Assignment 7
Author Identification

Prof. Darrell Long
CSE 13S – Winter 2023

First `DESIGN.pdf` draft due: March 16th at 11:59 pm PST
Assignment formally due: March 19th at 11:59 pm PST

1 Introduction

Stylometry is the application of the study of linguistic style. That is, by analyzing the linguistic styles, or *diction*, of certain authors, it becomes possible to identify the author of some anonymous sample of text by comparing its linguistic style to the styles of the analyzed authors. In practice, stylometry has been used for more than just author identification, but also composer identification, artist identification, and even programmer identification. With regard to the latter, stylometry has even been used to detect code plagiarism, flagging cases where the code style is too similar.

For your assignment, you will be creating a program that attempts to identify the most likely authors for an anonymous sample of text given a large database of texts with known authors. Modern-day stylometry usually is performed using machine learning, achieving high identification accuracies by learning over time, but implementing this from scratch would take an extraordinary effort. We instead will settle and be content with using an algorithm that's commonly used in machine learning to identify authors of anonymous samples of text, albeit less accurately. What is this algorithm? It is the k-nearest neighbors algorithm.

2 k-Nearest Neighbors

Commonly used in machine learning, the k-nearest neighbors (KNN) algorithm is typically used for either classification or regression of certain objects. In our case, we wish to classify samples of texts and determine which authors are most likely to have authored an anonymous sample of text.

Assume you have n samples of text and 1 anonymous sample of text to identify. To identify the top k likely authors of the sample text, we compute the distance from this anonymous sample to each of the n known samples using their respective *feature vectors*. A feature vector in this case is simply the count of various words in the sample of text. This means that we effectively have an n-dimensional vector, where n is the number of unique words in some text.

The basic algorithm to identify likely candidates as authors of some anonymous text is as follows: First, you must first read a set of sample texts from known authors. You will then count the unique words and total number of words in each of these texts. This will allow you to compute the frequency of occurrence of each word in each of the texts. You will do the same for the anonymous sample of text. From these frequencies, you can then attempt to identify the author of the sample by computing

the *distance* of the sample to each of the known texts. There are three distance metrics that you are considering to use: *Manhattan distance, Euclidean distance,* and *cosine distance.*

As an example to illustrate the three metrics we will use the examples texts:

1. "Hello world"
2. "Goodbye, goodbye world"

These will give us 3-dimensional vectors which correspond to: < "hello", "goodbye", "world" >, which the texts have respectively as:

1. $\vec{u'} =< 1,0,1 >$
2. $\vec{v'} =< 0,2,1 >$

Notice that "hello" and "goodbye" have been changed to their lowercase form and that we ignore the punctuation. We then want to normalize these vectors so that they each have a magnitude of 1, giving us:

1. $\vec{u} = \frac{\vec{u'}}{|\vec{u'}|} =< \frac{1}{2}, \frac{0}{2}, \frac{1}{2} >$
2. $\vec{v} = \frac{\vec{v'}}{|\vec{v'}|} =< \frac{0}{3}, \frac{2}{3}, \frac{1}{3} >$

We will define vectors $\vec{u}$ and $\vec{v}$ as having components $< u_1, u_2, \ldots, u_{n-1}, u_n >$ and $< v_1, v_2, \ldots, v_{n-1}, v_n >$, respectively, where they are the vectors of two texts we want to compare.

2.1 Manhattan Distance

The *Manhattan distance* is the simplest of the three to compute. To compute it, you simply take the magnitude of the difference between each component of the vector. You can think of it as how far you would have to go if you had to follow the gridded streets of Manhattan. This is computed as:

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

With the example texts we get the absolute value of the component distances before summing them:

- "hello" : $|u_1 - v_1| = |\frac{1}{2} - \frac{0}{3}| = \frac{1}{2}$
- "goodbye" : $|u_2 - v_2| = |\frac{0}{2} - \frac{2}{3}| = \frac{2}{3}$
- "world" : $|u_3 - v_3| = |\frac{1}{2} - \frac{1}{3}| = \frac{1}{6}$

This gives us a Manhattan distance of $\frac{4}{3}$.

2.2 Euclidean Distance

The *Euclidean distance* is calculated very similarly to the Manhattan distance, except you are able to go as the bird flies. Thus, we just need to remember how Euclid taught us to find the length of the hypotenuse of a triangle, so we add the squares of all the magnitudes together before getting the square root of the sum. This is computed as:

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

With the example texts we first square the component distances before summing them:

- "hello" : $(u_1 - v_1|)^2 = (\frac{1}{2} - \frac{0}{3})^2 = \frac{1}{4}$
- "goodbye" : $(u_2 - v_2|)^2 = (\frac{0}{2} - \frac{2}{3})^2 = \frac{4}{9}$
- "world" : $(u_3 - v_3|)^2 = (\frac{1}{2} - \frac{1}{3})^2 = \frac{1}{36}$

We then add all the values together to get a sum of $\frac{26}{36}$, giving us a Euclidean distance of $\sqrt{\frac{26}{36}} \approx 0.85$.

2.3 Cosine Distance

The *cosine distance* is different from the first two in that we derive it from the *cosine similarity* of two vectors, or the cosine of the angle between two vectors.

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

If we are to normalize the vectors, the bottom of the fraction is just 1, and so we only need to get the dot product of the two vectors:

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

Note that the angle approaches 0 as the cosine of the angle approaches 1. To stay consistent with the other metrics, we instead get the cosine *distance* by subtracting the similarity from 1. This ensures that the texts that are more similar will have a smaller computed distance. For our example texts we multiply the vector components together before summing all the results:

- "hello" : $u_1 \times v_1 = \frac{1}{2} \times \frac{0}{3} = 0$
- "goodbye" : $u_2 \times v_2 = \frac{0}{2} \times \frac{2}{3} = 0$
- "world" : $u_3 \times v_3 = \frac{1}{2} \times \frac{1}{3} = \frac{1}{6}$

The computed cosine similarity is $\frac{1}{6}$. We subtract this from 1 for the cosine distance, giving us a cosine distance of $1 - \frac{1}{6} = \frac{5}{6}$.

3 Hash Tables

You will require some method to quickly store and retrieve unique words in a sample of text, as well as the count of each unique word. A hash table would be the best data structure for this purpose. A hash table maps keys to values and provides, on average, fast *constant time* look-ups. It does so typically by taking a key k, hashing it with some hash function $h(x)$, and placing the key's corresponding value in an underlying array at index $h(k)$. This is the perfect way to store all unique words in a sample of text along with their respective counts. So what happens when two words have the same hash value? This is called a *hash collision*, and must be resolved. To resolve these collisions, you will perform *open addressing* using *linear probing*.

The open addressing method of resolving hash collisions requires searching through the hash table if index $h(k)$ for some key k doesn't contain k. This act of searching, or *probing*, can be done in multiple ways, the simplest of which is *linear probing*. With linear probing, successive indices of the hash table starting from $h(k)$ are probed until either k is found, or an empty slot shows up. If an empty slot shows up before k is found, it means k was never inserted. Using open addressing, it's possible that a key cannot be inserted into a hash table because it is completely filled up.

A hash function maps data of arbitrary size to fixed-size values. Its values are called hash values, hash codes, digests, or simply hashes, which index a fixed-size table called a hash table. Hash functions and their associated hash tables are used in data storage and retrieval applications to access data (on average) at a nearly constant time per retrieval. They require a storage space that is only fractionally greater than the total space needed for the data. Hashing avoids the non-linear access time of ordered and unordered lists and some trees and the often exponential storage requirements of direct access of large state spaces.

Hash functions rely on statistical properties of key and function interaction: worst-case behavior is an exhaustive search but with a vanishingly small probability, and average-case behavior can be nearly constant (with minimal collisions).

Below is the `struct` definition for a hash table. The hash table contains a salt that will be passed to the given hash function, `hash()` — more on this in §5. Each hash table entry is a `Node` that contains a unique word and its count: the number of times is appears in a sample of text. The node ADT will be discussed in §6. Since we are resolving hash collisions with open addressing, it only makes sense to store these nodes in an array.

```
1 struct HashTable {
2   uint64_t salt[2]; // The salt to use for the hash function.
3   uint32_t size;    // The number of slots in the hash table.
4   Node **slots;     // The array of hash table items.
5 };
```

This `struct` definition *must* go in `ht.c`.

`HashTable *ht_create(uint32_t size)`

The constructor for a hash table. The `size` parameter denotes the number of slots that the hash table can index up to. The salt for the hash table is provided in `salts.h`.

`void ht_delete(HashTable **ht)`

The destructor for a hash table. This should free any remaining nodes left in the hash table. Remember

to set the pointer to the freed hash table to `NULL`.

`uint32_t ht_size(HashTable *ht)`

Returns the hash table's size, the number of slots it can index up to.

`Node *ht_lookup(HashTable *ht, char *word)`

Searches for an entry, a node, in the hash table that contains `word`. If the node is found, the pointer to the node is returned. Otherwise, a `NULL` pointer is returned.

`Node *ht_insert(HashTable *ht, char *word)`

Inserts the specified `word` into the hash table. If the word already exists in the hash table, increment its count by 1. Otherwise, insert a new node containing `word` with its count set to 1. Again, since we're using open addressing, it's possible that an insertion fails if the hash table is filled. To indicate this, return a pointer to the inserted node if the insertion was successful, and return `NULL` if unsuccessful.

`void ht_print(HashTable *ht)`

A debug function to print out the contents of a hash table. Write this immediately after the constructor.

4 Iterating Over Hash Tables

You will need some mechanism to iterate over all the entries in a hash table. High-level languages such as **Python** and **Rust** natively provide iterators to do exactly this. **C**, on the other hand, provides no such utility. This means that you will need to implement your own hash table iterator type. This will be called `HashTableIterator` and should be defined in `ht.c`. The hash table and the hash table iterator should be implemented in the same file. You must use the following `struct` definition for your hash table iterator.

```
1 struct HashTableIterator {
2     HashTable *table; // The hash table to iterate over.
3     uint32_t slot;    // The current slot the iterator is on.
4 };
```

Storing each hash table entry in its own array index makes the hash table iterator implementation almost trivial. All an iterator needs to do is keep track of which slot it has iterated up to in the hash table. The following sections are the functions that must be implemented for your hash table iterator interface.

`HashTableIterator *hti_create(HashTable *ht)`

Creates a new hash table iterator. This iterator should iterate over the `ht`. The `slot` field of the iterator should be initialized to 0.

`void hti_delete(HashTableIterator **hti)`

Deletes a hash table iterator. You *should not* delete the `table` field of the iterator, as you may need to iterate over that hash table at a later time.

`Node *ht_iter(HashTableIterator *hti)`

Returns the pointer to the next valid entry in the hash table. This may require incrementing the `slot` field of the iterator multiple times to get to the next valid entry. Return `NULL` if the iterator has iterated over the entire hash table.

Example hash table iterator usage.

```
1  HashTable *ht = ht_create(4);
2  ht_insert(ht, "hello");
3  ht_insert(ht, "world");
4  HashTableIterator *hti = hti_create(ht);
5  Node *n = NULL;
6  while ((n = ht_iter(hti)) != NULL) {
7    print("%s\n", n->word);
8  }
9  hti_delete(&hti);
10 ht_delete(&ht);
```

The above code should print either "hello" then "world" or "world" then "hello" depending on how the hash function works.

5 Hashing with the SPECK Cipher

You will need a good hash function to use in your Bloom filter (discussed in §7) and hash table (discussed in §3). We will have discussed hash functions in lecture, and rather than risk having a poor one implemented, we will simply provide you one. The SPECK[1] block cipher is provided for use as a hash function.

SPECK is a family of lightweight block ciphers publicly released by the National Security Agency (NSA) in June 2013. SPECK has been optimized for performance in software implementations, while its sister algorithm, SIMON, has been optimized for hardware implementations. SPECK is an add-rotate-xor (ARX) cipher. The reason a cipher is used for this is because encryption generates random output given some input; exactly what we want for a hash.

Encryption is the process of taking some file you wish to protect, usually called plaintext, and transforming its data such that only authorized parties can access it. This transformed data is referred to as ciphertext. Decryption is the inverse operation of encryption, taking the ciphertext and transforming the encrypted data back to its original state as found in the original plaintext. Encryption algorithms that utilize the same key for both encryption and decryption, like SPECK, are symmetric-key algorithms, and algorithms that don't, such as RSA, are asymmetric-key algorithms.

You will be given two files, `speck.h` and `speck.c`. The former will provide the interface to using the SPECK hash function which has been named `hash()`, and the latter contains the implementation. The hash function `hash()` takes two parameters: a 128-bit salt passed in the form of an array of two `uint64_t`s, and a key to hash. The function will return a `uint32_t` which is exactly the index the key is mapped to.

[1] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers, "The SIMON and SPECK lightweight block ciphers." In Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE, 2015.

```
uint32_t hash(uint64_t salt[], char *key);
```

6 Nodes

As mentioned in §3, each array slot will contain a node. Each node should contain a word and its count. The node `struct` is defined for you in `node.h` as follows:

```
struct Node {
    char *word;
    uint32_t count;
};
```

The rest of the interface for the node ADT is provided in `node.h`. The node ADT is made transparent in order to simplify the program implementation.

`Node *node_create(char *word)`

The constructor for a node. You will want to make a *copy* of the `word` that is passed in. This will require *allocating memory* and copying over the characters for the `word`. You may find `strdup()` useful.

`void node_delete(Node **n)`

The destructor for a node. Since you have allocated memory for `word`, remember to free the memory allocated to that as well. The pointer to the node should be set to `NULL`.

`void node_print(Node *n)`

A debug function to print out the contents of a node.

7 Bloom Filters

With open addressing, it's possible that you need to iterate over the entire hash table (if it's full), before you can come to the conclusion that some key isn't present. This is very unsatisfactory with very large and full hash tables. Is there a faster way to check if a key has never been seen? That is where the *Bloom filter* comes into play.

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton H. Bloom in 1970, and is used to test whether an element is a member of a set. False-positive matches are possible, but false negatives are not—in other words, a query for set membership returns either "possibly in the set" or "definitely not in the set." Elements can be added to the set but not removed from it; the more elements added, the higher the probability of false positives.

A Bloom filter can be represented as an array of m bits, or a **bit vector**. A Bloom filter should utilize k different hash functions. Using these hash functions, a set element added to the Bloom filter is mapped to at most k of the m bit indices, generating a uniform pseudo-random distribution. Typically, k is a small constant which depends on the desired false error rate ϵ, while m is proportional to k and the number of elements to be added.

Assume you are adding a word w to your Bloom filter and are using $k = 3$ hash functions, $f(x)$, $g(x)$, and $h(x)$. To add w to the Bloom filter, you simply set the bits at indices $f(w)$, $g(w)$, and $h(w)$. To check if some word w' has been added to the same Bloom filter, you check if the bits at indices $f(w')$, $g(w')$, and $h(w')$ are set. If they are all set, then w' has *most likely* been added to the Bloom filter. If *any one* of those bits was cleared, then w' has definitely *not* been added to the Bloom filter. The fact that the Bloom filter can only tell if some word has *most likely* been added to the Bloom filter means that *false positives* can occur. The larger the Bloom filter, the lower the chances of getting false positives.

So what do Bloom filters mean for you? It means you can add each word of some sample of text into your Bloom filter. When comparing the words of this sample with the words of another sample, it is much more efficient to simply query the Bloom filter of the other sample of text for whether not a word has been seen, rather than look it up in the hash table. You decide to implement a Bloom filter with *three* salts for *three* different hash functions. Why? To reduce the chance of a *false positive*.

You can think of a "salt" as an initialization vector or a key. Using different salts with the same hash function results in a different, unique hash. Since you are equipping your Bloom filter with three different salts, you are effectively getting three different hash functions: $f(x)$, $g(x)$, and $h(x)$. Hashing a word w, with extremely high probability, should result in $f(w) \neq g(w) \neq h(w)$. These salts are to be used for the SPECK cipher, which requires a 128-bit key, so we have used MD5 [2] "message-digest" to reduce three books down to 128 bits each. These salts are provided for you in `salts.h`. Do not change them. You will use the SPECK cipher as a hash function, as discussed in §5.

The following `struct` defines the `BloomFilter` ADT. The three salts will be stored in the `primary`, `secondary`, and `tertiary` fields. Each salt is 128 bits in size. To hold these 128 bits, we use an array of two `uint64_t`s.

```
1 struct BloomFilter {
2   uint64_t primary[2];   // Primary hash function salt.
3   uint64_t secondary[2]; // Secondary hash function salt.
4   uint64_t tertiary[2];  // Tertiary hash function salt.
5   BitVector *filter;
6 };
```

This `struct` definition *must* go in `bf.c`.

`BloomFilter *bf_create(uint32_t size)`

The constructor for a Bloom filter. The primary, secondary, and tertiary salts that should be used are provided in `salts.h`. Note that you will also have to implement the bit vector ADT for your Bloom filter, as it will serve as the array of bits necessary for a proper Bloom filter. Bit vectors will be discussed in §8.

`void bf_delete(BloomFilter **bf)`

The destructor for a Bloom filter. As with all other destructors, it should free any memory allocated by the constructor and null out the pointer that was passed in.

`uint32_t bf_size(BloomFilter *bf)`

Returns the size of the Bloom filter. In other words, the number of bits that the Bloom filter can access.

[2] Rivest, R.. "The MD5 Message-Digest Algorithm." RFC 1321 (1992): 1-21.

Hint: this is the length of the underlying bit vector.

`void bf_insert(BloomFilter *bf, char *word)`

Takes `word` and inserts it into the Bloom filter. This entails hashing `word` with each of the three salts for three indices, and setting the bits at those indices in the underlying bit vector.

`bool bf_probe(BloomFilter *bf, char *word)`

Probes the Bloom filter for `word`. Like with `bf_insert()`, `word` is hashed with each of the three salts for three indices. If all the bits at those indices are set, return `true` to signify that `word` was most likely added to the Bloom filter. Else, return `false`.

`void bf_print(BloomFilter *bf)`

A debug function to print out the bits of a Bloom filter. This will ideally utilize the debug print function you implement for your bit vector.

8 Bit Vectors

A bit vector is an ADT that represents a one dimensional array of bits, the bits in which are used to denote if something is true or false (1 or 0). This is an efficient ADT since, in order to represent the truth or falsity of a bit vector of n items, we can use $\lceil \frac{n}{8} \rceil$ `uint8_t`s instead of n, and being able to access 8 indices with a single integer access is extremely cost efficient. Since we cannot directly access a bit, we must use bitwise operations to get, set, and clear a bit within a byte. Much of the bit vector implementation can be derived from your implementation of the `Code` ADT used in assignment 6. If there were any issues with getting, setting, or clearing bits in that assignment, make sure you address them here.

```
1 struct BitVector {
2   uint32_t length;
3   uint8_t *vector;
4 };
```

This `struct` definition *must* go in `bv.c`.

`BitVector *bv_create(uint32_t length)`

The constructor for a bit vector that holds `length` bits. In the even that sufficient memory cannot be allocated, the function must return `NULL`. Else, it must return a `BitVector *)`, or a pointer to an allocated `BitVector`. Each bit of the bit vector should be initialized to 0.

`void bv_delete(BitVector **bv)`

The destructor for a bit vector. Remember to set the pointer to `NULL` after the memory associated with the bit vector is freed.

`uint32_t bv_length(BitVector *bv)`

Returns the length of a bit vector.

`bool bv_set_bit(BitVector *bv, uint32_t i)`

Sets the i^{th} bit in a bit vector. If `i` is out of range, return `false`. Otherwise, return `true` to indicate success.

`bool bv_clr_bit(BitVector *bv, uint32_t i)`

Clears the i^{th} bit in the bit vector. If `i` is out of range, return `false`. Otherwise, return `true` to indicate success.

`bool bv_get_bit(BitVector *bv, uint32_t i)`

Returns the i^{th} bit in the bit vector. If `i` is out of range, return `false`. Otherwise, return `false` if the value of bit `i` is `0` and return `true` if the value of bit `i` is `1`.

`void bv_print(BitVector *bv)`

A debug function to print the bits of a bit vector. That is, iterate over each of the bits of the bit vector. Print out either 0 or 1 depending on whether each bit is set. You should write this immediately after the constructor.

9 Lexical Analysis with Regular Expressions

You will need a function to parse out the words from samples of text, which will be passed to you in the form of input streams. You are concerned with normal words, *contractions*, and *hyphenations*. A normal word is any sequence of one or more characters that are part of your regular expression word character set. Your word character set should contain characters from a–z and A–Z. You should also accept contractions like "don't" and "y'all've" and hyphenations like "pseudo-code" and "move-to-front".

You will need to write your own *regular expression* for a word, utilizing the `regex.h` library to lexically analyze the input stream for words. You will be given a parsing module that lexically analyzes the input stream using your regular expression. You are not required to use the module itself, but it is *mandatory* that you parse through an input stream for words using at least one regular expression. The interface for the parsing module will be in `parser.h` and its implementation will be in `parser.c`.

The function `next_word()` requires two inputs, the input stream `infile`, and a pointer to a compiled regular expression, `word_regex`. Notice the word *compiled*: you must first compile your regular expression using `regcomp()` before passing it to the function. Here is a small program that prints out words input to `stdin` using the parsing module. In the program, the regular expression for a word matches one or more lowercase and uppercase letters. The regular expression you will have to write for your assignment will be more complex than the one displayed here, as it is just an example.

Example program using the parsing module.

```c
#include "parser.h"
#include <regex.h>
#include <stdio.h>

#define WORD "[a-zA-Z]+"

int main(void) {
    regex_t re;
    if (regcomp(&re, WORD, REG_EXTENDED)) {
        fprintf(stderr, "Failed to compile regex.\n");
        return 1;
    }

    char *word = NULL;
    while ((word = next_word(stdin, &re)) != NULL) {
        printf("Word: %s\n", word);
    }

    regfree(&re);
    return 0;
}
```

10 Texts

We will need a new ADT to encapsulate the parsing of a text and serve as the in-memory representation of the distribution of words in the file. This is the `Text` ADT. It will contain a hash table, Bloom filter, and the count of how many words are in the text. This is *not* the number of unique words in the text, but the *total* number of words in the text. The following `struct` definition *must* go in `text.c`.

```c
struct Text {
  HashTable *ht;
  BloomFilter *bf;
  uint32_t word_count;
};
```

`Text *text_create(FILE *infile, Text *noise)`

The constructor for a text. Using the regex-parsing module, get each word of `infile` and convert it to *lowercase*. The `noise` parameter is a `Text` that contains noise words to filter out. That is, each parsed, lowercase word is only added to the created `Text` if and only if the word doesn't appear in the noise `Text`.

Why are we ignoring certain words? As you can imagine, certain words of the English language occur quite frequently in writing, words like "a", "the", and "of". These words aren't great indicators of an author's unique diction and thus add additional noise to the computed distance. Hence, they should be ignored. If `noise` is `NULL`, then the `Text` that is being created is the noise text itself.

If sufficient memory cannot be allocated, the function must return `NULL`. Else, it must return a `Text *`, or a pointer to an allocated `Text`. The hash table should be created with a size of 2^{19} and the Bloom filter should be created with a size of 2^{21}.

`void text_delete(Text **text)`

Deletes a text. Remember to free both the hash table and the Bloom filter in the text before freeing the text itself. Remember to set the pointer to `NULL` after the memory associated with the text is freed.

`double text_dist(Text *text1, Text *text2, Metric metric)`

This function returns the distance between the two texts depending on the metric being used. This can be either the Euclidean distance, the Manhattan distance, or the cosine distance. The `Metric` enumeration is provided to you in `metric.h` and will be mentioned as well in §12.

Remember that the nodes contain the counts for their respective words and still need to be normalized with the total word count from the text.

`double text_frequency(Text *text, char* word)`

Returns the frequency of the word in the text. If the word is not in the text, then this must return 0. Otherwise, this must return the normalized frequency of the word.

`bool text_contains(Text *text, char* word)`

Returns whether or not a word is in the text. This should return `true` if `word` is in the text and `false` otherwise.

`void text_print(Text *text)`

A debug function to print the contents of a text. You may want to just call the respective functions of the component parts of the text.

11 Priority Queue

You will use a priority queue to store the names of author along with the distance calculated between text authored by the author the anonymous sample of text. This interface for this priority queue is designed specifically for this task. It should enqueue and dequeue pairs of the author and the corresponding distance as separate parameters instead of as a single struct. This means that you can implement your priority queue as you wish, so long as you follow the defined API.

`PriorityQueue *pq_create(uint32_t capacity)`

The constructor for a priority queue that holds up to `capacity` elements. In the event that sufficient Memory cannot be allocated, the function must return `NULL`. Else, it must return a `PriorityQueue *`, or a pointer to an allocated `PriorityQueue`. The priority queue should initially contain no elements.

`void pq_delete(PriorityQueue **q)`

The destructor for a priority queue. Remember to set the pointer to `NULL` after the memory associated with the priority queue is freed. Anything left in the priority queue that hasn't been dequeued should be freed as well.

`bool pq_empty(PriorityQueue *q)`

Returns `true` if the priority queue is empty and `false` otherwise.

`bool pq_full(PriorityQueue *q)`

Returns `true` if the priority queue is full and `false` otherwise.

`uint32_t pq_size(PriorityQueue *q)`

Returns the number of elements in the priority queue.

`bool enqueue(PriorityQueue *q, char *author, double dist)`

Enqueue the `author`, `dist` pair into the priority queue. If the priority queue is full, return `false`. Otherwise, return `true` to indicate success.

`bool dequeue(PriorityQueue *q, char **author, double *dist)`

Dequeue the `author`, `dist` pair from the priority queue. The pointer to the author string is passed back with the `author` double pointer. The distance metric value is passed back with the `dist` pointer. If the priority queue is empty, return `false`. Otherwise, return `true` to indicate success.

`void pq_print(PriorityQueue *q)`

A debug function to print the priority queue.

12 Your Task

You will be writing a program called `identify` that identifies the top k likely authors that authored an anonymous sample of text passed into `stdin`. Your program will need to handle the following command-line options:

- `-d` : Specify path to database of authors and texts. The default is `lib.db`.
- `-n` : Specify path to file of noise words to filter out. The default is `noise.txt`.
- `-k` : Specify the number of matches. The default is 5.
- `-l` : Specify the number of noise words to filter out. The default is 100.
- `-e` : Set the distance metric to use as the Euclidean distance. This should be the default metric if no other distance metric is specified.
- `-m` : Set the distance metric to use as the Manhattan distance.
- `-c` : Set the distance metric to use as the cosine distance.
- `-h` : Display program help and usage.

You will be provided with a file, `metric.h`, that enumerates each of the possible distance metrics and contains an array of strings, one for each distance metric. Your main program, implemented in `identify.c`, should do the following:

- Create a new `Text` that contains words in the noise word file. This will be the noise word `Text` that contains words to filter out. Make sure that each word in the noise word `Text` is lowercased.
- Create a new anonymous `Text` from text passed in to `stdin`. This is the text that you want to identify. Remember to filter out all the noise words in the noise word `Text` that you just created.
- Open up the database of authors and text files. This file, by default, should be `lib.db`. The first line of this file indicates the number of author/text pairs. We'll refer to this number as n.
- Create a priority queue that can hold up to n elements. This will be used to efficiently find the top k likely authors who authored the anonymous text.
- Process the rest of the lines in the database file in pairs. You will need to do the following:
  - Use `fgets()` to read in the name of an author, then use `fgets()` again to read in the path to the file of text that author authored. You will need to remove the newline character that `fgets()` preserves.
  - Open up the author's file of text and use it to create a new `Text` instance. If the file could not be opened, continue to the next line of the database file. Remember to turn each parsed word to lowercase before checking if the word should be added to the `Text`.
  - Compute the distance between the author's `Text` and the anonymous `Text`, filtering out words in the noise word `Text`.
  - Enqueue the author's name and the computed distance into the priority queue.
- Get the top k likely matches by dequeueing the priority queue. Note that there may be less k entries in the priority queue, depending on whether or not a file could be processed.

Example program output is provided below. Note that the top k matches your program finds should match the reference program exactly, but the displayed distance may differ slightly due to floating point rounding errors.

```
$ ./identify -k 10 < texts/william-shakespeare.txt
Top 10, metric: Euclidean distance, noise limit: 100
1) William Shakespeare [0.000000000000000]
2) William D. McClintock [0.026516404305561]
3) Dante Alighieri [0.026865145178097]
4) Edgar Allan Poe [0.027829100437714]
5) Johann Wolfgang von Goethe [0.027873601844574]
6) Henry Timrod [0.028370953256889]
7) Lew Wallace [0.028546232963706]
8) Saxo Grammaticus [0.029282174356059]
9) Rudyard Kipling [0.029385152618790]
10) Various [0.029410943073263]
```

13 Deliverables

You will need to turn in the following source code and header files:

1. `bf.h`: Defines the interface for the Bloom filter ADT. Do not modify this.
2. `bf.c`: Contains the implementation of the Bloom filter ADT.
3. `bv.h`: Defines the interface for the bit vector ADT. Do not modify this.
4. `bv.c`: Contains the implementation of the bit vector ADT.
5. `ht.h`: Defines the interface for the hash table ADT and the hash table iterator ADT. Do not modify this.
6. `ht.c`: Contains the implementation of the hash table ADT and the hash table iterator ADT.
7. `identify.c`: Contains `main()` and the implementation of the author identification program.
8. `metric.h`: Defines the enumeration for the distance metrics and their respective names stored in an array of strings. Do not modify this.
9. `node.h`: Defines the interface for the node ADT. Do not modify this.
10. `node.c`: Contains the implementation of the node ADT.
11. `parser.h`: Defines the interface for the regex parsing module. Do not modify this.
12. `parser.c`: Contains the implementation of the regex parsing module.
13. `pq.h`: Defines the interface for the priority queue ADT. Do not modify this.
14. `pq.c`: Contains the implementation for the priority queue ADT.
15. `salts.h`: Defines the primary, secondary, and tertiary salts to be used in your Bloom filter implementation. Also defines the salt used by the hash table in your hash table implementation.
16. `speck.h`: Defines the interface for the hash function using the SPECK cipher. Do not modify this.
17. `speck.c`: Contains the implementation of the hash function using the SPECK cipher. Do not modify this.
18. `text.h`: Defines the interface for the text ADT. Do not modify this.
19. `text.c`: Contains the implementation for the text ADT.

You may have other source and header files, but *do not try to be overly clever*. You will also need to turn in the following:

1. `Makefile`: This is a file that will allow the grader to type `make` to compile your program.
    - `CC = clang` must be specified.
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be included.

- `make` should build the `identify` executable, as should `make all` and `make identify`.
- `make clean` must remove all files that are compiler generated.
- `make format` should format all your source code, including the header files.

2. Your code must pass `scan-build` *cleanly*. If there are any bugs or errors that are false positives, document them and explain why they are false positives in your `README.md`.

3. `README.md`: This must be in *Markdown*. This must describe how to use your program and `Makefile`. This includes listing and explaining the command-line options that your program accepts. Any false positives reported by `scan-build` should go here as well.

4. `DESIGN.pdf`: This *must* be a PDF. The design document should describe your design for your program with enough detail that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. For this program, pay extra attention to how you build each necessary component.

5. `WRITEUP.pdf`: This document *must* be a PDF. The writeup must discuss what you observe about your program's behavior as you tune the number of noise words that are filtered out and the amount of text that you feed your program. Does your program accurately identify the author for a small passage of text? What about a large passage of text? How do the different metrics (Euclidean, Manhattan, Cosine) compare against each other?

14 Submission

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

15 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 5 §5.7
  - Chapter 6
  - Chapter 7
- *Introduction to Algorithms* by T. Cormen, C. Leiserson, R. Rivest, & C. Stein
  - Chapter 11 (Hash tables and hash functions)
- *Introduction to the Theory of Computation* by M. Sipser
  - Chapter 1 §1.3 (Regular expressions)