FUNCTIONAL PROGRAMMING
CS-IS-2010-1
FINAL PROJECT REPORT

Topic: Haskell Scraper and Code-Text Separation

Name: Saptarishi Dhanuka

ID: 1020211525

Supervisor: Partha Pratim Das

Date: 13 May 2024

Ashoka University

Contents

1 Acknowledgements

I would like to thank Professor Das for taking this Independent Study Module and giving us the incentive to work on a software project in an unfamilar yet elegant language. I would also like to thank my fellow students as well as the teaching volunteers Gautam and Adwaiya for creating a wholesome learning environment.

2 Introduction

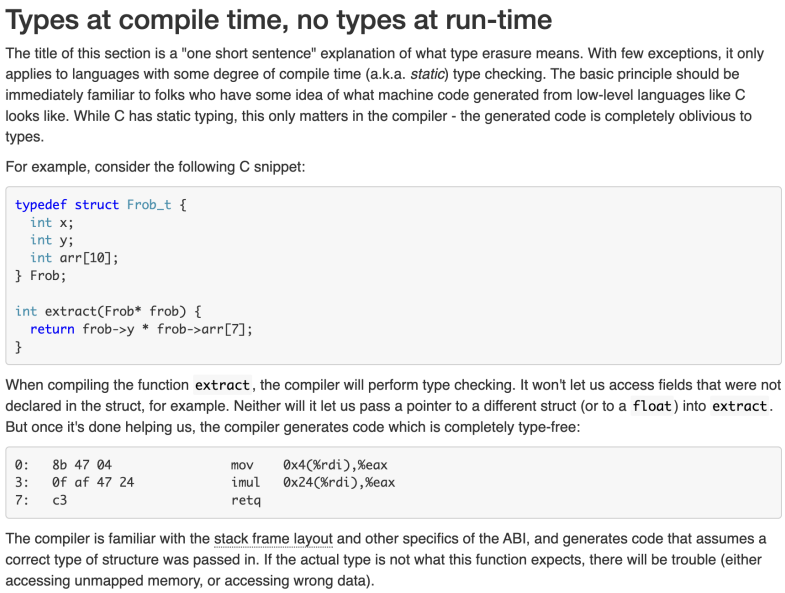
The large amounts of data in the form of interspersed code and natural language make it useful to extract knowledge from them in order to benefit various areas of software development. Examples like natural language comments accompanying code, stackoverflow questions and answers with code blocks, and developer emails containing discussions about code with reference to it are all important forms of such data.

INSERT STUFF HERE!!!!!!!

For instance, extracting code and natural language separately from developer emails at a company can give key insights about how the company codebase has evolved and can help future developers access the code of emails directly. Separation of code and natural language is also helpful as training data for natural language and code completion models which can used the separated text and code to give better results. It creates a structured view of the code and textual data which can then further be used to create an organized view of the code and associated natural language.

3 Background and Motivation

Consider the following image of an educational web page containing interspersed code and natural language:



Types at compile time, no types at run-time

The title of this section is a "one short sentence" explanation of what type erasure means. With few exceptions, it only applies to languages with some degree of compile time (a.k.a. *static*) type checking. The basic principle should be immediately familiar to folks who have some idea of what machine code generated from low-level languages like C looks like. While C has static typing, this only matters in the compiler - the generated code is completely oblivious to types.

For example, consider the following C snippet:

```
typedef struct Frob_t {
  int x;
  int y;
  int arr[10];
} Frob;

int extract(Frob* frob) {
  return frob->y * frob->arr[7];
}
```

When compiling the function `extract`, the compiler will perform type checking. It won't let us access fields that were not declared in the struct, for example. Neither will it let us pass a pointer to a different struct (or to a `float`) into `extract`. But once it's done helping us, the compiler generates code which is completely type-free:

```
0:   8b 47 04                mov    0x4(%rdi),%eax
3:   0f af 47 24             imul   0x24(%rdi),%eax
7:   c3                      retq
```

The compiler is familiar with the stack frame layout and other specifics of the ABI, and generates code that assumes a correct type of structure was passed in. If the actual type is not what this function expects, there will be trouble (either accessing unmapped memory, or accessing wrong data).

Figure 3.1: Example of Code and Language Together

Here it becomes useful to extract just the code or just the natural language separately if one just wants to run the code on their own machine or just wants the natural language descriptions for their notes or to get an overview of the idea being conveyed. Using HTML based separation of the text and code snippets, or trying to use regular expressions severely restricts the generalisability and accuracy of the system in extracting natural language and code separately.

INSERT STUFF HERE!!!!!!!

Towards this purpose, we will create a Naive Bayes classifier for classifying a particular line as natural language or code and creating two separate sections for all the natural language and all the code

4 Literature Survey

Researchers have used various methods to identify and distinguish code and natural language text, some of which I will discuss below.

Bacchelli et. al [1] attempted to extract source code from developer emails by first identifying emails that contained source code and then identifying code blocks by using regular expressions. The approach was very lightweight and programming language specific, with the use of end-of-line characters like semi-colons in order to find source code lines.
Bacchelli et. al [2] also used Naive Bayes with bigrams, in combination with island parsing for classifying pieces of development emails as various categories like code, natural language, stack trace, patch or junk. It reached an accuracy ranging between 89 and 94 % but, the source code island parser was specific to Java, which the authors acknowledged and stated that the approach could be easily generalized to other language as well.
Chatterjee et. al [3] focused on extracting the code segments along with their natural language descriptions from research articles using various detailed linguistic and structural heuristics to identify important features to identifying sentences that would be relevant to code segments
Cerulo et. al [4] used an approach based on Hidden Markov Models (HMMs) for extracting "information islands" of code from natural language by recognising whether the sequence of observed strings would change between the source code and natural language states of the HMM. While this approach did not require parsing or complicated regular expressions, it only performed classification of tokens into code or text, and not further knowledge extraction in the form of an abstract syntax tree.

The Naive Bayes approach for text classification also has a long history and it is extensively used even today despite it's simplicity and strong assumptions.

ADD STUFF HERE!!!!!!!!!!!!

5 Problem Statement and Objectives

Assigned Project Statement

Develop a scraper using Haskell to extract text and code snippets separately.

1. **Input**: Scrape the text and code snippets from the given text source
2. **Output**: A Word document containing the text and `.txt` file containing the code.
3. **Method**: Write the algorithm to scrape (you can use the `tagsoup` library) and all the input-output facilities using Haskell. Do not use any other language.

5.1 Requirements/Objectives

1. The user shall be able to give any text source as input.
2. The scraper shall get all the code snippets of the source and write it into a Plaintext file.
3. The scraper shall get all non-code text of the source and write it into a Word Document.

5.2 Specifications

1. The user will be able to enter a text source as input, whose code and non-code parts they wish to be separated.
2. The scraper will parse the contents of the text and separate the code snippets from the rest of the text.
3. The scraper will output a `.docx` file containing the textual content.
4. The scraper will output a `.txt` file containing the code snippets.

5.3 Analysis

1. There are many ways of solving the problem both by syntactic and semantic approaches. Some semantic approaches are as follows:
   a) Lexical and semantic analysis with the use of regular expressions
   b) Using a large language model to differentiate the code and the rest of the text
   c) Using computer vision to attempt to read text like a human and identify text from the code
   d) Use the frequency of occurrence of different words in some sample data and use it to predict whether sections of unseen samples of text are natural language or source code.

2. Extracting the text and code snippets from a text source boils down to a classification task where we consider each new line as a line to be classified as natural language or code, and grouping them all into two separate sections depending on the class assigned to them.

3. Hence we consider that the text source will consist of different newlines which need to be classified as language or code, without going into further granular details as to whether a particular word or phrase is code or text.

6 Scope and Methodology

6.1 Scope

Following from the analysis, the system will classify text sources on a line level of granularity, hence it assumes that there will be some newline separation of different lines and the text source will not be completely unstructured. This is still a fairly broad scope since most text sources that have natural language and code interspersed in them have a newline-structure that make it easy to split them on a line-by-line basis for classification.

6.2 Methodology

The broad methodology is to train a Naive Bayes classifier on some custom training data consisting of natural language and source code, and then using the calculated probabilities to classify each line in the given text source as being code or natural language, and then separating them based on that. A brief overview of Naive Bayes is given below.

6.2.1 Naive Bayes

Our main goal with Naive Bayes is to find, for each line in the text source, the best class (code or natural language) for it. That is, for each line l made of some t_k terms and class c, we want to find
$\hat{c} = \text{argmax}_c \, \mathbb{P}(c|l) = \text{argmax}_c \, \mathbb{P}(c|t_1, t_2, ...t_k)$. Instead of just considering the terms occuring in the line, we will consider all words in the vocabulary from now and see if they are occuring in the line or not.

First we consider the vocabulary of all the words in the training data. Now for each line in the text source, we have a binary feature vector whose elements are set to 1 if a word in the vocabulary occurs in that line, otherwise being set to 0. That is, if there are n words in the vocabulary, then the feature vector l_i corresponding to the i-th line will have $l_{ij} = 1$ if the j-th word from vocabulary occurs in the i-th line, otherwise being 0.

Now we want to calculate $\mathbb{P}(l_i|y)$ where y is a particular class like code or natural language. We make the strong Naive Bayes assumption that all the l_{ij}'s are conditionally independent given the class y. This is clearly not true in practice, but even with this assumption we get good performance. Moreover, we consider each line as an unordered Bag of Words where word position doesn't matter and only it's frequency matters, which

is another strong assumption that doesn't impact performance that much.

Hence $\mathbb{P}(l_{i1}...l_{in}|y) = \prod_{j=1}^{n} \mathbb{P}(l_{ij}|y)$.

Considering a particular line l and using Bayes Rule we get:

$\hat{c} = \text{argmax}_c\, \mathbb{P}(c|l) = \text{argmax}_c\, \mathbb{P}(l|c) \times \mathbb{P}(c) = \text{argmax}_c\, \mathbb{P}(c) \times \prod_{j=1}^{n} \mathbb{P}(l_j|c)$.

Usually, the probability numbers computed are very small, so we work in the logspace by using the natural logarithm to avoid issues with numerical underflow.

$\therefore \hat{c} = \text{argmax}_c \log \mathbb{P}(c|l) = \text{argmax}_c \log \mathbb{P}(l|c) \times \mathbb{P}(c) = \text{argmax}_c \log \mathbb{P}(c) \times \prod_{j=1}^{n} \mathbb{P}(l_j|c)$

$= \text{argmax}_c(\log \mathbb{P}(c) + \sum_{j=1}^{n} \mathbb{P}(l_j|c))$

How to calculate these terms?

We calculate $\mathbb{P}(c)$ by just calculating what percentage of the training data belongs to that particular class. For each $\mathbb{P}(l_j|c)$, we calculate the probability of the j-th word in the vocabulary being in the class c by counting the number of times that word occurs in the class c and dividing by the total number of words in the class c. Using a form of Laplace smoothing, we add small decimal numbers α and β in the numerator and denominator to avoid 0 probability in case of completely unseen words for the unseen data.

Hence $\mathbb{P}(l_j|c) = \dfrac{count(l_j, c) + \alpha}{\beta + \sum_{j=1}^{n} count(l_j, c)}$

7 Design and Architecture

7.1 High-Level Architecture

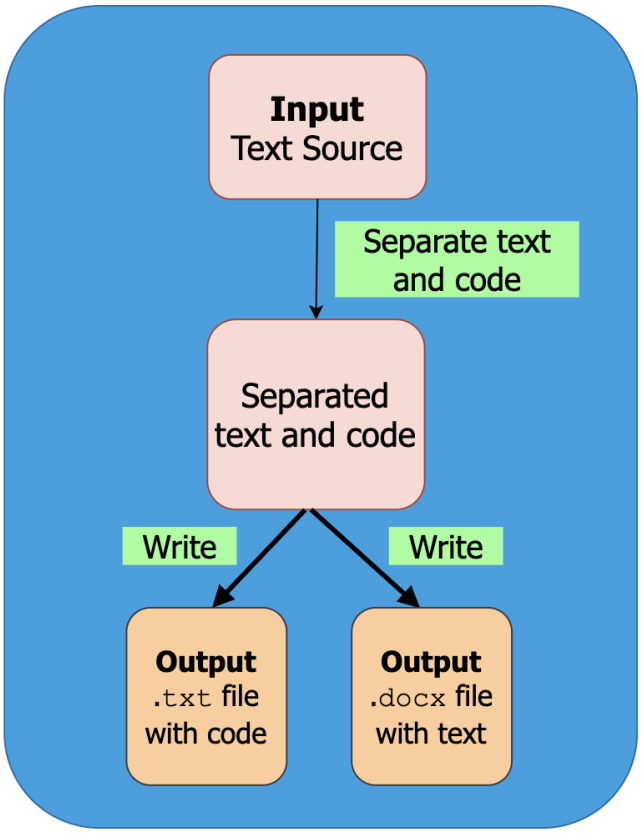


Figure 7.1: High-Level Architecture of Code-Text Separation Pipeline

The **high-level architecture** consists of the following:

1. Get the contents of the text source
2. Separate the code snippets from the text.
3. Writing the code snippets into a `.txt` file.
4. Write the non-code textual content into a `.docx` file.

7.2 High-Level Design

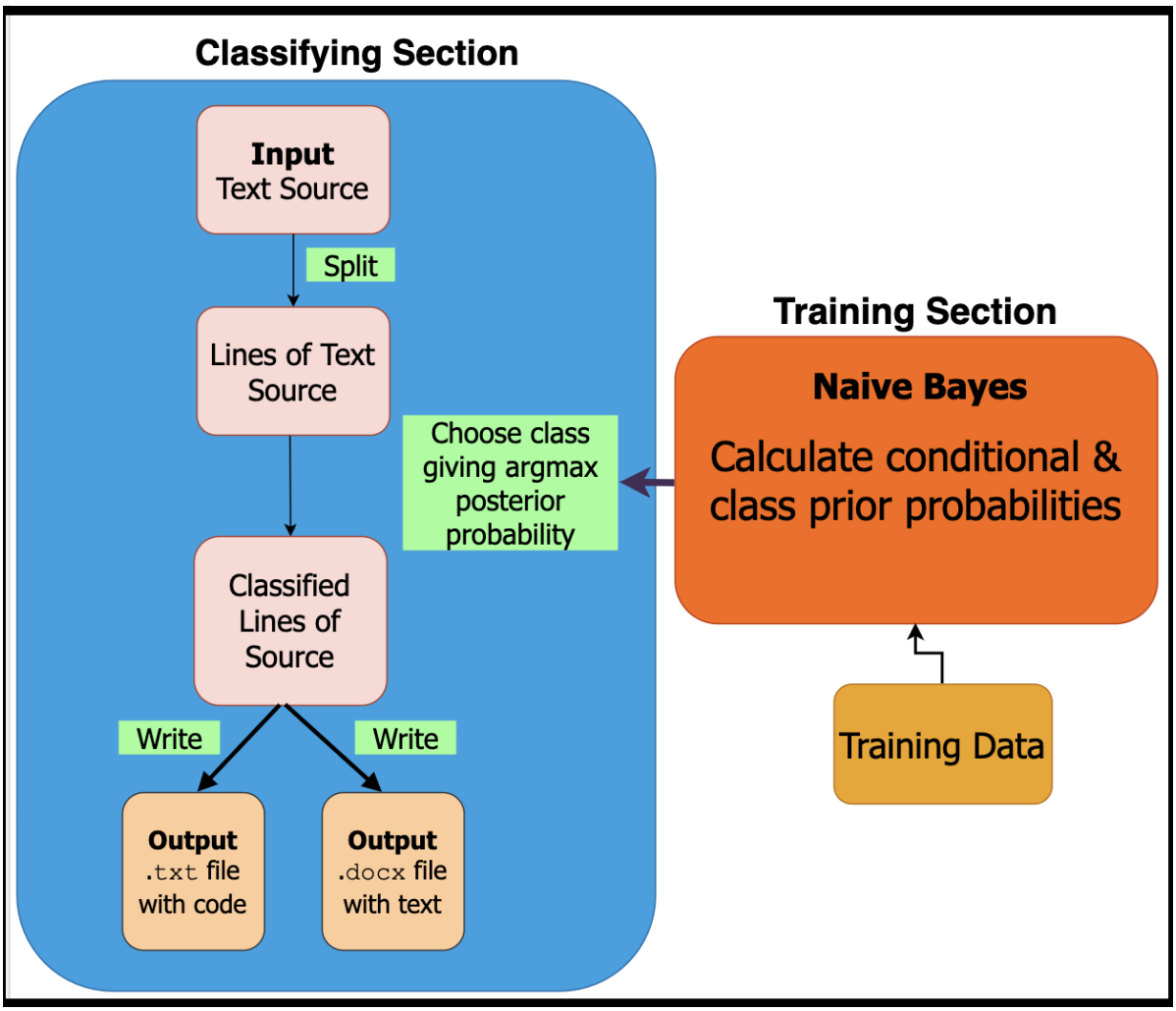


Figure 7.2: High-Level Design of the Classifying Pipeline and Training Section

A more detailed description of the **high-level design** which implements the architecture broadly consists of two sections:

1. **Training Section :**
   a) Use natural language and source code training data already pre-classified in order to calculate the conditional probabilities and the class priors which make up the trained Naive Bayes Model.
2. **Separation Pipeline Section :**
   a) Split the given text source into lines which will be considered as separate "documents" by the trained classificaton model.

b) Classify each different line as source code or natural language text depending on which class has the higher posterior probability given that line

c) Write the lines classified as source code into a `.txt` file

d) Write the lines classified as natural language text into a `.docx` file

The lower level implementation details are mentioned in the implementations section.

8 Implementation Details

8.1 File Structure

```
scraper
|-- app
|   | -- Main.hs (main driver code)
|-- test
|   |-- Spec.hs (tests)
|-- src
|   |-- Lib.hs (implementation of functions)
|-- input
|   | -- lang_train.txt (training data)
|   | -- code_train.txt (training data)
|-- output_files
|   | -- NB_code_class.txt (code snippets)
|   | -- NB_lang_class.docx (textual content)
```

8.2 Features and limitations

The system correctly identifies the code and text portions of the web page and writes them into the `.txt` and `.docx` files as per the requirements and specifications.

Limitations

1. As mentioned, the system works on the granularity of a line and not a deeper token level granularity. Hence it can only attempt to classify whether a whole line is natural language or source code, without being able classify individual words or phrases within a line as being of a different class.

2. Moreover, it cannot be used effectively in the case that the text source has data that is not separated by newlines, since each line is a document that needs to be classified here. However, these cases are relatively rare since most instances where code and natural language are found interspersed, there is some degree of organisation in terms of newline-separation.

3. Since the system is relatively simplistic and unoptimized, along with the fact that it works on the Naive Bayes assumption of independence between features, it may

not work well compared to other heavyweight methods in terms of accuracy.

8.3 Core Implementation Details

Some important code and data features that showcase the core implementation details of the project are shown.

8.3.1 Main.hs

Training and Main Classification Pipeline

In-line with the design, architecture and choice of tools as mentioned above, we first obtain the contents of the text source. Keeping in mind the assigned project statement that had a web page as text source, the user will supply input by giving a URL of the web page for which they wish the language and code to be separated. In principle, any text source can be used, which would just remove fetching the web page contents portion of the system.
We first just get all the textual HTML content from the web page and split the content based on newline separators.
Everytime the system is called, training on the data will be done since we wish to keep the Naive Bayes classifier dynamic in terms of the words in accounts for. Since training for Naive Bayes is cheap, this can be done without much additional cost.

In the below code, we train the Naive Bayes Classifier, then predict the classes of every line of the text source and finally separate the text and the code into `.txt` and `.docx` files.

```
    -- text_source is newline separated list of lines of the
    ↪  text_source
    let lang_train = "input/lang_train.txt"
    let code_train = "input/code_train.txt"
    mydata <- Lib.readTraining lang_train code_train
    let lang_data = fst mydata
    let code_data = snd mydata
    let trainedModel = Lib.trainNaiveBayes lang_data code_data

    let final_classes = Lib.classifyNaiveBayes text_source trainedModel
    let mapping = zip text_source final_classes
    let code_class = [x | x <- mapping, snd x == 0]
    let lang_class = [x | x <- mapping, snd x == 1]
    writeFile "output_files/NB_code_class.txt" (unlines (map fst
    ↪  code_class))
    writeToDocx "output_files/NB_lang_class.docx" (unlines (map fst
    ↪  lang_class))
```

Evaluating the Classifier

We use various test sets to evaluate the performance of the classifier on various metrics, which will be further discussed in the Results section.

```
    -- text_source is newline separated list of lines of the
    ↪  text_source
    lang_test <- readFile "input/lang_test.txt"
    src_test <- readFile "input/code_test.txt"
    let test_data = (lines src_test) ++ (lines lang_test)
    print test_data
    let final_probs = Lib.classifyNaiveBayes test_data trainedModel
    let mapping = zip test_data final_probs
    let src_test_len = length (lines src_test)
    let lang_test_len = length (lines lang_test)
    let yTest = replicate src_test_len 0 ++ replicate lang_test_len 1
    let test_accuracy_mapping = zip yTest final_probs
    -- precision_code, recall_code, precision_lang, recall_lang
    let evals = Lib.evaluateNaiveBayes test_accuracy_mapping
```

8.3.2 Lib.hs

trainNaiveBayes

This takes the natural language and source code training data and returns the prior probability of a word being a source code word, the conditional probabilities and the vocabulary based on the training data. The description of the helper functions is given later on below.

```

-- NLA refers to the hmatrix library
trainNaiveBayes :: [String] -> [String] -> (( Double, ([Double] ,
↪  [Double]) ), Vocabulary )
trainNaiveBayes natural_data source_data =
    let source_words = concat (getWords source_data)
        natural_words = concat (getWords natural_data)
        unique_src_words = getUniqueWords source_words
        unique_natural_words = getUniqueWords natural_words
        vocab = unique_src_words ++ unique_natural_words

        -- source_data is an array of strings, where each string
        ↪  represents a new line in the file. So here each line here
        ↪  is a document here.
        xTrain_src = myVectorizer vocab (source_data)
        xTrain_lang = myVectorizer vocab (natural_data)
```

```
14
15          sourceCodeMatrix = xTrain_src
16          naturalLanguageMatrix = xTrain_lang
17
18          -- NLA.Matrix Double -> [Int]
19          sum_src_cols = sumCols sourceCodeMatrix
20          sum_lang_cols = sumCols naturalLanguageMatrix
21
22
23          src_len = length source_data
24          natural_len = length natural_data
25
26          xgivenY_src = calcXGivenY src_len sum_src_cols
27          xgivenY_lang = calcXGivenY natural_len sum_lang_cols
28
29          prob_src_prior = (int2Double src_len) / (int2Double src_len +
            ↪  fromIntegral natural_len)
30
31      in ((prob_src_prior, (xgivenY_src,xgivenY_lang)), vocab )
32
33
```

classifyNaiveBayes

It takes the lines of the text source that we have to classify, along with the output of the trained Naive Bayes Model. Due to the conditional probabilities being very small numbers, we work in the logspace to prevent issues due to numerical underflow, as is the case with most Naive Bayes implementations.

TODO!!!!!!!!!!!!!!!!!!!!!!!!!

```
1
2
3   -- NLA refers to the hmatrix library
4   classifyNaiveBayes :: [String] -> (( Double, ([Double] , [Double]) ),
    ↪  Vocabulary ) -> [Int]
5   classifyNaiveBayes test_data trainedModel =
6       let vocab = snd trainedModel
7           xTest = (NLA.toLists (myVectorizer vocab (test_data)))
8           test_len = length xTest
9           y = NLA.fromLists [(replicate (test_len) (int2Double 0))]
10
11          -- [Double]
12          xgivenY_src = fst (snd (fst trainedModel))
```

```
        xgivenY_lang = snd (snd (fst trainedModel))
        prob_src_prior = fst (fst trainedModel)

        log_src = map log xgivenY_src
        log_lang = map log xgivenY_lang

        log_matrix = NLA.tr (NLA.fromLists [log_src, log_lang])

        -- [Double,Double]
        -- print (map typeOf (head (DM.toLists log_matrix)))

        -- mult
        prob1 = (NLA.fromLists xTest) NLA.<> (log_matrix)

        -- [Double,Double]
        -- print (map typeOf (head (DM.toLists prob1)))
        logp = log prob_src_prior
        log_not_p = log (1 - prob_src_prior)

        prob1_trans = NLA.toLists (NLA.tr prob1)

        prob2 = map (\x -> x + logp) (head prob1_trans)
        prob3 = map (\x -> x + log_not_p) (head (tail prob1_trans))

        combined = [prob2, prob3]
        combined_mat = NLA.tr (NLA.fromLists combined)

        final_probs = map (\x -> if (head x) > (head (tail x)) then 0
        ↪  else 1) (NLA.toLists combined_mat)
    in final_probs
```

Important Helpers

1. `myVectorizer` : gives a matrix where each row corresponds to a document i.e. line and each column is a word in the vocabulary. So an entry (i, j) represents the word count of word j from the vocab in document i. If supplied empty lists then we return a dummy value.

2. `matrixRow` : It corresponds to one row in the above matrix. For each word in the vocabulary it gets the count of that word in the current document.

3. `wordCounts` : For a document, creates a mapping between each word in the document and the count of that word in the document.

4. `sumCols` : Finds the sum of every column in the matrix i.e. finds the total occurences of each word across all documents.

5. `calcXGivenY` : For each word in the vocabulary, we sum the number of times it has appeared in source code documents. We divide by the total number of source code documents to get the probability of each word being in source code. We do a similar thing for natural language. We add small decimal numbers as a form of Laplace smoothing to avoid 0 probability in case of completely unseen words for the unseen data.

```
1   myVectorizer :: Vocabulary -> [Document] -> NLA.Matrix Double
2   myVectorizer vocab docs
3       | vocab == [] = NLA.fromLists [[int2Double 0]] --
        ↪  sinceFromlists doesn't accept empty lists
4       | docs == [] = NLA.fromLists [[int2Double 0]]
5       | otherwise = NLA.fromLists [matrixRow vocab doc | doc <- docs]
6
7   matrixRow :: Vocabulary -> Document -> [Double]
8   matrixRow vocab doc = [fromMaybe (int2Double 0) (lookup vocab_word
    ↪  mywordcounts) | vocab_word <- vocab]
9   where mywordcounts = wordCounts doc
10
11  wordCounts :: Document -> [(String, Double)]
12  wordCounts doc = Data.Map.toList $ fromListWith (+) [(oneword,
    ↪  int2Double 1) | oneword <- words doc]
13
14  sumCols :: NLA.Matrix Double -> [Double]
15  sumCols matrix = map sum (NLA.toLists (NLA.tr matrix))
16
17  calcXGivenY :: Int -> [Double] -> [Double]
18  calcXGivenY mylen my_cols_sum =  map (\x -> x + 0.001 / int2Double
    ↪  (mylen) + 0.9 ) my_cols_sum
19
```

Writers to files

`writeToDocx` takes the text that has been classified as natural language and writes it into a `.docx` file after some formatting in order to massage the text into a Pandoc-friendly format due to the absence of functionality to read a plainText document or string.

```
1  writeToDocx :: String -> String -> IO ()
2  writeToDocx filepath lang_class  = do
3      pandoc_lang <- runIO $ readHtml def ( TextConv.convertText ("<p>"
       ↪  ++ lang_class ++ "</p>" :: String ) :: T.Text )
```

```
4
5       case pandoc_lang of
6           Right x -> do
7               y <- runIO $ writeDocx def x
8               case y of
9                   Right direct_pan -> do
10                      LBS.writeFile filepath direct_pan
11
12                  Left err -> Prelude.putStrLn $ "Error with pandoc
                     ↪  writeDocx: " ++ show err
13
14          Left err -> Prelude.putStrLn $ "Error parsing pandoc for
             ↪  natural language " ++ show err
15
16      putStrLn "Completed writing to docx"
```

8.4 Training Data

Due to paucity of appropriate training data for this project, I curated my own small training dataset comprising of code and natural language that is typically seen accompanying the code.
I made sure to include a wide variety of programming languages in the source_code dataset to ensure wider generalisability. The statistics of the training data are given as follows, along with the sources where I took them from. The code was cleaned to remove comments and other non-code text.

Programming Language	**Lines of Code**	Source
C	1171	Andrej Karpathy's Github
Python	461	Sklearn's Github
Java	201	Jenkins Github

Table 8.1: Training Data Statistics

8.5 Challenges and Mitigations

Initially, I started the project by separating the code and text based on their HTML tags for the midterm project evaluation, but after feedback from Professor I decided to make the project much more generalized to any given text source instead of just HTML content due to the fragile nature of a system based on the latter.
I decided to implement the Naive Bayes algorithm from scratch due to it's ease of interpretability, simplicity, lightweightedness and surprisingly strong results. A brief outline of some challenges and mitigations are as follows:

1. Before deciding on implementing Naive Bayes from scratch, I thought about using various methods like Hidden Markov Models and Support Vector Machines to create an ensemble learning method to classify text and code. However, library support in Haskell for these was not of good quality, hence I decided to implement Naive Bayes on my own.

2. In the implementation of Naive Bayes, another hurdle I faced was the inefficiency of the default `matrix` library in doing calculations with extremely large and sparse matrices, which was restricting me to training and testing sets of very small sizes, albeit with considerably good performance. So I looked for more efficient libraries and found the `hmatrix` library, which sped up computations by a large extent. Earlier a training and classification that took 2.5 minutes overall, now happened in 2.5 seconds.

3. The paucity of appropriate training data was also an issue. The few datasets that were relevant to the project, such as those containing source code along with some form of natural language (usually in the form of code comments) were all restricted

to one language and were also immense for a classifier implemented from scratch without optimisations. Hence, I curated my own modestly size source code and natural language training data based on manual searching online

4. The Haskell language was itself relatively challenging to get used to throughout the project due to it's difference from the imperative languages I am used to. But after learning lambda calculus and spending time with the language, it became easier to use and I could appreciate the elegance of functional programming more, even though I am still not as accustommed to this paradigm as I would like to be.

9 Tooling and Testing

9.1 Tools and Languages

9.1.1 Languages

Only Haskell was used for the project as mentioned in the problem statement

9.1.2 Tools

1. The Glasgow Haskell Compiler (GHC) is used for compilation.
2. Stack is used as the build tool. This manages installing project dependencies, building and running the project and testing the project.
3. **Notable Libraries**:
   a) **hmatrix :** The fastest numerical linear algebra library that I could find in order to speed up the entire process, according to one benchmark.
   b) **hUnit :** The basic Haskell unit testing library

9.2 Testing

9.2.1 Test Suite Outline

1. **Unit Testing :** The following functions were unit tested in the `Spec.hs` file
   a) `myVectorizer :`
   b) `matrixRow :`
   c) `wordCounts :`
   d) `getWords :`
   e) `getHTML :`
   f) `getUniqueWords`
   g) `getHTML`
      i. Test if the function returns the correct HTML for various URLs by using sample inputs and outputs.
      ii. Test if the function gracefully handles invalid URLs with error handling.

iii. Test if the function handles network errors and HTTPS errors with error handling.

h) `writeToDocx` : While File I/O cannot be unit-tested in the traditional sense, we can still manually test the file outputs.

i. Test if the function correctly writes the content of the input Tag String to the `.docx` file using sample input output

2. **Performance Testing :**

a) Test the time taken for the full system to execute from start to finish

b) Test the resource consumption of the system

3. **Functional Testing :**
Test whether the program meets the requirements by using sample input URLs and sample output files

9.2.2 Test Report

1. **Unit Testing :**

10 Plan for Completion

- Implement error handling gracefully wherever possible so that no unhandled errors can occur.
- Implement the test suite
- Attempt to make the scraper more generalized so that it can work with other websites and other HTML structures.
- Document the code well
- Explore methods of separating the code from the non-code textual content that are independent of HTML structure. Such methods were mentioned in the analysis and implementation of these methods will make the scraper more generalizable.

11 Results and Discussions

The classifier was run on various natural language and source code files and the results were collected. Metrics like precison and recall with respect to code and language, along with the F-measure were calculated. We found that even with the highlight simplistic design and the strong Naive Bayes assumptions, the classifier could perform surprisingly well in classifying lines as code or natural language.

12 Conclusions

13 Extensions and Future Work

Bibliography

[1] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting source code from e-mails," in Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, June 2010, pp. 24–33.

[2] A. Bacchelli, T. Dal Sasso, M. D'Ambros and M. Lanza, "Content classification of development emails," 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2012, pp. 375-385, doi: 10.1109/ICSE.2012.6227177

[3] Chatterjee, Preetha & Gause, Benjamin & Hedinger, Hunter & Pollock, Lori. (2017). Extracting Code Segments and Their Descriptions from Research Articles. 10.1109/MSR.2017.10.

[4] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora, "A hidden markov model to detect coded information islands in free text," in Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on, Sept 2013, pp. 157–166.