generate precision and recall matrix	
1	Procedure ConstructPRMat(testResults);
2	Let PRMat \leftarrow zeros(5,5);
3	for instance in testResults do
4	curCompare \leftarrow instance.compare;
5	prediction, actual \leftarrow curCompare[0], curCompare[1];
6	PRMat[prediction, actual] += 1;
7	Output PRMat.

3.5.2 Precision and Recall Calculation

For each class $c \in C$,

$$precision = \frac{\# \text{ of correct predictions}}{\# \text{ of predictions}} = \frac{\# \text{ pred} = c, \text{ actual} = c}{\# \text{ pred} = c}$$

$$recall = \frac{\# \text{ of correct predictions}}{\# \text{ of facts}} = \frac{\# \text{ pred} = c, \text{ actual} = c}{\# \text{ actual} = c}$$

Which could be geometrically interpreted as below.

Table 14: Geometric interpretation					
prediction\actual	crude	grain	money- fx	acq	earn
crude	172	0	0	5	4
grain	1	147	0	1	0
money-fx	1	0	176	3	2
acq	3	0	0	698	28
earn	3	1	1	3	1049

$$\text{Horizontal Line: } P_{\text{money-fx}} = \frac{176}{1+0+176+3+2} = 0.9670329670$$

$$\text{Vertical Line: } R_{\text{money-fx}} = \frac{176}{0+0+176+0+1} = 0.9943502825$$

By applying this method for each class, we get a *Precision Array* and a *Recall Array*. Having these arrays, we can calculate the overall precision and recall using F_1 -Score. For each individual class, the F_1 -Score is:

$$F_c = \frac{2PR}{P + R}$$

3.5.3 F_1 -Score Calculation using Macro & micro-Averaging

3.5.3.1 Macro-Averaging method

To apply the Macro-Averaging method, we compute F_1 -Score for each class using individual precisions and recalls, then average the individual F_1 -Scores.

The algorithm for computing Macro-Averaging F_1 -Score is shown below.

Algorithm 10: Macro-averaging F_1 -Score	
1	Procedure CalcMacroFScore(precisions, recalls);
2	Let FScores \leftarrow zeros(1,5);
3	for index = 0...precisions.length() do
4	curPrec \leftarrow precisions[index];
5	curRec \leftarrow recalls[index];
6	FScores[index] $\leftarrow \frac{2 \times \text{curPrec} \times \text{curRec}}{\text{curPrec} + \text{curRec}}$;
7	FScoreMacro \leftarrow Average(FScores);
8	Output FScoreMacro.

3.5.3.2 Micro-Averaging Method

To apply the Micro-Averaging method, we first average individual precisions and recalls forming an aggregate precision and recall. Then, we use the aggregate values compute F_1 -Score.

Below is the algorithm to compute micro-Averaging F_1 -Score.

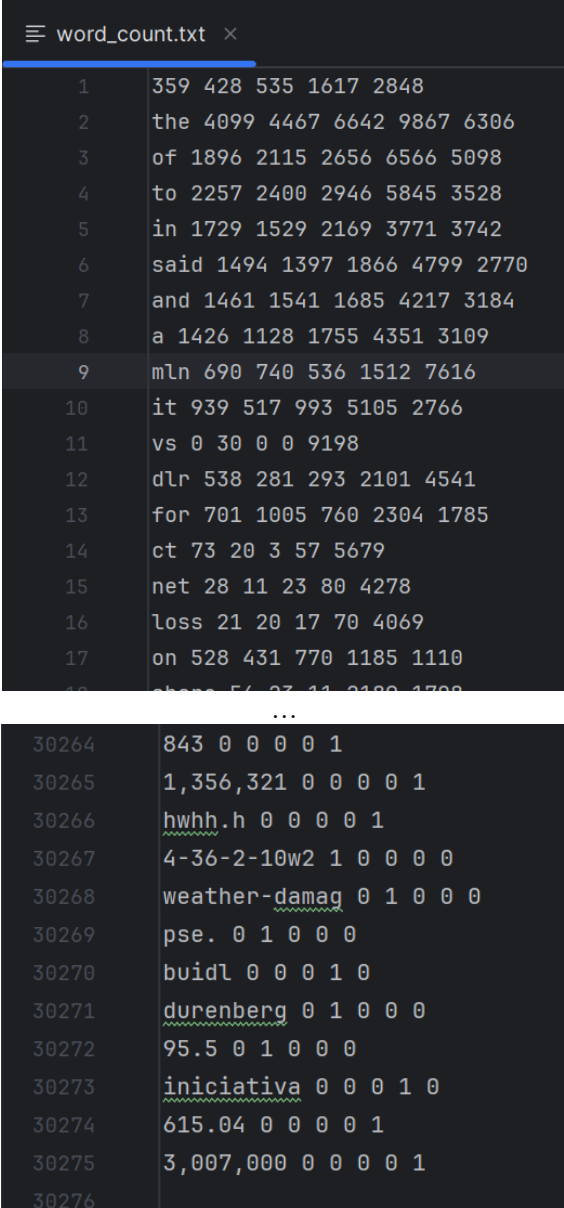
Algorithm 11: Micro-averaging F_1 -Score	
1	Procedure CalcMicroFScore(precisions, recalls);
2	avgPrec \leftarrow Average(precisions);
3	avgRec \leftarrow Average(recalls);
4	FScoreMicro $\leftarrow \frac{2 \times \text{avgPrec} \times \text{avgRec}}{\text{avgPrec} + \text{avgRec}}$;
5	Output FScoreMicro.

The successful computation of the two F_1 -Scores marks the end to model evaluation.

4 Results

4.1 Output Files and Explanations

4.1.1 word_count.txt

Table 15	
Result	Explanation
 <pre> word_count.txt 1 359 428 535 1617 2848 2 the 4099 4467 6642 9867 6306 3 of 1896 2115 2656 6566 5098 4 to 2257 2400 2946 5845 3528 5 in 1729 1529 2169 3771 3742 6 said 1494 1397 1866 4799 2770 7 and 1461 1541 1685 4217 3184 8 a 1426 1128 1755 4351 3109 9 mln 690 740 536 1512 7616 10 it 939 517 993 5105 2766 11 vs 0 30 0 0 9198 12 dlr 538 281 293 2101 4541 13 for 701 1005 760 2304 1785 14 ct 73 20 3 57 5679 15 net 28 11 23 80 4278 16 loss 21 20 17 70 4069 17 on 528 431 770 1185 1110 18 share 56 27 11 2180 1700 ... 30264 843 0 0 0 0 1 30265 1,356,321 0 0 0 0 1 30266 hwhh.h 0 0 0 0 1 30267 4-36-2-10w2 1 0 0 0 0 30268 weather-damag 0 1 0 0 0 30269 pse. 0 1 0 0 0 30270 buidl 0 0 0 1 0 30271 durenberg 0 1 0 0 0 30272 95.5 0 1 0 0 0 30273 iniciativa 0 0 0 1 0 30274 615.04 0 0 0 0 1 30275 3,007,000 0 0 0 0 1 30276 </pre>	<p>First Line: Document Frequencies for all classes.</p> <p>Following Lines: Word Frequency for each word type for each class.</p>

4.1.2 word_dict.txt

Result		Explanation
<pre> ≡ word_dict.txt × 1 72788 73010 95321 190880 208678 2 the 4099 4467 6642 9867 6306 3 of 1896 2115 2656 6566 5098 4 to 2257 2400 2946 5845 3528 5 in 1729 1529 2169 3771 3742 6 said 1494 1397 1866 4799 2770 7 and 1461 1541 1685 4217 3184 8 a 1426 1128 1755 4351 3109 9 mln 690 740 536 1512 7616 10 it 939 517 993 5105 2766 11 vs 0 30 0 0 9198 12 dlr 538 281 293 2101 4541 13 for 701 1005 760 2304 1785 14 ct 73 20 3 57 5679 15 net 28 11 23 80 4278 16 loss 21 20 17 70 4069 17 on 528 431 770 1185 1110 18 share 56 23 11 2189 1708 19 year 290 374 233 323 2558 20 from 498 442 454 823 1542 ... 9990 kb 0 0 0 1 2 9991 1,407,000 0 0 0 0 3 9992 redeploy 0 0 0 3 0 9993 <u>malmgreen</u> 3 0 0 0 0 9994 omer 0 0 3 0 0 9995 3,100,000 0 0 0 0 3 9996 aids-rel 0 0 0 1 2 9997 <u>gemeinwirtschaft</u> 0 0 0 0 3 9998 bill' 0 0 3 0 0 9999 <u>voyag</u> 1 2 0 0 0 10000 62,000 0 0 0 0 3 10001 <u>anixt</u> 0 0 0 3 0 10002 </pre>		<p>First Line: Word Frequencies for all classes.</p> <p>Following Lines: Word Frequency for each word type for each class, where the words are top 10,000 most frequent.</p>

4.1.3 word_probability.txt

Table 17.1

Result

1	0.06203559702782098 0.07395887333678935 0.0924485916709867 0.279419388284085 0.492137549680318
2	the 0.049521692917190066 0.053819005287946135 0.0630708467045174 0.04912703430628226 0.028842152437658055
3	of 0.022912841820465747 0.025488141268866164 0.02522643981543019 0.03269327465437329 0.023317922194326688
4	to 0.027273166489515895 0.0289210903528108 0.027979796061751133 0.029103834875813347 0.016138252093308273
5	in 0.020895738718716785 0.018429516134860695 0.020602700187987773 0.018778594791578507 0.01711688228542161
6	said 0.018057300222243694 0.016839518664402125 0.017725917627176575 0.023896409066695875 0.012671889076383459
7	and 0.01765871098656875 0.01857406135944784 0.01600744355619695 0.020998969467358998 0.014565126924677487
8	a 0.017235964827519566 0.013599296546573676 0.016672046788067524 0.021666077553804258 0.014222149853609728
9	mln 0.008346217025799595 0.008925667618256063 0.005098456221635683 0.0075323472745647616 0.03483283258564157
10	it 0.011353753985892357 0.00623953552801166 0.00943736589256214 0.025419805144697737 0.012653596923259845
11	vs 1.2078461687119528e-05 0.00037340849685011865 9.494331883865333e-06 4.9784185555616406e-06 0.04206737914603083
12	d1r 0.006510290849357426 0.0033968127777978534 0.0027913335738564077 0.010464635803790569 0.020770739871863467
13	for 0.00847908010435791 0.012117707994555462 0.007225186563621518 0.011475254770569581 0.00816744636969356
14	ct 0.0008938061648468451 0.00025295414302749973 3.797732753546133e-05 0.00028874827622257516 0.025974857435531594
15	net 0.0003502753889264663 0.0001445452245871427 0.00022786396521276797 0.0004032519030004929 0.01956803080398586
16	loss 0.00026572615711662964 0.00025295414302749973 0.000170897973989576 0.00035346771744487645 0.018612265803277038
...	
9987	fao 1.2078461687119528e-05 4.818174152904757e-05 9.494331883865333e-06 4.9784185555616406e-06 4.57303828090345e-06
9988	upstream 3.623538506135859e-05 1.2045435382261892e-05 9.494331883865333e-06 4.9784185555616406e-06 9.1460765618069e-06
9989	bleak 4.831384674847811e-05 1.2045435382261892e-05 9.494331883865333e-06 4.9784185555616406e-06 4.57303828090345e-06
9990	rheto1 1.2078461687119528e-05 1.2045435382261892e-05 3.797732753546133e-05 4.9784185555616406e-06 4.57303828090345e-06
9991	elcor 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 4.9784185555616406e-06 1.82921531236138e-05
9992	khartoum 1.2078461687119528e-05 1.2045435382261892e-05 3.797732753546133e-05 4.9784185555616406e-06 4.57303828090345e-06
9993	dime 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 4.9784185555616406e-06 1.82921531236138e-05
9994	fate 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 1.991367422246562e-05 4.57303828090345e-06
9995	alpin 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 1.4935255666684921e-05 9.1460765618069e-06
9996	struther 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 4.9784185555616406e-06 1.82921531236138e-05
9997	ltc 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 9.956837111123281e-06 1.3719114842710349e-05
9998	5-3/4 2.4156923374239057e-05 1.2045435382261892e-05 2.8482995651595996e-05 4.9784185555616406e-06 4.57303828090345e-06
9999	scm 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 1.4935255666684921e-05 9.1460765618069e-06
10000	449,000 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 4.9784185555616406e-06 1.82921531236138e-05
10001	10.75 1.2078461687119528e-05 1.2045435382261892e-05 9.494331883865333e-06 9.956837111123281e-06 1.3719114842710349e-05

Table 17.2

Explanation

First Line: Class Probability $P(c)$ for each class.

Following Lines: Word Probability $P(x|c)$ for each word in each class.

4.1.4 classification_result.txt

Result		Explanation
<div> <div>≡ classification_result.txt ×</div> <div> <div>1 test/14975 earn</div> <div>2 test/21067 crude</div> <div>3 test/20081 money-fx</div> <div>4 test/21040 acq</div> <div>5 test/16228 earn</div> <div>6 test/18689 crude</div> <div>7 test/21462 acq</div> <div>8 test/16833 crude</div> <div>9 test/15255 acq</div> <div>10 test/20548 earn</div> <div>11 test/19325 acq</div> <div>12 test/21406 earn</div> <div>13 test/19165 grain</div> <div>14 test/19048 acq</div> <div>15 test/16705 earn</div> <div>16 test/21063 acq</div> <div>...</div> <div>2286 test/16762 crude</div> <div>2287 test/19360 acq</div> <div>2288 test/21282 acq</div> <div>2289 test/16040 earn</div> <div>2290 test/21210 acq</div> <div>2291 test/18166 earn</div> <div>2292 test/20725 acq</div> <div>2293 test/20945 acq</div> <div>2294 test/21482 crude</div> <div>2295 test/15800 earn</div> <div>2296 test/16429 crude</div> <div>2297 test/15429 earn</div> <div>2298 test/15271 grain</div> <div>2299</div> </div> </div>		<p>All Lines: Test ID and the corresponding prediction classes.</p>

4.1.5 Additional: *f_scores.txt*

Table 19.1

Result

≡ f_scores.txt ×

1

Precisions: 0.9502762430939227 0.9865771812080537 0.967032967032967 0.9574759945130316 0.9924314096499527

2

Recalls: 0.9555555555555556 0.9932432432432432 0.9943502824858758 0.9830985915492958 0.9686057248384118

3

Avg Prec: 0.9707587590995855

4

Avg Rec: 0.9789706795344765

5

F-Scores: 0.9529085872576178 0.9898989898989898 0.9805013927576602 0.9701181375955525 0.9803738317757008

6

Macro-Average F-Score: 0.9747601878571042

7

Micro-Average F-Score: 0.9748474257285226

8

Table 19.2

Explanation

Precisions: Precisions for all classes.

Recalls: Recalls for all classes.

Avg Prec: Average precision of all classes.

Avg Rec: Average recall of all classes.

F_1 -Scores: Individual F_1 -Scores of all classes.

Macro-Average F_1 -Score: F_1 -Score gained using Avg Prec and Avg Rec.

Micro-Average F_1 -Score: F_1 -Score gained by averaging Individual F_1 -Scores.

4.2 Word Probability Fluctuation Phenomenon

4.2.1 Description of Fluctuation

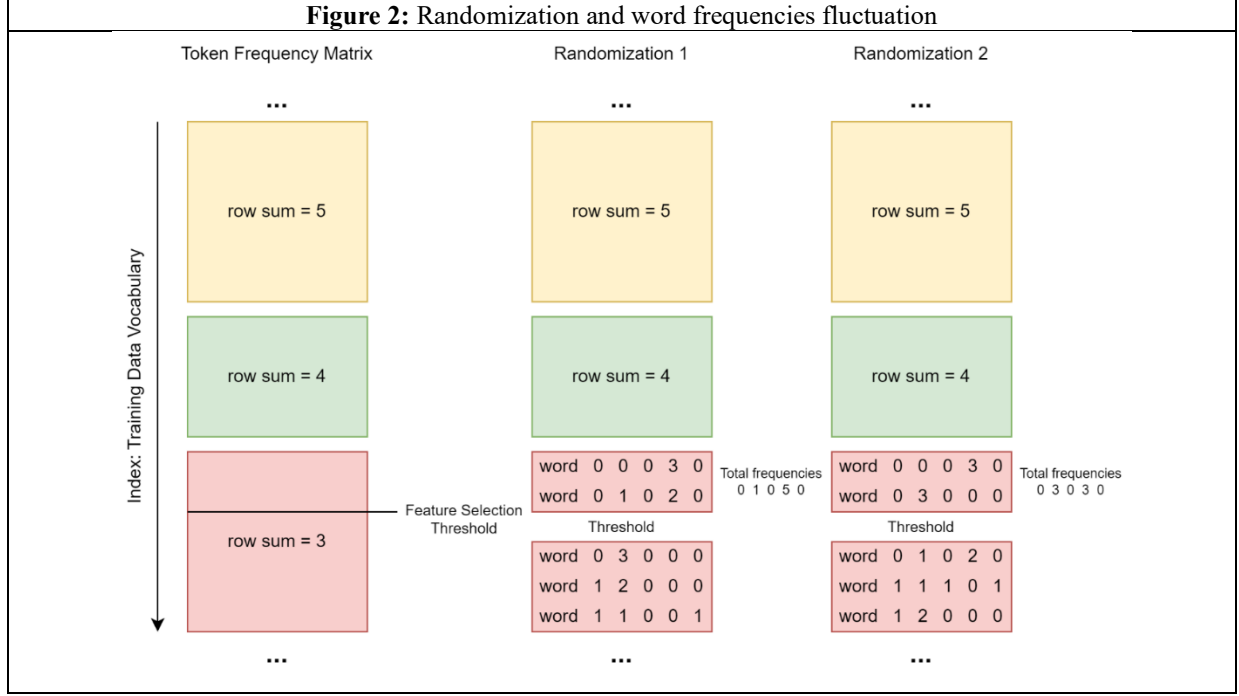
If the model is constructed multiple times, there would be a phenomenon that the word probabilities $P(x|c)$ may fluctuate a little bit. Below is a table demonstrating a comparison.

Table 20: Comparison of word probabilities before and after a duplicated model construction

Before	After
0.06203559702782098 0.07	0.06203559702782098 0.07
the 0.049521692917190066	the 0.04951930044929707
of 0.022912841820465747	of 0.02291173486641867 0
to 0.027273166489515895	to 0.027271848881588484
in 0.020895738718716785	in 0.02089472921397169 0

This fluctuation is caused by the randomization of the *DataFrame* object during feature selection. Specifically, although the *Token Frequency Matrix* (3.3.2, 3.3.3) is sorted using row sums, it remained a random order within a block of instances with the same row sum. That is, rather than saying that we are sorting rows in *Token Frequency Matrix*, it's better to say that we sort blocks in *Token Frequency Matrix* where each row in the blocks share the same row sum, where, again, within each block, rows are randomly listed.

The problem is that, although rows within each block have the same row sum, it's possible for them not to share the same frequency distribution in each class. The threshold of feature selection is 10,000, which is in the middle of the block with row sum of 3. For this special block, only rows above 10,000 will be accepted. Being re-ordered by the randomization algorithm of *DataFrame*, every time the model is constructed, the frequency distribution of each class for rows in this special block above 10,000 may change. Below shows a simplified version of this phenomenon for better clarity.



However, despite there are fluctuations in word probabilities, the changes are too minor to affect the final classification results, leaving the classification results and the F_1 -Scores unchanged. Below shows a demonstration on the F_1 -Scores before and after a duplicated model construction. To differentiate the two, model construction time is printed in the console.

Table 21: Comparison of F_1 -scores before and after a duplicated model construction	
Before	After
Macro-Average F-Score: 0.9747601878571042	Macro-Average F-Score: 0.9747601878571042
Micro-Average F-Score: 0.9748474257285226	Micro-Average F-Score: 0.9748474257285226
Current Time: 2024-03-12 11:03:54.373008	Current Time: 2024-03-12 11:08:57.767167

4.2.2 Analysis of Phenomenon

The filtered block with row sum of 3 starts from row 7935, having about 2,000 records. Yet it's not always true to say that feature selections as such won't affect classification results since the following two extreme situations are still possible.

Firstly, the block which the threshold slices is important. If the threshold cuts a block with a row sum that's relatively high, say, 2,000, there is a maximum flexibility window of 0~2000 for a single word at a single class, potentially causing a large fluctuation.

Secondly, the place within each block where the threshold slices also play a big role. Assume that the feature selection filters 2,000 rows in a block with row sum of 3 above it. It's possible that the total word frequency of a specific class will have an offset of 2,000.

However, there's no need to intentionally avoid randomization to prevent fluctuations. Randomization is important for the feature selection since word frequency, i.e., row sum in *Token Frequency Matrix*, is the only criteria for filtering. Randomization and multiple-time model construction ensures a fair chance of all the words within a block to be selected, making the model more representative and useful.

5 Conclusion

In this project, I first implemented the Naïve Bayes model using the training data. I then tested the model using the test data, getting the classification results. Lastly, I used the classification results to evaluate the model using F_1 -Score. For analytical purposes, the model is constructed multiple times and the robustness against limited randomizations is verified.

The model consists of two parts: The *Class Prior Probabilities* and the *Word Posterior Probabilities*. Correspondingly, the model implementation is divided into two steps: Model Preparation and Model Construction. *Class Prior Probabilities* is derived in the first step, while the *Word Posterior Probabilities* is derived in the second step.

In Model Preparation, the *Class Prior Probabilities* is already derived. In addition, sentence tokenizing and word stemming techniques are used to extract suitable word tokens from the documents. Moreover, counter techniques are used to convert document token array into document type-frequency dictionaries.

In Model Construction, individual document dictionaries are summarized into a token-frequency matrix, which stores the class-wise frequencies of unique word types in the vocabulary. This matrix is then sorted and filtered for a selection of feature words. After that, the matrix is used to compute the individual *Word Posterior Probabilities*. Logarithm space is used to store the word probabilities for testing due to precision issues.

In Model Testing, the test data is extracted. By applying Add-One smoothing, this model handles the existence of unexpected words in the test set. Classes of test documents are predicted using the independence assumption, yielding test results of prediction-actual classes for all the test subjects.

In Model Evaluation, the test results are summarized into a prediction-recall matrix to calculate prediction and recall values for all the classes. F_1 -Score is then derived using both Macro-Averaging and micro-Averaging methods.

Lastly, I investigated the performance of the model by constructing it multiple times. During this process, a fluctuating nature of word probabilities is discovered and analysed, according to which the stability of the model is proved.