1. What the Assignment Is About

The goal is to build a program that can identify the author of an anonymous "mystery" text. It does this by:

- Extracting **statistical fingerprints** (linguistic features) from a text.
- Comparing those fingerprints to a database of known authors.
- Identifying the author whose writing "fingerprint" most closely matches the mystery text.

---

2. How It Works (The Logic)

The program operates in four distinct phases:

Phase 1: Feature Extraction

The program calculates five specific values for any given text:

1. **Average Word Length:** How many characters are in their average word?
2. **Type-Token Ratio:** How diverse is their vocabulary? (Unique Words / Total Words).
3. **Hapax Legomana Ratio:** How often do they use "rare" words? (Words appearing only once / Total Words).
4. **Average Sentence Length:** Are they concise or do they write long, complex sentences?
5. **Sentence Complexity:** How many phrases (clauses separated by commas/colons) do they use per sentence?

Phase 2: Building a "Signature"

These five values are stored together as an **Author Signature**. Every author in our dataset has a signature file that serves as their benchmark.

Phase 3: The Similarity Measure

To compare two signatures (A and B), the program calculates a **Weighted Absolute Difference**. Each feature is assigned a specific "weight" based on how reliable it is at identifying an author.

- **The Math:** $\text{Score} = \sum(\text{Weight}_i \times |Feature_{A,i} - Feature_{B,i}|)$

  where $f_{i,x}$ is the value of feature i in signature x and w_i is the weight associated with feature i.

- **The Logic:** A lower score means higher similarity. A score of **0** would mean an identical writing style.

Phase 4: Prediction

The program iterates through every known author in the database, calculates the similarity score against the mystery text, and selects the author with the **lowest** score.

---

3. Data Set & Files

The data is split into two categories:

- **Signature Files (.stats):** Pre-calculated benchmarks for 13 famous authors (e.g., Jane Austen, Mark Twain, Shakespeare).
    - *Format:* Line 1 is the name; Lines 2–6 are the five numerical features.
- **Mystery Files (.txt):** Raw text files with unknown authors (e.g., mystery1.txt). These are the files our program will actually analyze.

---

4. Inputs and Outputs

Component	Description
User Input 1	A string representing the **filename** of the mystery text (e.g., "mystery3.txt").
User Input 2	A string representing the **directory path** where all the .stats files are stored.
Internal Data	The 13 signature files and the raw text content of the mystery file.
Program Output	The **name of the predicted author** (e.g., "The predicted author is Jane Austen").

---

5. How program works?

The program begins by asking the user for two strings: the first is the name of a file of text whose authorship is unknown (the mystery file) and the second is the name of a directory of files, each containing one linguistic signature.

The program calculates the linguistic signature for the mystery file and then calculates scores indicating how well the mystery file matches each signature file in the directory. The author from the signature file that best matches the mystery file is reported.

---

6. Some Definitions: What really is a sentence and phrases?

Before we go further, it will be helpful to agree on what we will call a sentence, a word and a phrase. Let's define a token to be a string that you get from calling the string method split on a line of the file. We define a word to be a non-empty token from the file that isn't completely made up of punctuation. You'll find the "words" in a file by using str.split to find the tokens and then removing the puctuation from the words using the clean_up function that we provided in find_author.cpp. If after calling clean_up the resulting word is an empty string, then it isn't considered a word. Notice that clean_up converts the word to lowercase. This means that once they have been cleaned up, the words yes, Yes and YES! will all be the same.

For the purposes of this assignment, we will consider a sentence to be a sequence of characters that (1) is terminated by (but doesn't include) the characters ! ? . or the end of the file, (2) excludes whitespace on either end, and (3) is not empty. Consider this file. Remember that a file is just a linear sequence of characters, even though it looks two dimensional. This file contains these characters:

this is the\nfirst sentence. Isn't\nit? Yes ! !! This \n\nlast bit :) is also a sentence, but \nwithout a terminator other than the end of the file\n

By our definition, there are four "sentences" in it:

Sentence 1	"this is the\nfirst sentence"

Sentence 2	"Isn't\nit"
Sentence 3	"Yes"
Sentence 4	"This \n\nlast bit :) is also a sentence, but \nwithout a terminator other than the end of the file"

Notice that:

- The sentences do not include their terminator character.
- The last sentence was not terminated by a character; it finishes with the end of the file.
- Sentences can span multiple lines of the file.
-

Phrases are defined as non-empty sections of sentences that are separated by colons, commas, or semi-colons. The sentence prior to this one has three phrases by our definition. This sentence right here only has one (because we don't separate phrases based on parentheses).

We realize that these are not the correct definitions for sentences, words or phrases but using them will make the assignment easier. More importantly, it will make your results match what we are expecting when we test your code. You may not "improve" these definitions or **your assignment will be marked as incorrect**.

---

7. Linguistic features we will calculate:

The first linguistic feature recorded in the signature is the **average word length**. This is simply the average number of characters per word, calculated after the punctuation has been stripped using the already-written clean_up function. In the sentence prior to this one, the average word length is 5.909. Notice that the comma and the final period are stripped but the hyphen inside "already-written" and the underscore in "clean_up" are both counted. That's fine. You must not change the clean_up function that does punctuation stripping.

Type-Token Ratio is the number of different words used in a text divided by the total number of words. It's a measure of how repetitive the vocabulary is. Again you must use the provided clean_up function so that "this","This","this," and "(this" are *not* counted as different words.

Hapax Legomana Ratio is similar to Type-Token Ratio in that it is a ratio using the total number of words as the denominator. The numerator for the Hapax Legomana Ratio is the number of words occurring exactly once in the text. In your code for this function you must not use a c++ dictionary (even if you have already learned about them on your own or are reading ahead in class) or any other technique that keeps a count of the frequency of each word in the text. Instead use this approach: As you read the file, keep two lists. The first contains all the words that have appeared at least once in the text and the second has all the words that have appeared at least twice in the text. Of course, the words on the second list must also appear on the first. Once you've read the whole text, you can use the two lists to calculate the number of words that appeared *exactly* once.

The fourth linguistic feature your code will calculate is the **average number of words per sentence.**

The final linguistic feature is a measure of **sentence complexity** and is the average number of phrases per sentence. We will find the phrases by taking each sentence, as defined above, and splitting it on any of colon, semi-colon or comma.

Since several features require the program to split a string on any of a set of different separators, it makes sense to write a helper function to do this task. To do this you will complete the function split_on_separators as described by the docstring in the code.

---

8. How to tackle this assignment?

This program is much larger than what you wrote for Assignment 1, so you'll need a good strategy for how to tackle it. Here is our suggestion.

Principles:

- To avoid getting overwhelmed, deal with one function at a time. Start with functions that don't call any other functions; this will allow you to test them right away. The steps listed below give you a reasonable order in which to write the functions.
- For each function that you write, plan test cases for that function and actually write the c++ code to implement those tests *before* you write the function. It is hard to have the discipline to do this, but you will have a huge advantage over your peers if you do.
- Keep in mind throughout that any function you have might be a useful helper for another function. Part of your marks will be for taking advantage of opportunities to call an existing function.
- As you write each function, begin by designing it in English, using only a few sentences. If your design is longer than that, shorten it by describing the steps at a higher level that leaves out some of the details. When you translate your design into c++, look for steps that are described at such a high level that they don't translate directly into c++. Design a

helper function for each of these, and put a call to the helpers into your code. Don't forget to write a great docstring for each helper!

Steps:

Here is a good order in which to solve the pieces of this assignment.

1. Read this handout thoroughly and carefully, making sure you understand everything in it, particularly the different linguistic features.
2. Read the starter code to get an overview of what you will be writing. It is not necessary at this point to understand every detail of the functions we have provided in order to make progress on this assignment.
3. Create a module called function_tester.cpp and have it import your find_author module. Add a main block to function_tester: that is where you can put your code that tests your individual functions. For now, it can just say pass.
4. Plan testing for, write, and test function get_valid_filename. In order to determine whether a file exists or not, use the function os.path.exists. Give it a string that is the name of a file, and os.path.exists will return a boolean indicating whether or not the file exists. The help for this function talks about giving it a path (which is a description of the file and where it exists among all your folders), but if you give it a string that is simply a file name, it will check in the folder in which your code is running.
5. Plan testing for, write, and test function read_directory_name. Use the function os.path.isdir to check if a string is a valid directory.
6. Plan testing for, write, and test functions for the first three linguistic features: average_word_length, type_token_ratio, and hapax_legomana_ratio.
7. Write, and test function split_on_separators. Test this function carefully on its own before trying to use it in your other functions.
8. Write and test function average_sentence_length. Begin by writing code to extract the sentences from the file. Test that this part of your code works correctly before you worry about calculating the average length.
9. Complete avg_sentence_complexity and finally compare_signatures.
10. Now you have implemented all the functions. Run our a2_self_test module to confirm that your code passes our most basic tests. Correct any errors that this uncovers. But of course if you did a great job of testing your functions as you wrote them, you won't find any new errors now.
11. You are now ready to run the full author detection program: run find_author. To do this you will need to set up a folder that contains **only** valid linguistic signature files. We are providing a set of these on the data files page You'll also want some mystery text files to analyze. We have put a number of these on the data files page. If you are copying them to your home machine, don't put them in the same folder as the linguistic signature files.

9. How to Test your code?

We are providing a module called a2_self_test that imports your find_author and checks that most of the required functions satisfy some of the basic requirements. It does not check the two functions that involve input and output with the user: get_valid_filename and read_directory_name.

When you run a2_self_test.cpp, it should produce no errors and its output should consist only of one thing: the word "okay". If there is any other output at all, or if any input is required from the user, then your code is not following the assignment specifications correctly and **will be marked as incorrect**. Go back and fix the error(s).

While you are playing with your program, you may want to use the signature files and mystery text files we have provided. This in conjunction with a2_self_test is still **not** sufficient testing to thoroughly test all of your functions under all possible conditions. With the functions on this assignment, there are many more possible cases to test (and cases where your code could go wrong). If you want to get a great mark on the correctness of your functions, do a great job of testing them under all possible conditions. Then we won't be able to find any errors that you haven't already fixed!

10. How does the Marking of the assignment work?

These are the aspects of your work that we will focus on in the marking:

- **Correctness:** Your code should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks.
- **Docstrings:** For each function that you design from scratch, write a good docstring. (Do not change the docstrings that we have already written for you.) Make sure that you read the Assignment rules page for some important rules and guidelines about docstrings.
- **Internal comments:** Within functions, the more complicated parts of your code should also be described using "internal" comments. For this assignment, internal comments will be more important than on assignment 1.
- **Programming style:** Your variable names should be meaningful and your code as simple and clear as possible.
- **Good use of helper functions:** If you find yourself repeating a task, you should add a helper function and call that function instead of duplicating the code. And if a function is more than about 20 lines long, consider introducing helper functions to do some of the work -- even if they will only be called once.
- **Formatting style:** Make sure that you read the Assignment rules page for some important rules and guidelines about formatting your code.

Submitting your assignment

This assignment will be completed on your own without a partner.
You must hand in your work electronically, using the MarkUs online system. Instructions for doing so are posted on the Assignments page of the course website.

For this assignment, hand in just one file:

- find_author.cpp

Once you have submitted, be sure to check that you have submitted the correct version; new or missing files will not be accepted after the due date. Remember that spelling of filenames, including case, counts. **If your file is not named exactly as above, your code will receive zero for correctness.**