CSE13s Winter 2022
Assignment 7: Author Identification
Theodore Ikehara, tikehara@ucsc.edu
Due: March 13th at 11:59 pm
Design Doc (Draft)

1. Introduction:
   I. About:

      In this assignment we will attempt to apply stylometry to text to identify an author. Stylometry is the study of how certain people write and their choice of words. Stylometry has been able to be applied to many other studies not only linguistics, but also including and not limited to music, coding, and genetics. We will be using this analysis technique to attribute authorship of certain writings. We will be making use of a large database that contains the works of many different authors, and given sample text with anonymous authors we will attempt to find the most likely author given this database. In order to attempt this we will be using the k-nearest neighbors algorithm.

      This algorithm is commonly used in machine learning and allows for classification of different objects. In this particular assignment we will be using this algorithm to compute the distance of the text to the authors contained in the database and determine which is closer.

   II. Manhattan Distance:

      This is one of the methods we will be using to compute the distance between each component of each vector. This will be the easiest of the three. All that needs to be done is to take the magnitude of the difference between each component of the vectors. This is shown in the formula below. You take the magnitude here

$$\mathrm{MD} = \sum_{i \in n} |u_i - v_i|$$

   III. Euclidean Distance:

The Euclidean distance is very similar to the Manhattan distance. This is similar to finding the hypotenuse of a triangle. This distance is close to what we are calculating for the vectors. The formula is shown below. This is close to the pythagorean theorem.

$$ED = \sqrt{\sum_{i \in n} (u_i - v_i)^2}$$

IV. Cosine Distance:

The cosine distance will be taken by a similar method to how cosines of angles between two vectors after taking the relative cosine the vector needs to be normalized. The formulas are shown below.

$$Cos(\Theta) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \times |\vec{v}|}$$

$$\vec{u} \cdot \vec{v} = \sum_{i \in n} u_i \times v_i$$

2. The Implementation: This is the description of the ADT implementations

I. Hash Tables:

This is going to be our solution to storing and retrieving unique words quickly to be able to compare them to our database. A hash table is almost like a dictionary in python where a hash table maps keys to values and creates a pair. This method of creating a hash table generally provides fast constant lookups. This is typically done by creating a hash key and then putting the values in that index thus, being able to call the values really quickly. Sometimes the keys may overlap and when this happens we need to fix the values so the keys are unique.

Hash table create node that contains the word and the count

II. Hash Table Iterator:

This will be the implementation of the mechanism that allows us to iterate through the entire hash table. As C is not a high level

language where you can just call the values and get the value of the keys like in python or other languages. We need to manually create a way to iterate through our hash table to get the values.

III. Nodes:

The words that we get from the text files will be stored with these nodes along with the count of the appearance of the nodes

IV. Bloom Filter:

This implementation of the bloom filters allows us to make our hash tables more efficient as for we will be using very full hash tables it will take longer to iterate through the hash table for each value. And thus we use a bloom filter to make the hash table more time and space efficient. False positives for this implementation are possible but false negatives are not. This fact is important to keep in mind when executing and using this implementation.

V. Bit Vectors:

Bit vectors will be the way that we use to find the distance between the similarities with the formulas. This bit vector will represent a one dimensional array of bits. In which the array will be of 1 and 0, true or false.

VI. Texts:

This will be the ADT that allows us to interact with the text. This ADT will contain all the functions that parse and interact with the text files provided. This will give us a bridge for our program to interact and interpret the text file.

VII. Priority Queue:

This is a queue that sorts itself whenever we enqueue items. This is a very similar ADT to the one created previously for assignment 6. This will be implemented using insertion sort when items are enqueued.

Create a new node that has the types that can be queued.

VIII. Identify.c:

This is where the main function is located. This will make use of all the other ADTs and implementations that we have created. The main function's main purpose is to take in a text file and calculate the percentage that it is written by one of the provided authors that are in

the big database. This will also make use of the command line options so users can easily interact with the program.

3. Pseudo Code top level:
    a. Hash Table:
        i. Ht_create:
            // creating the hashtable type
            // defines the size
            // defines the salts
            // dynamically allocates the node to the hash table
            // returns the hashtable
        ii. Ht_delete
            // iterates through the nodes to delete them
            // deletes the slots
            // deletes the hash table
        iii. Ht_size
            // returns the size of the hashtable
        iv. Ht_lookup
            // this creates the index with the hash function
            // if the index exceeds the cap
            // returns NULL
            // if the strings are the same return the node
        v. Ht_insert

            // This finds the index using hash
            // if the index exceeds its cap
            // comparing the words if they are the same
            // increments count if the same word is inserted
            // increments the index after
            // inserts the node here
        vi. Hti_create
            // creates a hashtable
            // equates the two
            // sets the slot location to 0

vii. Hti_delete
// frees the memory location
viii. ht_iter
// iterates through the entire hash table by adding one to the slot every time

b. Nodes:
I. node_create
// creates the node
// the node keeps a word and the amount of times this word occurs
II. node_delete
// this deletes this node

c. Bloom Filter:
I. Bf_create
// allocates the space for the bloom filter here
// setting primary bloomfilter
// setting secondary bloomfilter
// setting tertiary bloomfilter
// returns the bloom filter
II. Bf_delete
// frees the bloomfilter array
// then frees the bloomfilter object
III. Bf_insert
// gets the length of the bv and mods it to find location
// insert with primary hash
// insert with secondary hash
// insert with tertiary hash
IV. bf_size
// returns the length of the bloomfilter
V. bf_probe
// this is the length of the bv
// get with primary
// get with secondary
// get with tertiary

// return pri && sec && tri;

d. Bit Vectors:
   I. Bv_create
      // dynamically allocates the space for the bit vector
         // allocates space for the vector itself
      // returns the created bit vector
   II. Bv_delete
      // frees the vector first
      // frees the entire adt
   III. Bv_length
      // gets the length
   IV. Bv_set_bit
      // checks to see if i is in length
      // calculates the byte
      // calculates the location
      // set the bit
      // returns true if set success

   V. Bv_clr_bit
      // checks to see if i is in length
      // sets the byte
      // location
      // bit mask
      // clears the bit
      // returns true if success
      // return true

   VI. Bv_get_bit
      // checks if this location is valid
      // gets the bit
      // this is if the bit is equal to 1
      // this is if the bit is 0
      // return false;

e. Texts:

I. Regex: [a-zA-Z] all the alphabet (‘|-)

II. text_create

// creating the text data type here

// 2^21 for the bloom filter size

// 2^19 for the hash table

// allocates memory to text

// if this space is allocated correctly

// inside loop

// lowers the case of the word

// inserts the word into the hash table or bloom filter depending on

// weather this is noise text or the actual text

// This occurs when this is the noise text

// loads in all the values to bloomfilter

// accounts for the noise limit

// this occurs when the text is not noise text

// checking weather the noise is in the bloom filter

// needs check bloom filter correctness

// iterating the create...

// might be in bloom filter

// else insert

III. Text_delete

// delete the hash table

// delete the bloomfilter

// delete the text

IV. Euclidean distance

// create two hti to iterate through text1 and text2

// keep the sum

// find the difference between distance the two texts

// (u - v)^2

// sum this

// return the square root of sum

V. Manhattan distance

// same as the euclidean

// but instead sum |u - v|

VI. Cosine distance
// This one we will use
// u * v
// sum of this
// then return 1-sum

VII. Text_dist
// we will use a metric enumeration to see which formula to use

VIII. Text_frequency
// finds the normalized frequency of the words in the text
// the node for the looked up word
// if the word is not in the text

IX. text_contains
// search in the hashtable if it is in it
// if word is inside then return true
// if not in return false

f. Priority Queue:
// create a node adt that pq can make use of having it contain an author and a dist

I. Pq_create
// creating the pq
// setting all the values
// loads in the items so no segfault

II. Pq_delete
// delete the nodes in a loop
// delete the auth node
// delete the pq

III. Pq_empty
// return true is pq is empty
// false otherwise

IV. Pq_full
// return true if pq is full
// return false otherwise

V. Pq_size
// returns the size of the pq

VI. Enqueue

// this checks if the pq is full first
// this puts in the node to the pq
// swap_temp(q->auth[0]->author, author);
// printf("%s\n", q->auth[0]->author);
// sorting with insertion sort
// sorting nested loop
// checking if previous index is greater than itself
// swaps here
// sorting...

VII. dequeue

// checks if the pq is empty first
// takes out this position
// makes sure to decrement the top

g. Identify.c:

- Create a noise text
- Create an anonymous text
- Opens database
- Create priority queue that holds author and distance pair
- Reads in the first line from db tells the size of the pq
- In loop
- Fgets author
- Fgets text location compare with anonymous text
- Enqueues the author and the dist
- Then dequeue k times to show witch is most similar