|  |  |  |
| --- | --- | --- |
| 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

[Introduction 3](#_Toc160150013)

[Background 3](#_Toc160150014)

[Approaches, Designs and Methods 3](#_Toc160150015)

[3.1 Python Packages 3](#_Toc160150016)

[3.2 Preparations 3](#_Toc160150017)

[3.2.1 Retrieve Information from train.json 3](#_Toc160150018)

[3.2.2 String Tokenization 3](#_Toc160150019)

[3.2.3 Instance-wise Word-Frequency Dictionary Construction 5](#_Toc160150020)

[3.2.4 Vocabulary Construction 5](#_Toc160150021)

[3.2.5 Data Summarization using DataFrame 5](#_Toc160150022)

[3.2.6 Feature Selection 5](#_Toc160150023)

[3.3 Naïve-Bayes Probability Calculation & Model Implementation 5](#_Toc160150024)

[3.4 Evaluation using F-Score 5](#_Toc160150025)

[Results 5](#_Toc160150026)

[Conclusion 5](#_Toc160150027)

# Introduction

# Background

# Approaches, Designs and Methods

## 3.1 Python 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 | Substitute varieties of delimiters or useless symbols in natural language writings by a single space. |
| nltk | Construct the tokenizer for tokenizing a string into an array of words with a given delimiter. |
| pandas | Construct a DataFrame object to contain the word frequency matrix, indexed by vocabulary. |
| numpy | Some basic math tasks, like initializing an array filled with 0. |
| collections | Turn a word array (which may contain duplicates) into a dictionary of word-frequency pairs. |

## 3.2 Data Retrieval and Sentence Pre-Processing

In this part, this report demonstrates the ideas and methods for information retrieval and data processing for counting frequencies of all the individual words in different classes. Since we need to extract json data in both the training and testing process, the following algorithm is encapsulated into a single function extract\_data\_from\_json(), stored in the \_\_funcs\_\_.py file.

### 3.2.1 Parse json Data & Class-wise Document Frequency

By applying the internal json package in Python, it is easy to extract the training data into a two-dimensional list called train\_class\_and\_sentence\_list. Each element in this list is a document with its labelled class, represented as:

[‘training/2118’, ‘acq’, ‘SHAD SEES …’]

The first attribute of each instance is the training id. The second attribute is the class of the document. The third attribute is the document string, which would be further tokenized into word arrays, accompanied by its class.

Having this array, we can count the document frequency for each class in the training set by iterating each instances of the list, accumulating the occurrence of each class whenever one of them is met. The algorithm below shows the process of class-wise document frequency calculation:

|  |  |
| --- | --- |
| **Algorithm 1:** Calculate class-wise document frequency | |
| 1 | **Procedure** CalcDocFreq(trainData, classMap); |
| 2 | Let docFreqs ← zeros(1,5); |
| 3 | **for** instance **in** trainData **do** |
| 4 | curClass ← trainData.class; |
| 5 | curClassIndex ← classMap[curClass]; |
|  | docFreqs[curClassIndex] += 1; |
| 6 | **Output** docFreqs. |

The output array of document frequencies is called train\_class\_doc\_freqs for better convenience.

### 3.2.2 Pre-Processing: String Tokenization and Stemming

After retrieving a bag of string sentences, we should pre-process them to make sure it can be used to run the Naïve Bayes algorithm. This process involves tokenizing each sentence into a list of tokens, stemming the tokens to reduce polymorphism of words, constructing a sentence-wise dictionary for better word counting, as well as constructing a vocabulary for the set of sentences, i.e. training data.

#### 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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table2:** 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.

|  |  |  |
| --- | --- | --- |
| **Table3:** 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.

|  |  |
| --- | --- |
| **Table4:** Regular expression for the second rule | |
| **Rule** | r'(?<A-Z])([.,?!"]\s+)' |
| **Interpretation** | This rule basically 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 very easy for a tokenizer to parse a sentence using the delimiter of “\s”. However, there are still space to improve, like there are still some words that contain some invalid words 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 |

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, the data should be of the following form.

|  |
| --- |
| **Code2:** Data structure after processing |
| [  [‘acq’,[‘word1’,’word2’,’word3’,’word4’]],  [‘grain’,[‘word1’,’word2’,’word3’]],  [‘grain’,[‘word1’,’word2’,’word3’,’word4’,’word5’]],  [‘money-fx’,[‘word1’,’word2’]],  [‘acq’,[‘word1’,’word2’,’word3’,’word4’]],  …  ] |

#### 3.2.2.2 Word Stemming

Moreover, to make it further prepared for dictionary and vocabulary construction, we need to stem all instances of words to eliminate some redundant information, like tense of a verb. This is done using the PorterStemmer package. After stemming, we receive a more mature version of training data.

The overall data structure of the training data retrieved from the json file is an array of arrays, as shown in 3.2.2.1. An instance or element of this array is a tuple. The first element of the tuple is the class of the sentence. The second element of the tuple is the tokenized sentence, which is in the form of an array of tokens.

Here, the data is well pre-processed and is ready for further processing. The overall data here is called train\_class\_sentences for convenience.

#### 3.2.2.3 Instance-wise Word-Frequency Dictionary Construction

This part only regards the scope of an instance of train\_class\_sentences, i.e. a class-sentence tuple. The goal of this part is to merge multiple identical words, i.e. tokens in a sentence into a single key-value pair, where the key is the potentially duplicated token, and the value is the frequency of occurrence of the token. The stemming before allows this process to merge multiple words with the same base word but were originally in different forms. After merging, the original token list were transformed into a dictionary, called

An instance is a string-list pair, where the list represents the tokenized version of the original sentence, and the string is the class of the sentence. The Counter() function extracted from package collections turns the list into a dictionary of word-frequency pairs.

|  |  |
| --- | --- |
| **Table 5:** Construct word-frequency dictionary | |
| Instance before construction | [‘acq’, [‘the’,’**cake’**,’is’,’a’,’lying’,**’cake’**]] |
| Instance after construction | [‘acq’, {‘the’:1,’**cake’:2**,’is’:1,’a’:1,’lying’:1}] |

The reason why converting original list into a dictionary is necessary is that the order of the words doesn’t matter much for the present tasks, yet merging word counts saves space and time for frequency statistics.

After the construction of the dictionary, we now have two versions of training data. As mentioned, the first version is the training data with plain sentences, called train\_class\_sentences. The second one is the training data set with sentences in the form of word dictionaries, called train\_class\_dictionaries.

#### 3.2.2.4 Vocabulary Construction

Unlike dictionary construction, the vocabulary construction regards the scope of all instances of the data, to be more specific, train\_class\_dictionaries. 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.

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. By traversing all the instances, the vocabulary only inserts words it had never seen before into its storage. This process yields a better performance when the instances are processed to merge identical words into one, saving the time for traversing. The built-in set() function is used to perform the construction. Below shows the algorithm for vocabulary construction using train\_class\_dictionaries.

|  |  |
| --- | --- |
| **Algorithm 2:** 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. |

### 

Here comes to an end of the extract\_data\_from\_json() function. This function returns: The list of document frequencies of each class, a list of class-sentence pairs in the form of both tokenized sentence train\_class\_sentences and dictionary train\_class\_dictionaries. Below shows the formats of returning values:

|  |  |  |
| --- | --- | --- |
| **Table6:** Formats of outputs of extract\_data\_from\_json() | | |
| train\_class\_doc\_freqs | train\_class\_sentences | train\_class\_dictionaries |
| [  num1,  num2,  num3,  num4,  num5  ] | [  [‘class’,[‘t1’,’t2’,’t2’]],  [‘class’,[‘t1’,’t2’,’t1’]],  [‘class’,[‘t1’,’t1’,’t2’]],  [‘class’,[‘t2’,’t1’,’t2’]],  …  ] | [  [‘class’,{‘t1’:1,’t2’:2}],  [‘class’,{‘t1’:2,’t2’:1}],  [‘class’,{‘t1’:2,’t2’:1}],  [‘class’,{‘t1’:1,’t2’:2}],  …  ] |

## 3.3 Data Summarization and Feature Word Selection

### 3.3.1 Data Summarization using DataFrame

It is observed that the size of vocabulary is around 30,000. From this gigantic vocabulary, there are words with very few occurrences compared to others. Therefore, we need to further select the most frequent 10,000 words to construct the model to save space and performance. To realize this, it is better to sort the vocabulary using the word frequencies.

The row indexes of the matrix are the words in the vocabulary, while the column indexes of the matrix are five different classes. By traversing every instances of the train data, the program first get the class of the instance. It then adds the frequency value of every unique word in the dictionary into the matrix. Notice that the matrix cell is located by the word-class pair. For each instance, the insertion is performed on only one column of the matrix since there’s only one class for an instance.

Below shows one possibility of the 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 7:** 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 universal occurrences of each word token in the vocabulary. Due to lack of functionalities of DataFrame, it is required to calculate the sum during matrix construction, and then slice out the last column storing the sum of each row after sorting. The algorithm below shows the full construction process.

|  |  |
| --- | --- |
| **Algorithm 2.1:** 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. |

### 3.2.6 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 along the index of the vocabulary, 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 featureMatrix.

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.

## 3.4 Naïve-Bayes Probability Calculation & Model Implementation

This corresponds to the Requirement 3 and 4.

### 3.4.1 Naïve Bayes Algorithm Analysis

An essential goal of the Naïve Bayes algorithm is to predict a class for a sentence based on the current datastore. To be more specific, the below table demonstrates the inputs and outputs of Naïve Bayes Classification Algorithm.

|  |
| --- |
| **Table 8:** Inputs and outputs of Naïve Bayes Classification Algorithm |
| **Given:** |
| 1. A test document , whereare words of the document. |
| 2. A set of class , where are classes stored in the set. |
| 3. A set of documents , where are classes stored in the set. |
| 4. A mapping relation , i.e. a training data , 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,

### 3.4.2 Class Prior Probability

The prior probability of a specific class *c* is the portion of documents that is labelled as class *c* in the training set. It is equal to the number of documents in labelled in class *c* divide by , the number of all documents.

Remark that the given training set is in the data structure of an array of tuples, where each tuple stores both the sentence itself and the class that this sentence is labelled to, i.e., train\_class\_doc\_sentences & train\_class\_doc\_freqs. To calculate , we should calculate both the numerator and denominator.

Fortunately, we have already calculated the numerator for each class, that is the document frequencies train\_class\_doc\_freqs retrieved above. To calculate the denominator is also simple enough, just by calculating the size of the document set, that is, , towards which one of many ways is by summing up the items in train\_class\_doc\_. At this point, we’re able to successfully calculate the prior probability of all the classes in , finishing the first subtask.

### 3.4.3 Word Posterior Probability

The posterior probability of a word is . It 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. This formula is smoothed in case there are words in test set that doesn’t exist in the training set.

Again, we need to find out the numerator and denominator to calculate this probability.

The calculation of this denominator is like the previous one, where the only difference is that it takes regard to not the occurrence of document that belongs to this class, but the number of words contained in the sentence in that instance. This is done by iterating through the Token-Frequency Matrix generated before.

The calculation of the numerator is just an altered version of calculating the denominator, only filtering the word that is equal to .

The numerator refers to the word frequencies of all five classes of a single token, which is represented by a row of the featureMatrix.

The denominator refers to the word frequencies of all five classes. Given featureMatrix, we can retrieve the target by the algorithm:

|  |  |
| --- | --- |
| **Algorithm 3:** Calculate word frequencies for each class | |
| 1 | **Procedure** CalcClassWordFreqs(featureMatrix); |
| 2 | Let trainClassWordFreqs ← zeros(1,5); |
| 3 | **for** token, frequencies **in** featureMatrix **do** |
| 4 | **for** index **in** 0… frequencies.length() **do** |
| 5 | trainClassWordFreqs[index] += curWordFreqs[index]; |
| 6 | **Output** trainClassWordFreqs. |

Having both the numerator and the denominator, we can calculate the posterior probability of all words in all classes, i.e., . Add-One smoothing is used and the data structure featureProbMatrix is used to store all the posterior probabilities of each word in each class.

|  |  |
| --- | --- |
| **Algorithm 4:** Calculate word prob for each class | |
| 1 | **Procedure** CalcWordProbs(featureMatrix, trainClassWordFreqs); |
| 2 | Let featureProbMatrix ← zeros(1,5); |
| 3 | **for** token, frequencies **in** featureMatrix **do** |
| 4 | **for** classIndex **in** 0…frequencies.length() **do** |
| 5 | featureProbMatrix[token, classIndex]  ← (featureMatrix[token, classIndex] + 1) / (trainClassWordFreqs[classIndex] + |vocab|); |
| 6 | **Output** featureProbMatrix. |

By now, we have constructed the probabilistic model for given classes, documents, and test set.

## 3.4 Evaluation using F-Score

# 

# Designs and Approaches

## 3.1 Tools & Packages

## 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:** Inputs and outputs of Naïve Bayes Classification Algorithm |
| **Given:** |
| 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).

![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 within python. Raw training data called *Training Class-Sentence String Tuple List* is retrieved after parsing. It is a list whose elements are tuples of three elements. Within a specific element, 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, in the format of a string. Below shows the data format of *Training Class-Sentence String Tuple List*:

|  |
| --- |
| **Table:** 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:** 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:** 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 on the documents itself. Remark that in each instance of the raw training data, the third attribute is the document. The format of a document in the original raw data is simply a plain string, where tokens are separated by natural delimiters.

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:** 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:** 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:** Regular expression for the second rule | |
| **Rule** | r'(?<![A-Z])([.,?!"]\s+)' |
| **Interpretation** | This rule basically 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 very easy for a tokenizer to parse a sentence using the delimiter of “\s”. However, there are still space to improve, like there are still some words that contain some invalid words 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:** 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:** 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:** 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:** 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:** 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:** 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:** 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:** 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:** 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:** 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

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 , and .

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*.

#### 3.3.4.1 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*,

## 3.4 Model Testing

## 3.5 Model Evaluation

# Results

# Conclusion