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

Haskell Scraper & Code-Text Separator

Saptarishi Dhanuka

Advisor: Dr. Partha Pratim Das

Codebase link: https://github.com/SaptarishiD/haskell-scraper

Introduction

- Separate text and code from text source
- Useful for software development information & knowledge extraction
- Implement Naïve Bayes classifier from scratch
- Learnt Haskell starting from no experience
- Got idea of software specifications and development

Motivation

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).

A slightly adjusted example will clarify this:

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:

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

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

Some other use cases

- Developer Emails at company
- StackOverflow Q & A
- Training Language Models
- Any area where language and code are together

Literature Survey

- **Papers**
  - Regex and programming language specific
  - Naive Bayes with bigrams & island parsers (specific to Java)
  - Variety of heuristics & island parsing
  - Hidden Markov Models

Literature Survey

- **Pre-existing libraries**

  - HMM library : **`hmm`**
  - **`NaiveBayes`**

- Decided to implement lightweight algorithm from scratch instead of heavyweight ML methods

Problem Statement & Requirements

Assigned Problem 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.

Requirements

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

Specifications

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

Analysis

- Various ways to solve the problem
- Boils down to classification of every line as text or code
- Reasonably restrict ourselves to text sources with newlines
- Don’t consider completely unstructured data
- Don’t go more granular than a line
- Consider text as natural language

Scope & Methodology

Scope

- Fairly broad range of inputs that are structured with newlines
- Classify each line as code or text without classifying *within* each line
- Consider text as natural English language, not as some other non-code or non-English textual content

Methodology

- **Overview**
  - Build and train Naïve Bayes classifier on custom data
  - Use calculated probabilities to classify each line in the text source
  - Combine all code and natural language separately

Methodology

- **Naïve Bayes Classification**

Goal**:** For each line, find the best class

$$\hat{c} = \mathrm{argmax}_c\, \mathbb{P}(c|l) = \mathrm{argmax}_c\, \mathbb{P}(c|t_1, t_2, \ldots t_k)$$

Vocabulary: All words in training data

Classes: Code and Text

Instead of just the terms, consider all words with indicator

- For each line i in text source, have binary feature vector for presence of vocabulary word j

$$l_{ij} \begin{cases} 1, & \text{if } w_j \in l_{ij} \\ 0, & \text{otherwise} \end{cases}$$

- Bag of Words approach
- Make Naïve Bayes assumption of independence of words

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

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

- Work in logspace due to numerical precision and underflow

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

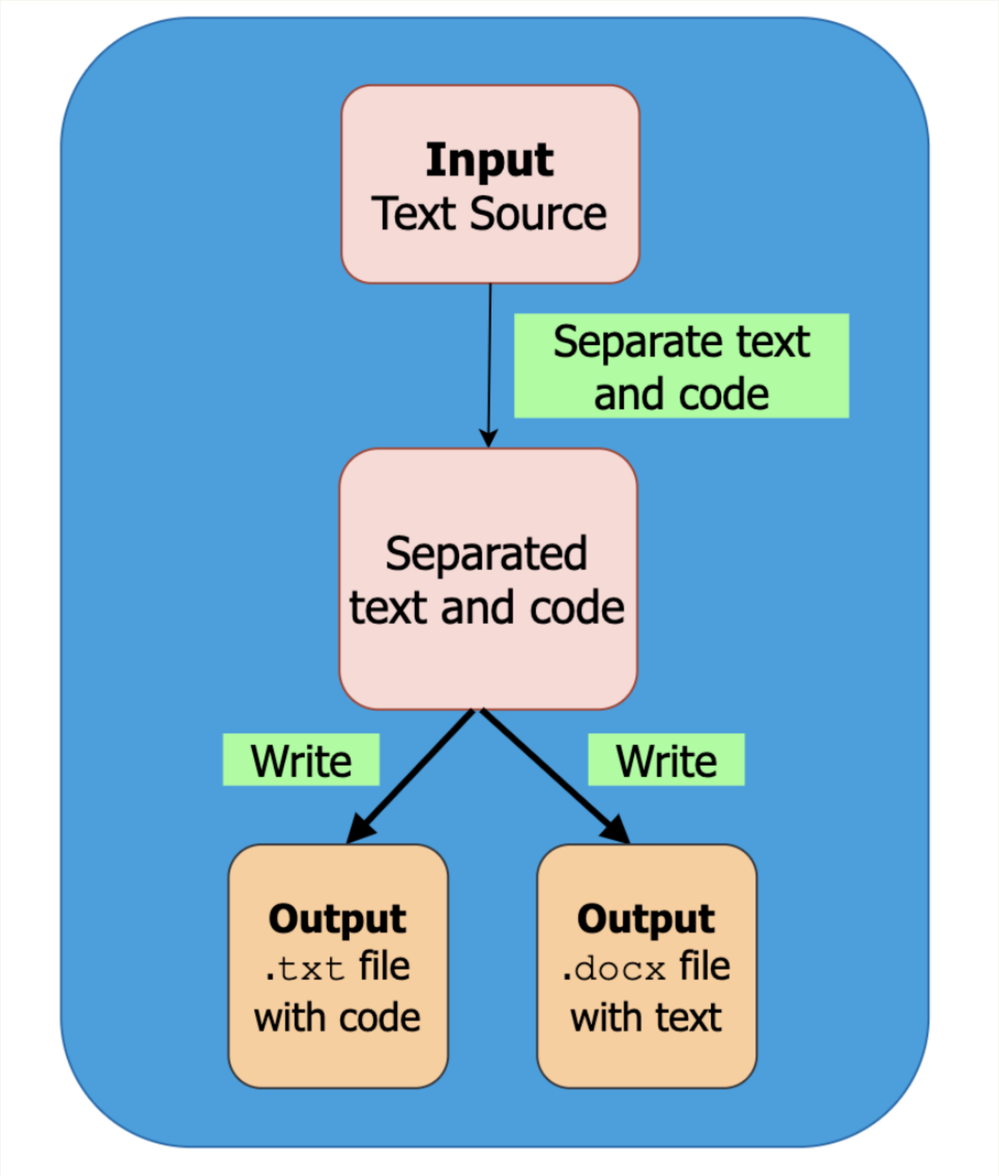
- Calculate probabilities
- Avoid 0 probabilities with Laplace Smoothing
- Count occurrences of *j-th* word in line l in class c and count all words in class c

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

- Prior probability of each class is simply it's percentage of the training data

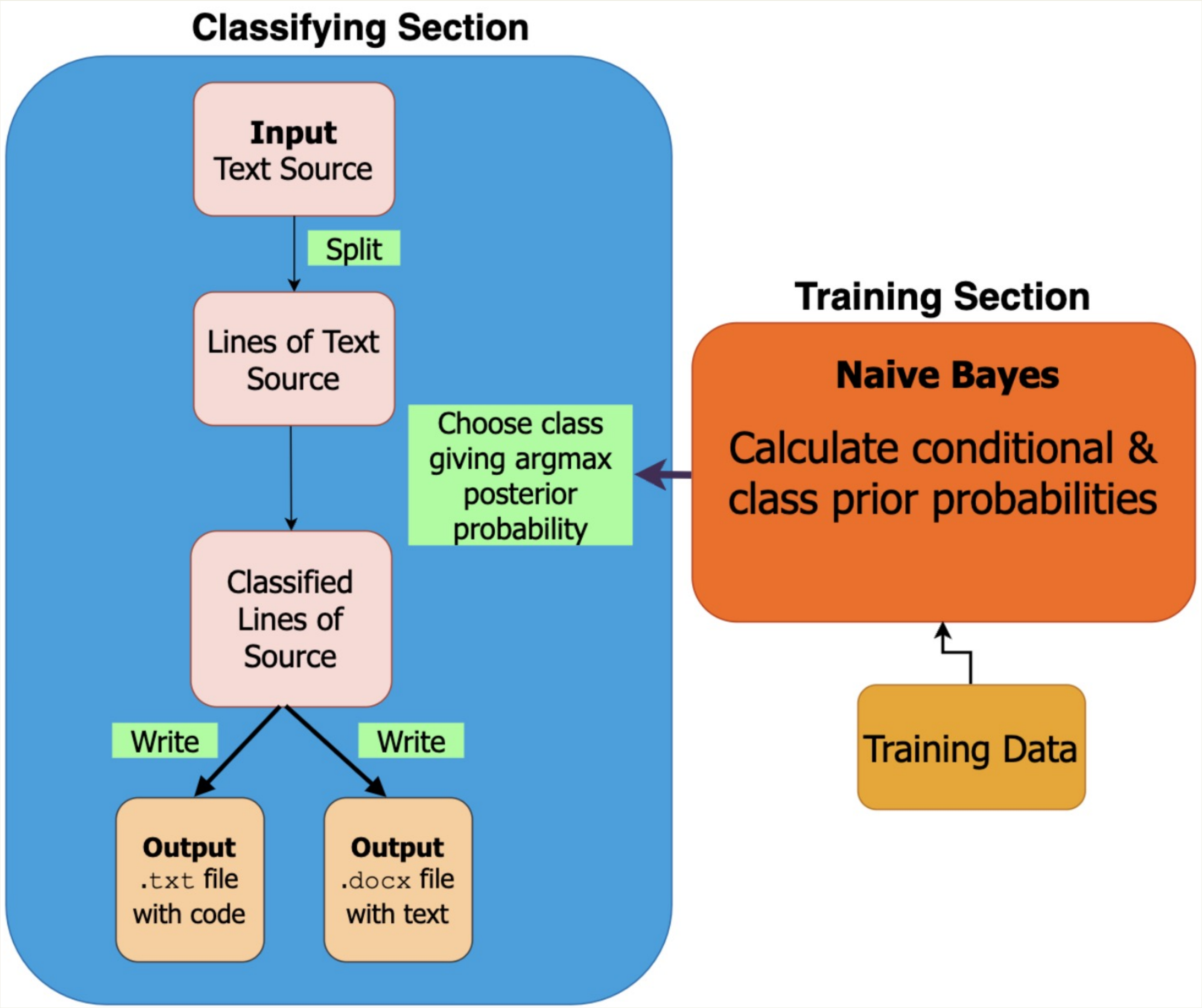
Architecture

Code-Text Separation Pipeline



Design

Classifying Pipeline and Training Section



Work Done

Implementation Details

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)
|-- cases
|    | -- code_test{1-7}.txt
|    | -- lang_test{1-7}.txt
|-- output_files
|    | -- NB_code_class.txt (code snippets)
|    | -- NB_lang_class.docx (textual content)
```

Main.hs

Classification and training pipeline

```
read_text_source <- readFile "input/sample.txt"
let text_source = lines read_text_source -- newlines
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))
```

Lib.hs : trainNaiveBayes

```haskell
-- NLA refers to the hmatrix library
trainNaiveBayes :: [String] -> [String] -> (( Double, ([Double] ,
↪  [Double]) ), Vocabulary )
-- the strange output type is due to packaging various things together
trainNaiveBayes natural_data source_data =
    let source_words = concat (getWords source_data) -- [String]
        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
        xTrain_src = myVectorizer vocab (source_data)
        xTrain_lang = myVectorizer vocab (natural_data)
        sourceCodeMatrix = xTrain_src
        naturalLanguageMatrix = xTrain_lang
        -- NLA.Matrix Double -> [Int]
        sum_src_cols = sumCols sourceCodeMatrix
        sum_lang_cols = sumCols naturalLanguageMatrix
        src_len = length source_data
        natural_len = length natural_data
        xgivenY_src = calcXGivenY src_len sum_src_cols
        xgivenY_lang = calcXGivenY natural_len sum_lang_cols
        prob_src_prior = (int2Double src_len) / (int2Double src_len +
        ↪  fromIntegral natural_len)
    in ((prob_src_prior, (xgivenY_src,xgivenY_lang)), vocab )
```

Lib.hs :
classifyNaiveBayes

```haskell
classifyNaiveBayes :: [String] -> (( Double, ([Double] , [Double]) ),
↪  Vocabulary ) -> [Int]
classifyNaiveBayes test_data trainedModel =
    let vocab = snd trainedModel
        xTest = (NLA.toLists (myVectorizer vocab (test_data)))
        test_len = length xTest
        y = NLA.fromLists [(replicate (test_len) (int2Double 0))]
        -- [Double]
        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])
        -- (head (DM.toLists log_matrix)) :: [Double,Double]
        -- matrix mult
        prob1 = (NLA.fromLists xTest) NLA.<> (log_matrix)
        -- (head (DM.toLists prob1))) :: [Double,Double]
        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
```

Lib.hs : Important Helpers

```
myVectorizer :: Vocabulary -> [Document] -> NLA.Matrix Double
myVectorizer vocab docs
    | vocab == [] = NLA.fromLists [[int2Double 0]] --
    ↪  sinceFromlists doesn't accept empty lists
    | docs == [] = NLA.fromLists [[int2Double 0]]
    | otherwise = NLA.fromLists [matrixRow vocab doc | doc <- docs]

matrixRow :: Vocabulary -> Document -> [Double]
matrixRow vocab doc = [fromMaybe (int2Double 0) (lookup vocab_word
↪  mywordcounts) | vocab_word <- vocab]
where mywordcounts = wordCounts doc

wordCounts :: Document -> [(String, Double)]
wordCounts doc = Data.Map.toList $ fromListWith (+) [(oneword,
↪  int2Double 1) | oneword <- words doc]

sumCols :: NLA.Matrix Double -> [Double]
sumCols matrix = map sum (NLA.toLists (NLA.tr matrix))

calcXGivenY :: Int -> [Double] -> [Double]
calcXGivenY mylen my_cols_sum =  map (\x -> x + 0.001 / int2Double
↪  (mylen) + 0.9 ) my_cols_sum
```

Training Data

Language	LoC	Source
C	1171	Andrej Karpathy's Github
Python	461	Sklearn's Github
Java	201	Jenkins Github

Description	Words & Lines	Source
CS50 Lec1	780 and 37	CS50 Lec1
CS50 Lec6	229 and 11	CS50 Lec6
Python 4 Everybody Text	4910 and 589	Python 4 Everybody Text

Challenges and Mitigations

- After midterm eval, redid project to change from HTML to general text
- Tried implementing HMM but decided on Naïve Bayes
- Poor library support : Started from scratch
- Inefficient `matrix` library : Found efficient `hmatrix`, reducing time from 2.5 min to 2.5 sec in one case

Challenges and Mitigations

- Paucity of appropriately sized training data : created custom small dataset
- Changing libraries and refactoring code

Tooling

- Haskell
- Stack build tool
- Notable Libraries
  - hmatrix
  - hUnit
  - pandoc

Testing

- Unit Testing
  - **`myVectorizer`**
  - **`matrixRow`**
  - **`wordCounts`**
  - **`getWords`**
  - **`getUniqueWords`**

Testing

- End-to-end testing
  - Tested by the evaluation tests
- Performance Testing
  - Time usage
- Functional Testing

Results & Discussions

Results : Evaluation on Test Set

Case	Details of Code Portion
1	Supplied Web Page containing C, Python, Java
2	Assembly Code from the Apollo Guidance Computer
3	SQL Code from this blog
4	Haskell Code from ShellCheck
5	Mix of Python and C Code
6	Java Code
7	Python Code

Results : Evaluation

Case	Precision (Code)	Recall (Code)	Precision (Text)	Recall (Text)
1	0.918	0.975	0.956	0.86
2	0.997	1.0	1.0	0.889
3	0.979	0.92	0.667	0.889
4	0.967	0.93	0.474	0.667
5	0.992	0.89	0.383	0.9
6	0.978	0.986	0.882	0.833
7	0.956	0.869	0.435	0.714

Results : Tests

- **Unit Tests :** All passed successfully
- **Performance Testing**
  - On given source : **1.32 seconds**
  - Text of Frankenstein comprising 75,000 words and 1665 lines : **3.74 seconds**
- **Functional Testing**
  - By nature, can’t achieve 100 % accuracy, but has good performance in meeting requirements and specifications

Discussions

- Surprisingly good performance
- Lightweight in terms of training data and time
- Code metrics better than text metrics
- Output is simply classified, not formatted or parsed
- High accuracy despite small and skewed dataset
- Quality of training and testing datasets need to be analyzed further

Test Case 5 Text Misclassification Example

Predict Text

Predict Code

```
# Predict the class for a given row
def predict(summaries, row):
	probabilities = calculate_class_probabilities(summaries, row)
	best_label, best_prob = None, -1
	for class_value, probability in probabilities.items():
		if best_label is None or probability > best_prob:
			best_prob = probability
			best_label = class_value
	return best_label
```

Limitations

- Granular upto line-level and not token level
- Needs new-line separation
- Will likely have inferior performance compared to heavyweight models
- Comments classified as text and not code

Limitations

- Doesn’t take into account context
- Naïve independence assumption & Bag of Words approach
- Hyperparameters and training size not tuned or optimized
- Datasets Quality

Conclusions

- Developed classifier that
  - Is Lightweight, Interpretable & Simple
  - Has strong results despite assumptions and improper training data
  - Can be used in a unified manner with more complex models, which needs to be further investigated
- Analysis of classified results needed to understand why it’s correctly or incorrectly classifying lines

Extensions and Future Work

- Add context and wider window with n-grams
- Semantic approach to induce newlines or,
- Semantic approach to classify within lines
- Extract knowledge from classifier output

Extensions and Future Work

- Tune hyperparameters like Laplacian constants
- Increase training data size and quality
- Write to Word doc without going through pandoc.readHTML
- User Interface
- More testing and profiling

Demo

References

- 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.
- 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
- Chatterjee, Preetha & Gause, Benjamin & Hedinger, Hunter & Pollock, Lori. (2017). Extracting Code Segments and Their Descriptions from Research Articles. 10.1109/MSR.2017.10.
- 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.
- https://github.com/Magalame/fastest-matrices
- D. Jurafsky and J. H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition. Prentice Hall, 2nd edition, 2009.

Thank you!