|  |  |  |
| --- | --- | --- |
| A logo on a black background  Description automatically generated | | |
| **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](#_Toc161608566)

[2 Background 3](#_Toc161608567)

[2.1 Classification 3](#_Toc161608568)

[2.2 Classification Model 3](#_Toc161608569)

[3 Designs and Approaches 4](#_Toc161608570)

[3.1 Tools & Packages 4](#_Toc161608571)

[3.2 Model Preparation 4](#_Toc161608572)

[3.2.1 The Beginning Principles: Naïve Bayes Basics 4](#_Toc161608573)

[3.2.2 Json Parsing & Prior Probability Calculation 6](#_Toc161608574)

[3.2.3 Pre-Processing: Tokenization, Stemming and Lowercase 7](#_Toc161608575)

[3.2.4 Dictionary Construction 9](#_Toc161608576)

[3.2.5 Vocabulary Construction 10](#_Toc161608577)

[3.3 Model Construction 10](#_Toc161608578)

[3.3.1 Posterior Probability of Word Type 10](#_Toc161608579)

[3.3.2 Token Frequency Matrix Construction, Sorting and Feature Selection 11](#_Toc161608580)

[3.3.3 Token Probability Matrix Construction 12](#_Toc161608581)

[3.3.4 Model Storage Strategies and Adaption of Logarithm Space 13](#_Toc161608582)

[3.4 Model Testing 14](#_Toc161608583)

[3.4.1 Decapsulate Testing Data 14](#_Toc161608584)

[3.4.2 Perform Classification 14](#_Toc161608585)

[3.5 Model Evaluation 15](#_Toc161608586)

[3.5.1 Precision and Recall Matrix 15](#_Toc161608587)

[3.5.2 Precision and Recall Calculation 16](#_Toc161608588)

[3.5.3 -Score Calculation using Macro & micro-Averaging 17](#_Toc161608589)

[4 Results 18](#_Toc161608590)

[4.1 Output Files and Explanations 18](#_Toc161608591)

[4.1.1 word\_count.txt 18](#_Toc161608592)

[4.1.2 word\_dict.txt 19](#_Toc161608593)

[4.1.3 word\_probability.txt 20](#_Toc161608594)

[4.1.4 classification\_result.txt 21](#_Toc161608595)

[4.1.5 Additional: f\_scores.txt 22](#_Toc161608596)

[4.2 Word Probability Fluctuation Phenomenon 22](#_Toc161608597)

[4.2.1 Description of Fluctuation 22](#_Toc161608598)

[4.2.2 Analysis of Phenomenon 23](#_Toc161608599)

[5 Conclusion 24](#_Toc161608600)

# 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 -Score. A higher -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 *.json* 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 , whereare words of the document. |
| 2. A finite set of class , where are classes stored in the set. |
| 3. A finite set of documents , where are documents stored in the document set. |
| 4. A mapping relation , i.e. a training data set , where |
| **Compute:** |
| Target Class: |
|  |
|  |
|  |
|  |
|  |
|  |

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

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 for each class and for each word type *x* in each class . The below algorithm flowchart demonstrates the model preparation process (later shown in 3.2) and the model construction process (later shown in 3.3).

|  |
| --- |
| **Figure 1:** Algorithm flowchart for the implementation of Naïve Bayes classification model |
| 图示  描述已自动生成 |

### 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 ← zeros(1,5); |
| 3 | **for** instance **in** trainClassSentenceStr **do** |
| 4 | curClass ← 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 , the number of documents.

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 ← zeros(1,5); |
| 3 | Let sum ← sum(classFreqArr); |
| 4 | **for** index = 0…classFreqArr.length()-1 **do** |
| 5 | classPriorProbs[index] ← 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. |  |
| “&lt;” and “>” | &lt;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** | r'\"|\.\.+|\(|\)|\s--+\s|(?<=[A-Za-z])/|&[a-z]+;|>' | |
| **Patterns** | | **Interpretation** |
| \" | | Any double-quotation mark won’t be a part of a valid word. |
| \.\.+ | | Any continuous periods always separates two disjoint words. |
| \(|\) | | Any brackets won’t be a part of a valid word. |
| \s--+\s | | Any continuous hyphens always separates two disjoint words. |
| (?<=[A-Za-z])/ | | A slash only separates two disjoint words but doesn’t separate two numbers. |
| &[a-z]+;|> | | 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** | r'(?<![A-Z])([.,?!"]\s+)' |
| **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 ?<![A-Z] 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 |
| 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 |

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

|  |  |
| --- | --- |
| **Algorithm 3:** String tokenization | |
| 1 | **Procedure** tokenize(trainClassSentenceStr); |
| 2 | Let trainClassSentenceArr ← zeros(size(trainClassSentenceStr)); |
| 3 | **for** instance **in** trainClassSentenceStr **do** |
| 4 | curClass ← instance.class; |
| 5 | curDocStr ← instance.doc; |
| 6 | curDocStr ← curDocStr.**regexpReplace**(‘\"|\.\.+|\(|\)|\s--+\s|(?<=[A-Za-z])/|&[a-z]+;|>’, ‘\s’); |
| 7 | curDocStr ← curDocStr.**regexpReplace**(‘(?<![A-Z])([.,?!"]\s+)’, ‘\s’); |
| 8 | curDocArr ← curDocStr.**tokenizeWithDelimiter**(‘\s’); |
| 9 | curInstance ← (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 ← ∅; |
| 3 | **for** instance **in** trainClassDictionaries **do** |
| 4 | **for** word, frequency **in** instance.dictionary **do** |
| 5 | vocab ← vocab ∪ {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 for all word types in all classes.

The posterior probability of a word , i.e., is a conditional probability stating that, within the scope of all documents with label class , the portion of word 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.

To calculate , 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 .

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 ← new Matrix(row=vocab, col=classes); |
| 3 | **for** instance **in** trainClassDictionaries **do** |
| 4 | Let curClass ← instance.class; |
| 5 | Let curWordDict ← instance.wordDictionary; |
| 6 | **for** word, frequency **in** curWordDict **do** |
| 7 | TFM [word, curClass] += frequency; |
| 8 | TFM ← sortByRowSum(TFM, order=descending); |
| 9 | **Output** TFMatrix. |

#### 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 |
| 10,001 | anixt | | 0 | 0 | 0 | 3 | 0 | Disregarded |
| … | … | | … | … | … | … | … |
| 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 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 ← 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 represents the conditional word probability regarding to a specific class. Therefore, to calculate 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 TPMatrix ← zeros(featureMatrix.size()); |
| 3 | Let vocabSize ← featureMatrix.height(); |
| 4 | **for** wordType, frequencies **in** featureMatrix.rows() **do** |
| 5 | **for** index = 0…frequencies.length() - 1 **do** |
| 6 | TPMatrix[wordType, index]  ← (featureMatrix[wordType, index] + 1) / (classWordFreq[index] + vocabSize) |
| 7 | **Output** TPMatrix. |

### 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:

for .

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

for , for .

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 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*,

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:

This number is even less than the machine epsilon of 64-bit floating point storage, whose value is . 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:

## 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:

Target class would be:

The below algorithm demonstrates the classification process.

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

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

Correspondingly,

This model finishes in 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

-Score is going to be the evaluation metric for this Naïve Bayes model. To implement the -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 ← zeros(5,5); |
| 3 | **for** instance **in** testResults **do** |
| 4 | curCompare ← instance.compare; |
| 5 | prediction, actual ← curCompare[0], curCompare[1]; |
| 6 | PRMat[prediction, actual] += 1; |
| 7 | **Output** PRMat. |

### 3.5.2 Precision and Recall Calculation

For each class ,

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 |

Horizontal Line:

Vertical Line:

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 -Score. For each individual class, the -Score is:

### 3.5.3 -Score Calculation using Macro & micro-Averaging

#### 3.5.3.1 Macro-Averaging method

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

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

|  |  |
| --- | --- |
| **Algorithm 10:** Macro-averaging -Score | |
| 1 | **Procedure** CalcMacroFScore(precisions, recalls); |
| 2 | Let FScores ← zeros(1,5); |
| 3 | **for** index = 0…precisions.length() **do** |
| 4 | curPrec ← precisions[index]; |
| 5 | curRec ← recalls[index]; |
| 6 | FScores[index] ← ; |
| 7 | FScoreMacro ← 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 -Score.

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

|  |  |
| --- | --- |
| **Algorithm 11:** Micro-averaging -Score | |
| 1 | **Procedure** CalcMicroFScore(precisions, recalls); |
| 2 | avgPrec ← Average(precisions); |
| 3 | avgRec ← Average(recalls); |
| 4 | FScoreMicro ← ; |
| 5 | **Output** FScoreMicro. |

The successful computation of the two -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 |
|  | **First Line:** Document Frequencies for all classes.  **Following Lines:** Word Frequency for each word type for each class. |
| … |
|  |

### 4.1.2 word\_dict.txt

|  |  |
| --- | --- |
| **Table 16** | |
| Result | Explanation |
|  | **First Line:** Word Frequencies for all classes.  **Following Lines:** Word Frequency for each word type for each class, where the words are top 10,000 most frequent. |
| … |
|  |

### 4.1.3 word\_probability.txt

|  |
| --- |
| **Table 17.1** |
| Result |
|  |
| … |
|  |

|  |
| --- |
| **Table 17.2** |
| Explanation |
| **First Line:** Class Probability for each class.  **Following Lines:** Word Probability for each word in each class. |

### 4.1.4 classification\_result.txt

|  |  |
| --- | --- |
| **Table 18** | |
| Result | Explanation |
|  | **All Lines:** Test ID and the corresponding prediction classes. |
| … |
|  |

### 4.1.5 Additional: f\_scores.txt

|  |
| --- |
| **Table 19.1** |
| Result |
|  |

|  |
| --- |
| **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.  **-Scores:** Individual -Scores of all classes.  **Macro-Average -Score:** -Score gained using Avg Prec and Avg Rec.  **Micro-Average -Score:** -Score gained by averaging Individual -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 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 |
|  |  |

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.

|  |
| --- |
| **Figure 2:** Randomization and word frequencies fluctuation |
| 图表, 树状图  描述已自动生成 |

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 -Scores unchanged. Below shows a demonstration on the -Scores before and after a duplicated model construction. To differentiate the two, model construction time is printed in the console.

|  |  |
| --- | --- |
| **Table 21:** Comparison of -scores before and after a duplicated model construction | |
| Before | After |
| 图形用户界面, 文本  描述已自动生成 | 文本  描述已自动生成 |

### 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 -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. -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.