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

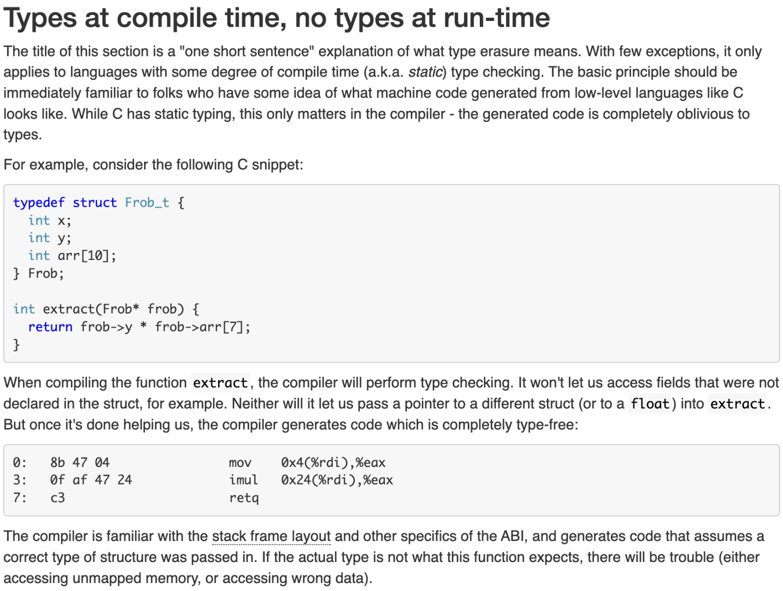
The large amounts of data in the form of interspersed code and natural language make it useful to extract knowledge from them in order to benefit various areas of software development. Examples like natural language comments accompanying code, stackoverflow questions and answers with code blocks, and 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

Reseachers 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

7 Design and Architecture

7.1 High-Level Architecture

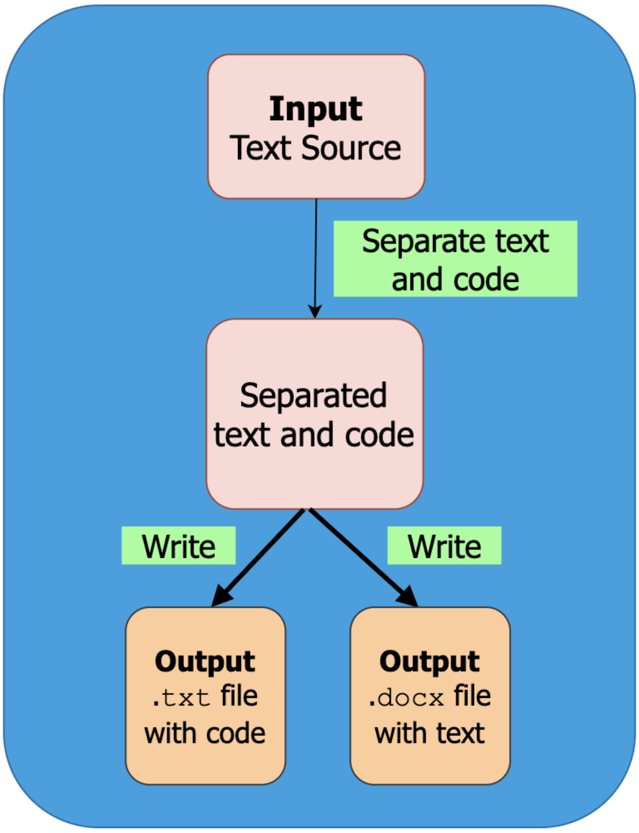


Figure 7.1: High-Level Architecture of Code-Text Separation Pipeline

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

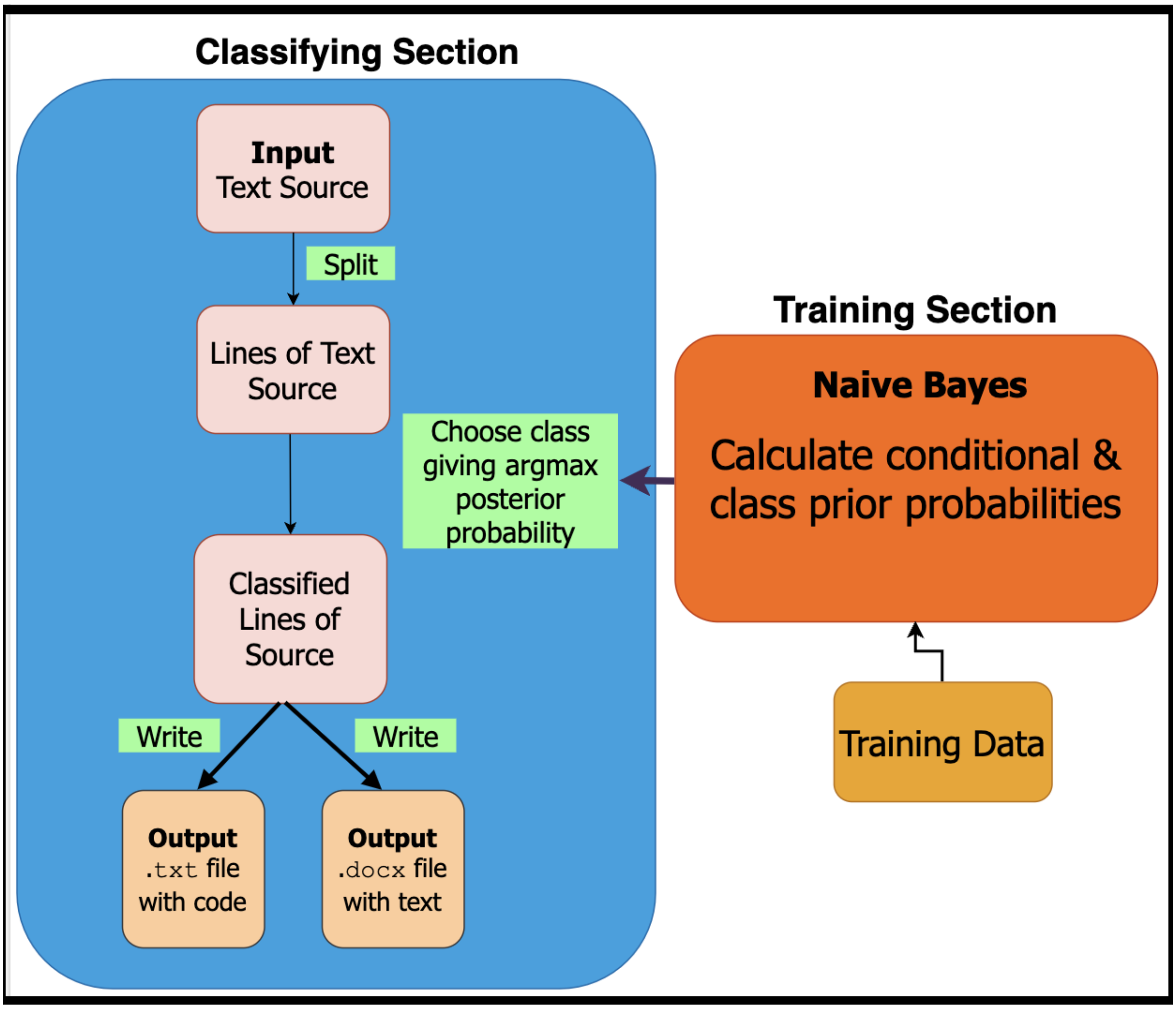


Figure 7.2: High-Level Design of the Classifying Pipeline and Training Section

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. **Separatation 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

8.2 Features and limitations

The prototype 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. The prototype has not been robustly tested, verified for correctness or structured to handle errors. However, this can be developed while making the final system since the prototype is already relatively modular.

2. Since the HTML structure of different web pages can vary, it is not necessary that this particular scraper will work for all web pages. It is designed specifically for the given page and may work for some other pages. But no generalisation can be made about the correctness of its text and code extraction for other pages.

3. Moreover, it will not necessarily work for web pages with malformed HTML or different structure.

4. It is not designed to be robust to design changes, which is in line with the rule stated on the `tagsoup` library's documentation example, since the website's HTML can change in unpredictable ways. If the site's HTML structure changes, for instance if the code snippets change from being enclosed in `<pre>` tags to `<code>` tags, then the scraper will not be able to separate out the code from the text and extract them accurately.

5. It clearly does not work with text sources that are not HTML formatted web pages.

8.3 Core Implementation Details

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

8.3.1 Main.hs

In-line with the design, architecture and choice of tools as mentioned above, the prototype obtains the HTML content of the url in line 2 and parses it into Tag Strings in line 3. Then we separate it into code tags and non-code textual tags in lines 4-6. Then we write the Tags to respective files in like 7-8.

The core structure of the `Main.hs` file is

```
1. let url = "https://eli.thegreenplace.net/2018/type-erasure-and-reification/"
2. response_html <- getHTML url
3. let parsed_tags = parseTheTags response_html
4. let separated_text_code = separateTextCode parsed_tags
5. let preTags             = fst separated_text_code
6. let nonPreTags          = snd separated_text_code
7. writeToTxt preTags
8. writeToDocx nonPreTags
```

8.3.2 Lib.hs

Imports and pragmas

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Network.HTTP.Client as Client
import qualified Network.HTTP.Client.TLS as ClientTLS

import qualified Text.HTML.TagSoup as Soup
import Text.Pandoc

import qualified Data.Text.Conversions as TextConv
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.ByteString.Lazy as LBS
import qualified Data.ByteString.Lazy.Char8 as LBSC

```

getHTML

getHTML takes the url as input and sets up a new TLS manager with newTlsManager, parses the url with parseRequest and then executes the request with httpLbs. Then it gets the HTML content from the response using responseBody.

```
getHTML :: String -> IO LBSC.ByteString
getHTML url = do
    mymanager <- ClientTLS.newTlsManager
    myrequest <- Client.parseRequest url
    response <- Client.httpLbs myrequest mymanager
    let response_html = Client.responseBody response
    return response_html

```

parseTheTags

parseTheTags takes the HTML ByteString and parses it into a list of Tag Strings using parseTags from Tagsoup.

```
parseTheTags :: LBSC.ByteString -> [Soup.Tag String]
parseTheTags response_html = (Soup.parseTags :: String -> [Soup.Tag
 ↪  String]) (LBSC.unpack response_html)
```

fillTrue

fillTrue is a helper function that fills in True for every element enclosed within a matching pair of True statements corresponding to an opening and closing <pre> tags

```
fillTrue :: [Bool] -> [Bool]
fillTrue [] = []
fillTrue [a] = [a]
fillTrue (False:False:xs) = False:fillTrue (False:xs)
fillTrue (False:True:xs) = False:fillTrue (True:xs)

-- fill True for elements after a <pre> tag starts
fillTrue (True:False:xs) = True:fillTrue (True:xs)

-- seeing 2 consecutive Trues denotes end of <pre> tag, so skip them
fillTrue (True:True:xs) = True:True:fillTrue (xs)
```

insertNewlines

`insertNewLines` is a helper function that inserts a TagText for a delimiter before every closing `<pre>` tag for formatting purposes

```
insertNewlines :: [Soup.Tag String] -> [Soup.Tag String]
insertNewlines [] = []
insertNewlines (x:xs) = if (Soup.isTagCloseName "pre" x)
    then Soup.TagText
    ↪  "\n\n=======================\n\n":x:insertNewlines (xs)
    else x:insertNewlines (xs)
```

separateTextCode

The idea of `separateTextCode` is as follows. Create a boolean list corresponding to every element of the list of Tag Strings. A boolean list element will be True if a Tag is a `<pre>` tag or within a `<pre>` tag. Then zip the boolean list with the original Tag list and get the Tags which have a correponding True value; this is the list of code tags. Now mark elements to be true if the corresponding tags are `<img>` tags and create an overall boolean list which has false if the tag is not code and not an image. Extract the tags which have a false value; this is the list of non-code textual tags.
It essentially solves the parenthesization problem with the `<pre>` tags.

```
separateTextCode :: [Soup.Tag String] -> ([Soup.Tag String], [Soup.Tag
↪  String])
separateTextCode parsed_tags =
    let bool_mapping_open = Prelude.map (Soup.isTagOpenName "pre")
    ↪  parsed_tags
        bool_mapping_close = Prelude.map (Soup.isTagCloseName "pre")
        ↪  parsed_tags

        -- do element-vise or of the two lists
        bool_mapping1 =  Prelude.zipWith (||) bool_mapping_open
        ↪  bool_mapping_close

        filled_true_pre = fillTrue bool_mapping1
        combined_pre = Prelude.zip filled_true_pre parsed_tags
        preTags = Prelude.map snd (Prelude.filter (fst) combined_pre)

        bool_mapping_img_open = Prelude.map (Soup.isTagOpenName "img")
        ↪  parsed_tags
        bool_mapping_img_close = Prelude.map (Soup.isTagCloseName
        ↪  "img") parsed_tags
        bool_mapping2 =  Prelude.zipWith (||) bool_mapping_img_open
        ↪  bool_mapping_img_close

        bool_mapping =  Prelude.zipWith (||) bool_mapping1
         ↪  bool_mapping2

        filled_true = fillTrue bool_mapping
        -- zip parsed_tags list with the boolean list and
         ↪  Prelude.filter elements of the first list based on the
         ↪  second list to create tuple list
        combined = Prelude.zip filled_true parsed_tags
        -- get tags of tuples from combined where the boolean is
         ↪  false
        nonPreTags = Prelude.map snd (Prelude.filter (\(a,_) -> not a)
         ↪  combined)

    in (preTags, nonPreTags)


```

Writers to files

writeToTxt takes the code Tags and renders them as an HTML formatted string after delimiting the snippets. Then the string is converted into the Text type, which is then fed into readHtml which converts it into an intermediate Pandoc format.
writePlain converts this pandoc into Text, which is finally written to the .txt file.

writeToDocx takes the text Tags and renders them as an HTML formatted string. Then the string is converted into the Text type, which is then fed into readHtml which converts it into an intermediate Pandoc format.
writeDocx converts this pandoc into a ByteString, which is finally written to the .txt file.

```
writeToTxt :: [Soup.Tag String] -> IO ()
writeToTxt preTags = do
    let htmlPre = Soup.renderTags (insertNewlines preTags)
    pandocPre <- runIO $ readHtml def ( TextConv.convertText (htmlPre
     ↪  :: String ) :: T.Text )

    case pandocPre of
        Right x -> do
            y <- runIO $ writePlain def x
            case y of
                Right direct_pan_pre -> do
                    TIO.writeFile "output_files/final_code.txt"
                     ↪  direct_pan_pre

                Left err -> Prelude.putStrLn $ "Error with pandoc
                 ↪  writePlain: " ++ show err

        Left err -> Prelude.putStrLn $ "Error parsing pandoc for pre
         ↪  tags: " ++ show err

    putStrLn "Completed writing to txt"


writeToDocx :: [Soup.Tag String] -> IO ()
writeToDocx nonPreTags = do
    let htmlNonPre = Soup.renderTags nonPreTags
    pandocNoPre <- runIO $ readHtml def ( TextConv.convertText
     ↪  (htmlNonPre :: String ) :: T.Text )

    case pandocNoPre of
        Right x -> do
            y <- runIO $ writeDocx def x
            case y of
                Right direct_pan -> do
                    LBS.writeFile "output_files/final_text.docx"
                     ↪  direct_pan

                Left err -> Prelude.putStrLn $ "Error with pandoc
                 ↪  writeDocx: " ++ show err

        Left err -> Prelude.putStrLn $ "Error parsing pandoc for non
         ↪  pre tags: " ++ show err

    putStrLn "Completed writing to docx"

```

8.4 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. **Libraries**:
   a)

9.2 Testing

9.2.1 Test Suite Outline

1. **Unit Testing :** The following functions can be tested
   a) `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.
   b) `parseTheTags` :
      i. Test if the function returns the correct list of Tag Strings for various HTML content.
      ii. Test if the function handles malformed HTML and erroneous inputs with error handling.
   c) `fillTrue` :

   i. Test if the function correctly fills in True values for all elements within any two matching pair of True values corresponding to `<pre>` tags.

   ii. Test if the function handles edge cases with empty lists or single element lists.

d) `insertNewLines` :

   i. Test if the function correctly adds the delimiter before every closing `<pre>` tag.

   ii. Test if the function handles cases where there are no `<pre>` tags, or the Tag String is malformed

e) `separateTextCode` :

   i. Test if the function correctly returns a tuple of two lists where one list has the Tag Strings corresponding to all the content enclosed within `<pre>` tags, and that the other list has all the other elements, excluding images.

   ii. Test edge cases of no `<pre>` tags, nested `<pre>` tags

f) `writeToTxt` : 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 with delimters to the `.txt` file using sample input output

g) `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

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.