CSC 510 SOFTWARE ENGINEERING PROJECT REPORT

CELT 2.0: SENTIMENT ANALYZER

SOFTWARE CODE DOCUMENTATION

TEAM MEMBERS

Setu Kumar Basak (sbasak4)

Conor Thomason (cjthom24)

Keertana Vellanki (kvellan)

Muntasir Hoq (mhoq)

Matthew Sohacki (mjsohack)

INTRODUCTION

Sentiment analysis is among the most rapidly increasing research areas in computer science. It is usually quite difficult to keep up with all of the development taking place in the area. By analyzing the data provided in different methods, we expect to achieve our goal of accurately predicting a user's sentiment in our project. Although it is still in its early phases of development, this project has the potential for application to a variety of sectors that could be beneficial to society. This report offers users a broad overview of the project, allowing individuals to understand it as open-source software and add improvements. The report also helps developers in comprehending the code and serves as a starting point for the project.

Potential Users:

1. Customer review analyzers
2. Online shopping analytics
3. Online shopping manager
4. Movie critics

ARCHITECTURE

The following technologies were used to complete the development, and it is recommended that the next group of developers who take on this project have these technologies installed and running before proceeding:

1. FrontEnd:
    a. HTML
    b. CSS
    c. Javascript

2. API:
    a. ASP.NET Core Web API 3.1
    b. Nunit and Moq (Test Framework)
3. Server:
    a. Flask Server & Dockerfile for Containerization
    b. Pytest (Test Framework)
4. Model:
    a. Flair
    b. Torch

Although we have used HTML and CSS for the FrontEnd, the users can merge the backend logic with any of the front-end frameworks they wish to use such as React, angularJS, etc.

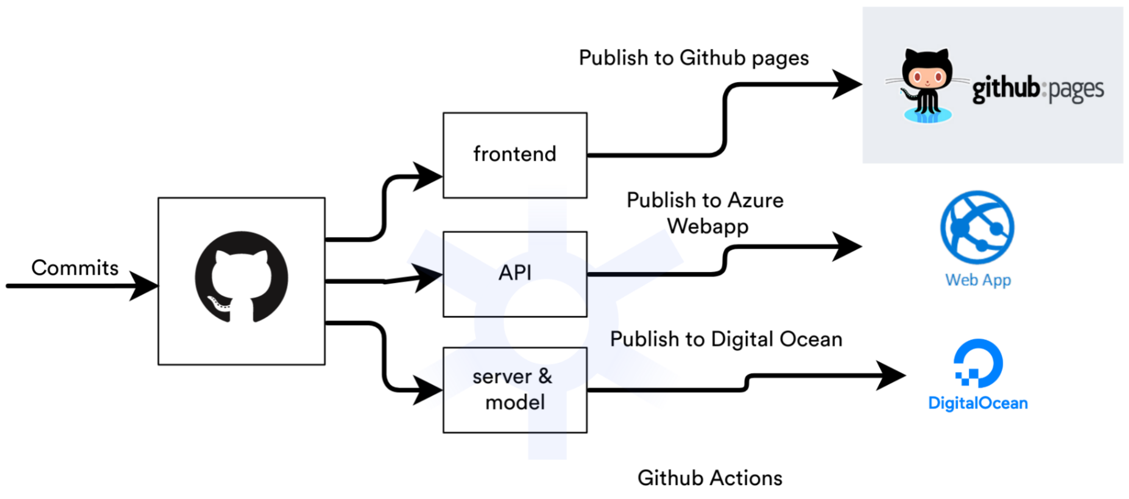


Fig: Architecture of CELT2.0

In our repo, we have separate directories of frontend, API, server, and model.

Frontend: If any code is committed into the frontend directory, then the frontend gets published to GitHub pages using the Github actions. Our published frontend URL: https://team-glare.github.io/CELT2/

API: If any code is committed into the API directory, then the API gets published to Azure Webapp using the Github actions. Our published API URL: https://celtapi.azurewebsites.net/

Model & Server: If any code is committed into the model and server directory, then the model & server gets published to Digital Ocean using the Github actions.

The advantages of CELT2.0 architecture:

1. Easily maintainable.
2. Can be easily swapped out any submodule with a different module.
3. Each sub-module can be tested separately.
4. Each sub-module change can be published separately.

CODE FUNCTIONALITIES

Server:

App.py: This is the file that generates the Flask server - upon calling “flask run” within this directory, this is the file that will be called. It includes a sentence tokenizer for future development for multiple sentence sentiment analysis, and defines the routes currently required.

- **/sentiment/text** - this is the route defined for passing a POST request containing a JSON body. The body is expected to have the K/V pair of “sentimentText”: “<YOUR TEXT HERE>”. This JSON is parsed and passed to the model method classify()

Defining additional routes can be done with the decorator @app.route(“URL ROUTE HERE”, methods = [‘REQUEST’, ‘TYPES’, ‘HERE’].

The server itself is relatively straightforward - run a server that has routes defined for HTTP calls. However, we wanted to focus our attention on the architecture itself - in this case, containerization.

Dockerfile: This Dockerfile defines everything required for hosting a Flask server with our desired capabilities. It runs off the python:3.9-slim-bullseye Docker repository image, and we adjust it to fit our needs. We copy both the model and the server files into the image, as both are hosted on this image. We set the arg DEBIAN_FRONTEND to “noninteractive” so that any installs that usually require user input default to “y” (yes), and we also add the “-y” tag to any future apt calls, just to be on the safe side.

The flair dependencies require the installation of gcc and g++, something that doesn’t come preinstalled on slim images. We use apt to install these manually.

Next, we run pip through the requirements files defined in the model and server subdirectories - it is crucial to note that we need the LTS cpu-only version of pytorch for the model, so we use the -f tag to specify a download location.

The final lines then call the app.py file with python.

Shell scripts: For the sake of ease of development, multiple shell scripts were created and used while building the server. A quick summary of them all will be shown, their primary use is for local testing, debugging, and setup.

- **Boot.sh:** Run a Docker image locally, provided that you have Docker installed. Note that there are various issues with networking between WSL/2 and a local machine, so adjustments may need to be made accordingly.
- **Boot_dockerless.sh**: Run the server without requiring a Docker image to be used to start a container. This means that we need to install all of the requirements to be certain they are prepared, and then run the flask server. This script *must* be run within the context of the server subdirectory.
- **Build.sh:** Forces a manual build of a Docker image locally. Useful for attempting to locally debug Docker images, but can take quite some time given the requirements. Possibility for improvement here.
- **Debug.sh:** Run a Docker image already created, but instead of running the server, you are able to navigate the files inside. Useful for figuring out file structures, as well as the WORKDIR and COPY commands.
- **Removeall.sh:** This will end any Docker containers running locally, and then delete the containers - the images remain unaffected. Useful for keeping resource usage minimal.
- **Stopall.sh:** This will end any Docker containers running locally, but will not delete their corresponding containers - good for forcing restarts.

Gunicorn_config.py: We are required to use GUnicorn to provide an external IP address that Digital Ocean can bind to. This file provides the configuration necessary to do so - nothing particularly special, just some boilerplate code.

Test_server.py: The tests for the server. Tests for the server status, as well as variations of sentiment that may be provided, comparing them to the expected result. These results are sent to the codecov.

All requirements for the server specifically can be found in requirements.txt. We recommend using **setup.sh** in the root directory for all requirements.

Model:

model.py: For the sentiment analysis of texts, we have used flair to implement the model. The reason Flair is exciting news for NLP is because a recent paper Contextual String Embeddings for Sequence Labelling from Zalando Research covers an approach that consistently outperforms previous state-of-the-art solutions. It's implemented and fully supported in Flair and can be used to build text classifiers. The model is built in python. The model outputs the sentiment of each sentence passed to the model as a command-line argument. The output is in this format: Sentiment: (Probability). This model classifies each sentence as either positive or negative. So this is a binary classification. To install Flair you will need Python 3.6. The requirements to run the model are:

flair==0.9

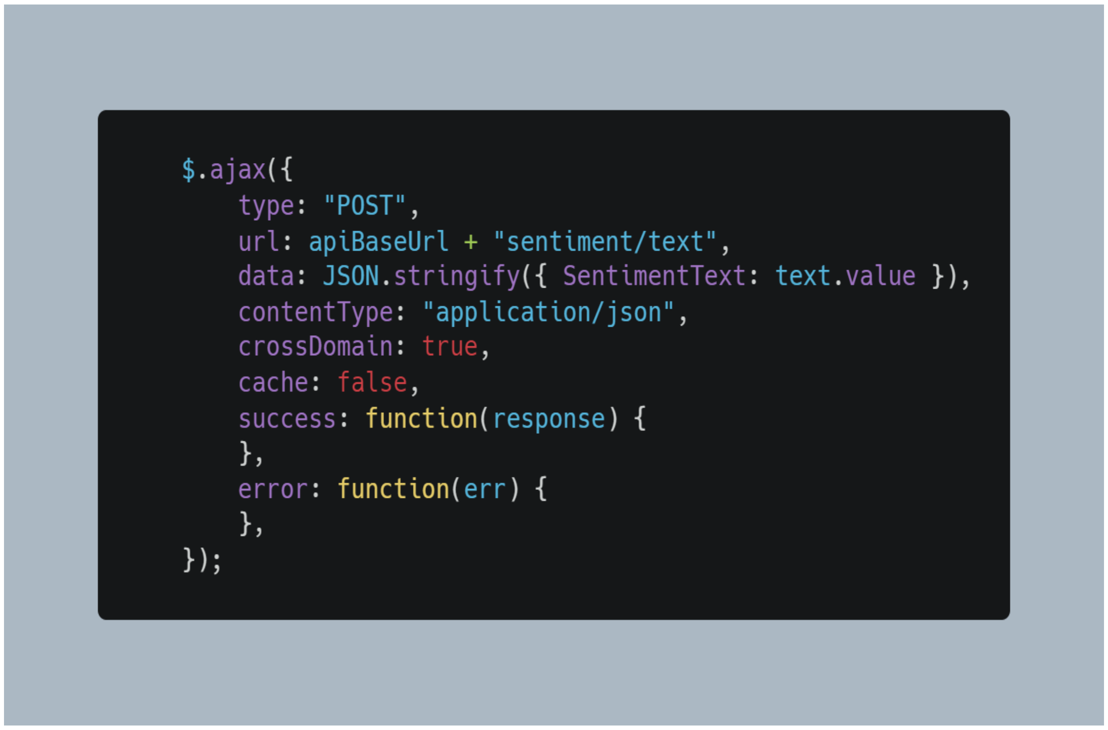
torch==1.8.2+cpu torchvision==0.9.2+cpu torchaudio===0.8.2

API:

For API, the project is structured into the below subdirectories.

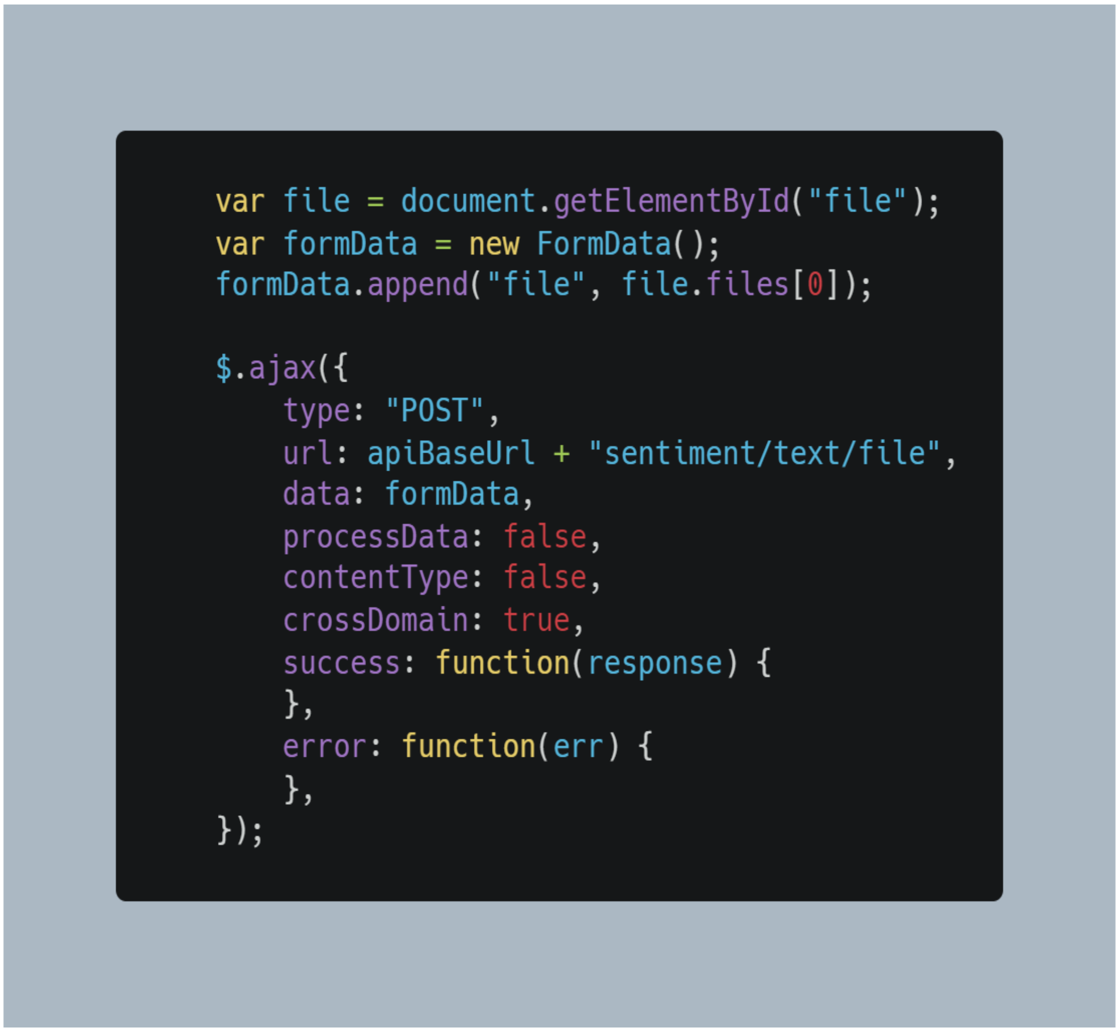
1. Controllers:
    a. *SentimentController.cs:* Defines all the endpoints for sentiment analyzer.
        i. *public SentimentController(ISentimentService sentimentService)*: Constructor for SentimentController with the injected dependencies.
            1. Inputs:
                a. *ISentimentService sentimentService*: The ISentimentService for calculating the sentiments.
        ii. *public async Task<string> GenerateSentimentFromText(TextInput input):* HttpPost endpoint for calculating sentiment from plain text.
            1. Route: */sentiment/text*
            2. Inputs:
                a. *TextInput input*: The plain text input.

3. Outputs:
    a. *Task<string>*: Returns the calculated sentiment label with cumulative probability.



```
$.ajax({
    type: "POST",
    url: apiBaseUrl + "sentiment/text",
    data: JSON.stringify({ SentimentText: text.value }),
    contentType: "application/json",
    crossDomain: true,
    cache: false,
    success: function(response) {
    },
    error: function(err) {
    },
});
```

iii. *public async Task<string> GenerateSentimentFromFile(IFormFile file):* HttpPost endpoint for calculating sentiment from text file.
1. Route: */sentiment/text/file*
2. Inputs:
    a. *IFormFile file:* The input text file.
3. Outputs:
    a. *Task<string>*: Returns the calculated sentiment label with cumulative probability.



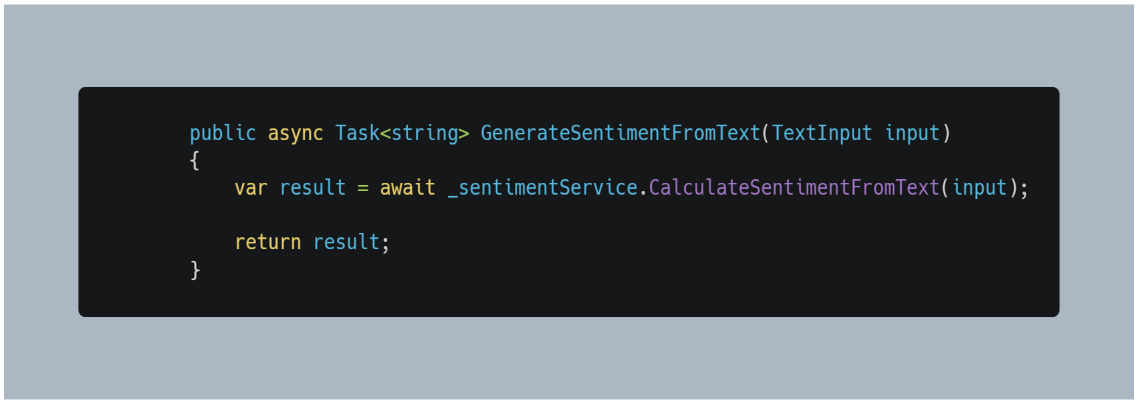
```
var file = document.getElementById("file");
var formData = new FormData();
formData.append("file", file.files[0]);

$.ajax({
    type: "POST",
    url: apiBaseUrl + "sentiment/text/file",
    data: formData,
    processData: false,
    contentType: false,
    crossDomain: true,
    success: function(response) {
    },
    error: function(err) {
    },
});
```

2. Services:
    a. *ISentimentService.cs*: Interface for sentiment service which calculates sentiments from different media.
        i. *Task<string> CalculateSentimentFromText(TextInput input)*: Interface method for calculating sentiment from plain text.
            1. Inputs:
                a. *TextInput input*: The plain text input.
            2. Outputs:
                a. *Task<string>*: Returns the calculated sentiment label with cumulative probability.
        ii. *Task<string> CalculateSentimentFromTextFile(IFormFile file)*: Interface method for calculating sentiment from text file.

        1. Inputs:
            a. *IFormFile file:* The input text file.
        2. Outputs:
            a. *Task<string>*: Returns the calculated sentiment label with cumulative probability.

b. *SentimentService.cs*: Defines the service for calculating the sentiment.
    i. *public SentimentService(IStreamReader reader, IServerClient serverClient)*: Constructor for SentimentService with the injected dependencies.
        1. Inputs:
            a. *IStreamReader reader*: The IStreamReader for reading streams for files.
            b. *IServerClient serverClient*: The IServerClient for the connection to the server.
    ii. *public async Task<string> CalculateSentimentFromText(TextInput input)*: Calculates the sentiment from plain text.
        1. Inputs:
            a. *TextInput input*: The plain text input.
        2. Outputs:
            a. *Task<string>*: The calculated sentiment label with cumulative probability.

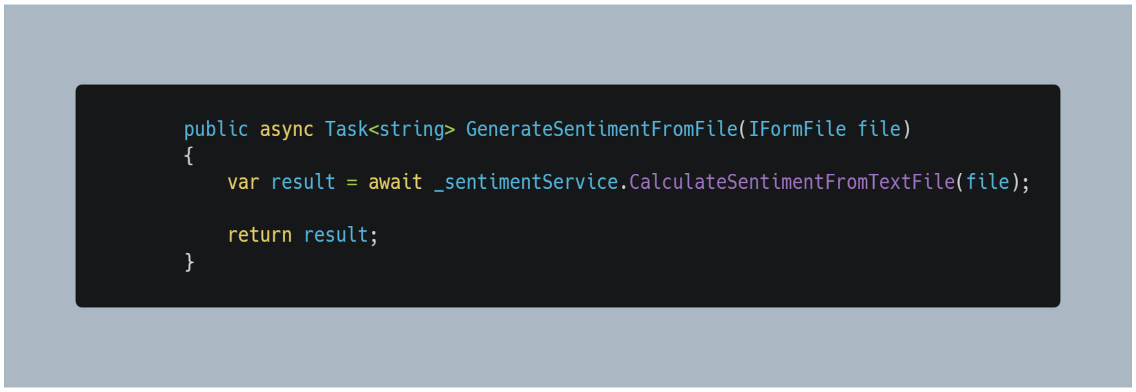


```
public async Task<string> GenerateSentimentFromText(TextInput input)
{
    var result = await _sentimentService.CalculateSentimentFromText(input);

    return result;
}
```

    iii. *public async Task<string> CalculateSentimentFromTextFile(IFormFile file)*: Calculates the sentiment from text file.

1. Inputs:
    a. *IFormFile file*: The input text file.
2. Outputs:
    a. *Task<string>*: The calculated sentiment label with cumulative probability.



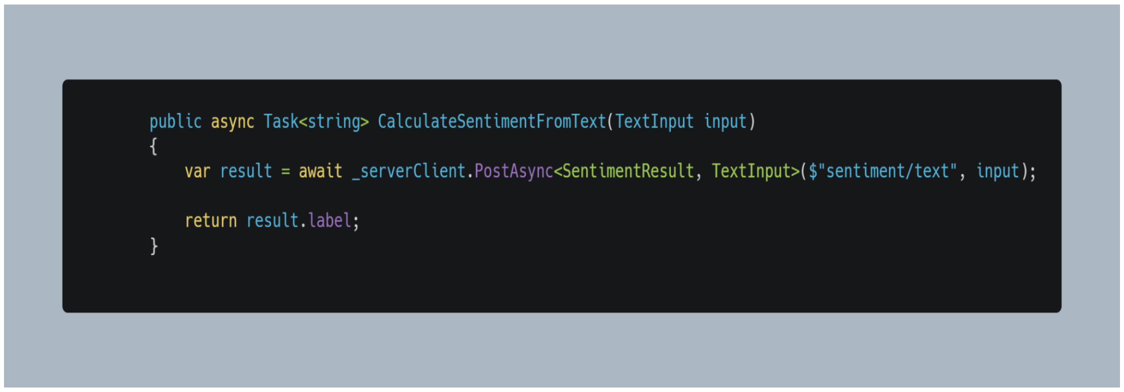
```
public async Task<string> GenerateSentimentFromFile(IFormFile file)
{
    var result = await _sentimentService.CalculateSentimentFromTextFile(file);

    return result;
}
```

3. Models:
    a. *SentimentResult.cs*: Dto for sentiment result.
        i. *public string label { get; set; }*: The cumulative result label getter/setter method.
    b. *TextInput.cs*: Model for text input.
        i. *public string sentimentText { get; set; }*: The plain text getter/setter method.
4. Utilities:
    a. *AppSettings.cs*: Defines the app settings.
        i. *public string ServerBaseURL { get; set; }*: Defines the server base URL getter/setter method.
    b. *IStreamReader.cs*: Defines the interface for stream reader.
        i. *StreamReader GetReader(Stream inputStream)*: Defines the function of returning a stream reader with a stream as input.
    c. *FileStreamReader.cs*: Defines File stream reader which implements IStreamReader.
        i. *public StreamReader GetReader(Stream inputStream)*: Method for getting a StreamReader.
            1. Inputs:

            a. *Stream inputStream*: Stream input.
        2. Outputs:
            a. *StreamReader*: Return an instance of StreamReader with the inputted Stream.

d. *IServerClient.cs*: Defines the interface for server clients.
    i. *Task<TResult> GetAsync<TResult>(string relativeUrl)*:
        1. Types:
            a. *TResult*: Defines the return result type.
        2. Input:
            a. *string relativeUrl*: Relative URL input.
    ii. *Task<TResult> PostAsync<TResult, TPayload>(string relativeUrl, TPayload payload)*
        1. Types:
            a. *TResult*: Defines the return result type.
            b. *TPayload*: Defines the payload type.
        2. Inputs:
            a. *string relativeUrl*: Relative URL input.
            b. *TPayload payload*: Input payload.



```
public async Task<string> CalculateSentimentFromText(TextInput input)
{
    var result = await _serverClient.PostAsync<SentimentResult, TextInput>($"sentiment/text", input);

    return result.label;
}
```

FUNCTIONAL TESTING

API:

For functional testing of the API, we have a separate test project CELTAPI.Tests in the celt2/API directory, and included Nunit and Moq framework.

Sentiment Service Test 1 (CalculateSentimentFromTextEquality): In this test, we have provided a positive text input and checked the equality of the positive result by mocking the output.

Sentiment Service Test 2 (CalculateSentimentFromTextInEquality): In this test, we have provided a positive text input and checked the inequality with a negative result by mocking the output.

Sentiment Service Test 3 (CalculateSentimentFromTextFileEquality): In this test, we have mocked a text file stream and checked the equality with a positive result by mocking the output.

Sentiment Service Test 4 (CalculateSentimentFromTextFileInEquality): In this test, we have mocked a text file stream and checked the inequality with a negative result by mocking the output.

FUTURE SCOPE

1. Add sentiment analysis functionality from the audio file.
2. Create a separate DEV server for code integration.
3. Add integration tests.
4. Add sentiment comparison functionality.
5. Optionally show line by line analysis.
6. Implement style and code checkers
7. Allow server/model to analyze larger amounts of data.