Project Report of Text Analyzer

Introduction to Programming – November 2025

By Filippos Zaravelas and Matteo Becatti

Introduction

This project was developed by Filippos Zaravelas and Matteo Becatti, using different tools, like PyCharm as the IDE, since it has a great Live Coding feature that facilitates collaborative coding, and Git as the Version Control System.
The code is also fully open source and publicly available on GitHub at the following link: https://github.com/LNU-Programming/Text-Analysis

Once the project was finished, we moved everything to JupyterHub, to meet the delivery requirements, tested every functionality, and then packaged the whole project as per instructions.

For this project, we are aiming for an A ++ or A #.

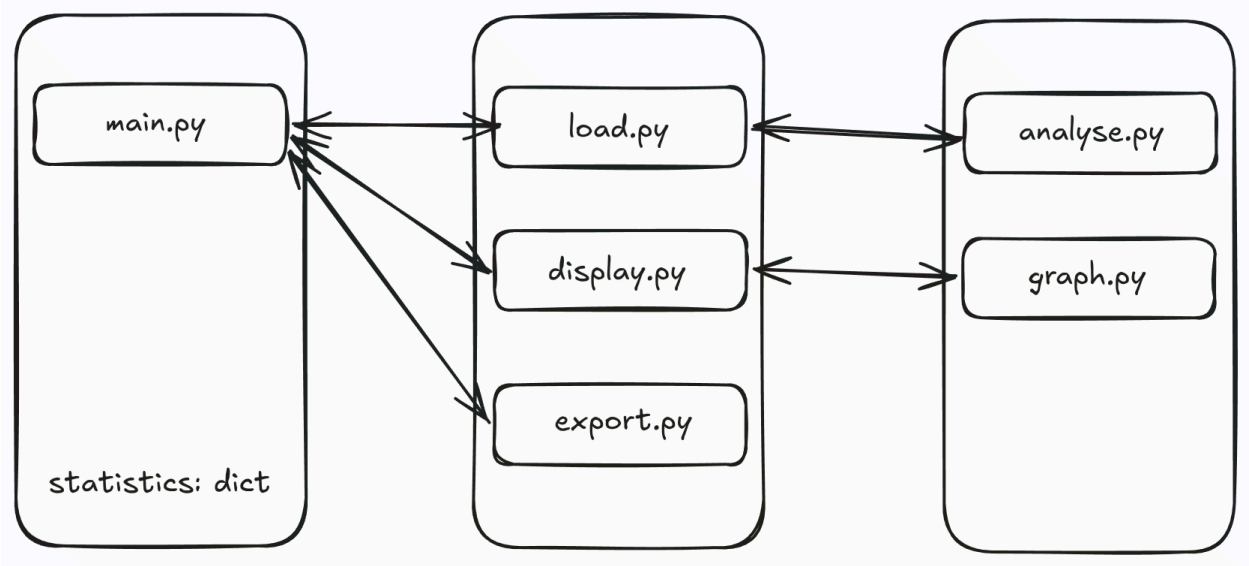
We didn't actively keep track of the time spent on developing the project. Using a tool called `git-hours`, which analyzes the full commit history, we managed to get an estimate, and we worked on it for roughly 74 hours.

Implementation

Our idea

Before starting to write the code, we sat down for quite a while, and tried to come up with an efficient and easy to follow architecture for our codebase. We focused on readability and ease of maintenance before anything else.

After some iterations, we ended up with the following:



Given the constraints, we decided that the best way to save the results was to a struct-like structure, that can be passed around every module.
Structs don't exist in Python, so we ended up with a dictionary called `statistics` .
This dictionary is passed between all the modules, so that they can easily read the data from it, and generate outputs, whether they are in the console, a file, or a graph.

The file structure is shown on the side.

```
NGDPV_en_13
├── data
│   ├── All Shakespear Plays.txt
│   ├── Atheist Manifesto.txt
│   ├── Carmilla.txt
│   ├── Communist Manifesto.txt
│   ├── Dracula.txt
│   ├── Moby Dick.txt
│   ├── Pride and Prejudice.txt
│   └── War and Peace.txt
├── exports
│   └── Dracula.txt_results.txt
├── __init__.py
├── main.ipynb
└── src
    ├── analyse.py
    ├── display.py
    ├── export.py
    ├── graph.py
    ├── __init__.py
    └── load.py
```

How the program works

Let's simulate a clean run of the code, to explore its inner workings. It's useful to look at the above drawing to keep track of where we are.

We start in `main` , which only handles the menu system. From here, we can call the other modules, depending on the choice made via input. Let's start with "1) Load a basic text file".

This leads us to `load` . This module handles the listing of the available files in the `data/` directory, and lets the user choose which one they want to select. Once the choice is made, the filename is sent to the `analyse` module.

Here is where all the calculations happen. This module is divided into several functions. The core loop is in `analyse_file()`: we read a file line by line, and we process each line one at a time. The line processing happens in `process_line()`. This function increments the number of lines in the code, and then calls `process_character()`. We can start seeing a certain pattern with the names. We tried to make the code as readable as possible, by giving meaningful names to each function, variable and constant, and writing it in a way that makes commenting almost redundant.

The character processing is where we check for a lot of properties, but most importantly, we check if we are at the end of a sentence and if we are at the end of a word.
Here it is important to make a distinction: for words, we check if the current character is alphabetic, or an apostrophe. The apostrophe is important, to include words like "don't" and "I'm" which are considered single words. If the check passes, we add the character to the buffer `current_word`, otherwise, we need to call `finalize_current_word()`.

In this function, we take the data from the word and get the necessary statistics, like checking if it's the longest or shortest, its uniqueness and so on.

Going back to `process_character()`, we also check if we are at the end of a sentence, and if we are, we calculate the relevant statistics for sentences.

This process is repeated for each line, until we exit the loop. Now we have a problem. The process relies on sentences ending with some type of punctuation, as they usually do. But sometimes, authors don't put punctuation in the very last sentence of their books (for instance, in Dracula). This means that we are left with elements in our buffer that have not been processed. We solve this problem by calling `finalize_remaining_data()`, which executes the same steps as the main cycle, but only if there are elements left in the buffers.

Once this is done, we call `calculate_final_statistics()` to compute all the data that are being used for the graphs and for the displayment in the terminal. Finally, we output a success message, and we return `statistics: dict`.

The other menus don’t contain that much logic. Menu 2 to 5 calls the `display` module, that handles the outputs, and it has a lot of print statements to show all our data. The module is also divided in functions, to ensure readability, reusability and maintainability. At the end of the console output of each analysis type, we also call the respective graph outputs, which are displayed by calling the necessary function from the `graph` module. This module contains a function for each type of graph, and using Matplotlib displays the results in a graphical output. The final menu handles the exporting results of the file. Also in here there’s not much logic to be taken into consideration, the file is divided into functions, one for each type of analysis, and they all contain a long list of write statements to output the data to a file in the `exports/` directory.

Error handling

The program should be able to handle all the errors that may arise during its execution. From the user typing wrong input, to the IO operations with files.

Things to take away

This project taught us many things, teamwork being one of them. By working together, we learned how important it is to communicate clearly and to make sure we both understand what needs to be done. We saw how sharing ideas and listening to each other helps us avoid mistakes and keep the project moving forward.

Another big takeaway was experiencing what a real-life project can look like. We had to plan how to divide the tasks and stay organized just like in a real job. The tools we used, such as Git and PyCharm, made our work easier and helped us stay on track.

Overall, this project showed us the value of teamwork, communication, and staying organized. These skills will help us in future projects and in pursuing our careers.