|  |  |  |
| --- | --- | --- |
| Patrick Tan  [ptan@college.harvard.edu](mailto:ptan@college.harvard.edu) | Timothy Kang  [tkang01@college.harvard.edu](mailto:tkang01@college.harvard.edu) | Kevin Zhang  [kevinzhang@college.harvard.edu](mailto:kevinzhang@college.harvard.edu) |

CS51 Final Project: Text Classification

**Brief Overview**

Documents exist to express an opinion or share information on a subject. The opinion or information shared can take many forms, but generally are limited to several possibilities for any particular topic. For example, reviews of a movie are generally either positive or negative; blog posts can be humorous or serious; and emails can be work-related or personal. The project we are undertaking seeks to develop and apply an algorithm to classify text documents based on their content. Using the Naïve Bayes algorithm, our algorithm will allow us to “seed” a program with a set of documents along with their known characteristics, and create a program that is able to classify a document presented to it using the same characteristics used to seed the program.

**Extensions**

The first extension we can implement involves writing the program in a dynamic way; although we seed our program using an initial set of data, the program, without a dynamic implementation, will become a static entity that does not grow as it is used. We can write our code in such a way as to continually update the stored data as more and more documents are fed to the program. This approach requires that our initial algorithm be accurate enough to ensure accurate results (for the most part) with later documents. The Naïve Bayes algorithm has been shown to be fairly accurate after only 250 test documents, so this should not be a concern.

The second extension we can implement is integration with the Twitter API (and potentially Facebook’s or Instagram’s as well). Tweets are especially great for our project since they are usually very opinionated and contain a few number of words that makes it easy to analyze and use to train the program. Implementing this extension may be difficult, but only because of technical issues not compatability; pulling data from the Internet and filtering out nonsense may be the hardest challenge here.

The third extension we can implement is something called n-grams. N-grams have to do with strings of words instead of individual ones; in particular, the modification we will make to our program in order to include this extension is allow the program to evaluate strings of words instead of only one word. Runtime and space complexity increase exponentially with this extension; in addition, coding a program to store every possible string of words will be difficult. A larger seed will be needed as well.

**Methods**

We will need two main methods for the bulk of this project: the first will read in sets of documents along with their classifications (that we have determined manually) and create a database that the program will use to evaluate future documents that are fed to it. The other method will take in a document that we wish the program to evaluate using the Naïve Bayes algorithm. Steps beyond this will include the extensions that are mentioned above.

**Structures**

Documents will be abstracted into a Document class; one variable will be a large string containing the full text of the document. One of the class methods should be a method that returns a list of strings, where each string is a single token in the large string delimited by spaces. Another will be an initializer that takes in a filename and reads in the data from that string; the last will be an initializer that just takes a string as an argument.

The seed that is passed into the analyzer will be implemented as a Seed class with one variable; this variable will be a dictionary of String:[Documents] elements. The string here will represent the heuristic (e.g. “positive”, “negative”, etc). The initializer will be a method that takes in an array of Strings and initializes the class variable to a dictionary of String:[] elements. There will be an additional method that updates the class variable by taking in a String then a Document. We may either use Python’s built-in dictionary structures or implement our own [String:BinTree] structure.

The analyzer program will be implemented as a Analyzer class. It will have one initializer method that takes in a Seed object. A Heuristic class will be implemented that is a collection of a string (the heuristic’s name), total word count, vocabulary, and a dictionary of Word:Count values. These values will correspond to the values calculated as part of the Naïve Bayes algorithm.

**Signatures/Interfaces**

class Document:

@ initializer

def \_\_init\_\_(self, text=None, filename=None):

# if text is None, initialize using filename

# if filename is None, initialize using text

# if both are None, throw exception

# text, filename are of type String

@ tokenizer

def tokenize(del=” “):

# tokenize the string using an optional delimiter

# del is of type String

class Seed:

@ initializer

def \_\_init\_\_(self, opts):

# initialize class variable using opts array

# opts is of type [String]

# class variable is of type [String:[Document]]

@ update stored variables

def update(opt, doc):

# update opt with the new doc

# opt is of type String

# doc is of type Document

class Analyzer:

@ utility class, hidden from users

class Heuristic:

@ initializer

def \_\_init\_\_(self, opt, text):

# initialize the Heuristic using the heuristic’s name (opt) and combined text (text) of all of the documents in the seed

# opt, text are of type String

@ updater

def update(self, doc):

# update the stored values with a new document

# doc is of type Document

@ initializer

def \_\_init\_\_(self, seed):

# initialize the object

# seed is of type Seed

@ updater

def update(self, opt, doc):

# update stored values using a new doc

# opt is of type String

# doc is of type Document

@ analyzer

def analyze(self, doc):

# analyze a new document using the stored values

# needs to output the probabilities of each of the stored heuristics, ranked from highest to lowest probability

# doc is of type Document

class BinaryTree:

# initializer

def \_\_init\_\_(self, val=None):

# create initial tree with empty left and right nodes

# root node is None by default

# val is any type

# accessor methods

def get\_left(self):

# return left tree

def get\_right(self):

# return right tree

def set\_node\_val(self, val=None):

# set root node value

# root node is None by default

# val has any type

def get\_node\_val(self):

# return node value

def insert\_left(self, val=None):

# insert value into left subtree

def insert\_right(self, val=None):

# insert value into right subtree

def is\_leaf(self):

# returns true if subtrees are empty, false otherwise

def print\_tree(self):

# prints tree from left to right

Timeline

* Find test text documents (4/18, Saturday)
* Write Document class (4/19, Sunday)
* Write BinTree class (4/20, Monday)
* Write Seed class (4/20, Monday)
* Write Heuristic class (4/21, Tuesday)
* Write Analyzer class (4/22, Wednesday)
* Test on documents and graph performance (4/23, Thursday)
* Adapt code to learn dynamically (4/24, Friday)
* Write Twitter API integration code (4/26, Sunday)
* Test on tweets (4/27, Tuesday)
* Write n-grams code (4/29, Thursday)
* Test n-grams code (4/30, Friday)
* Finish report (4/30, Friday)
* Finish up any last minute changes and submit (5/1, Saturday)