CSE 546 — SpeechMiner

Nishanth Bangalore Murali
Adersh Ganesh
Karthik Nishant Sekar
Jose Elenes

1. Introduction

In this project, we aim to build a web-application that analyses the speeches, which can be an audio file or a transcript, given as input by the users. The analyses include:
1. Topic modelling: This tells us about the topics the speaker has talked about.
2. Sentiment analysis: We can know how positive/negative the sentiments of the speaker are.

2. Background

Describe the background of the problem.

- **Technologies relevant to the project**
  1. Google Cloud APIs like 'speech', 'translate' and 'pubsub' to handle the work of our application.
  2. Python libraries like sklearn, Pandas and nltk that are used to build the NLP model

- **Importance of the problem/application:**

  This application can be used in/for;

  1. Recommendation Engines: The NLP model can be used to recommend similar speakers to the users.
  2. Political speech analysis: The application can be used by media houses to analyze the speeches given by politicians.

- **Why the existing solutions are not sufficient?**

  There are no topic modeling applications in the market yet, and our model has an accuracy rate of 75%, which is more than the current state of the art rate of 71%.

  Other speech analyzing applications do not translate transcripts from languages other than English, but our project supports 19 different languages.

  This application can be used in/for;

  1. Recommendation Engines: The NLP model can be used to recommend similar speakers to the users.
  2. Political speech analysis: The application can be used by media houses to analyze the speeches given by politicians.

3. Design and Implementation

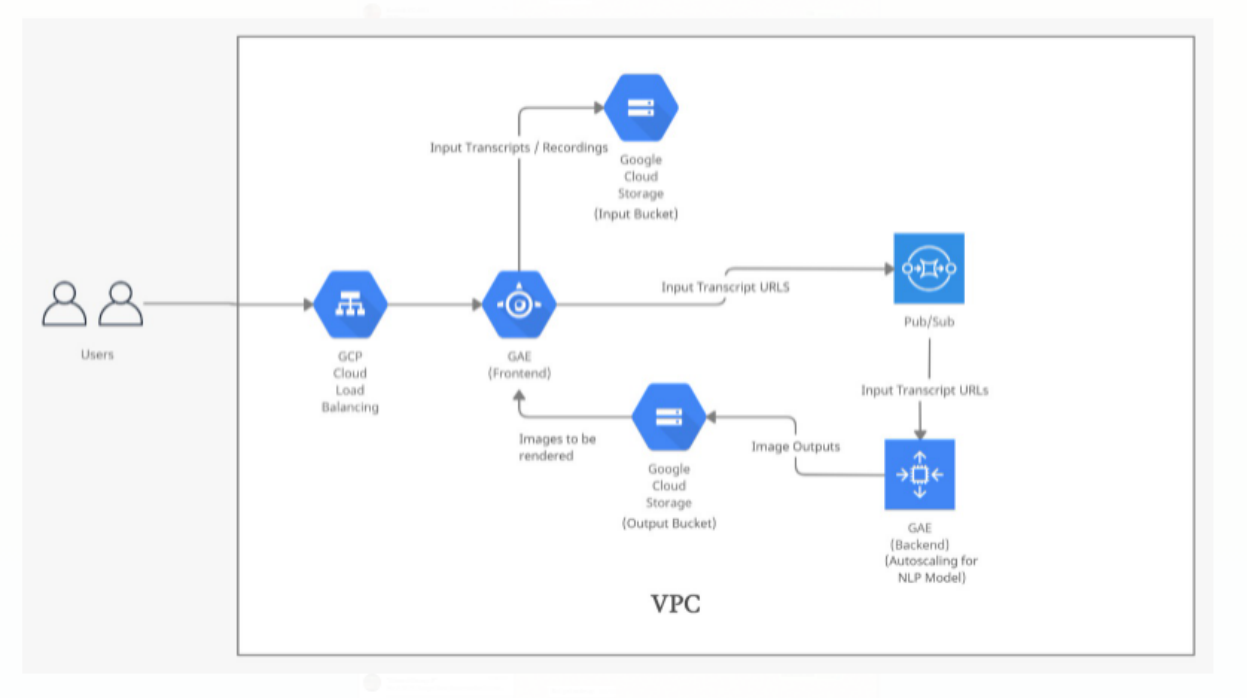


Fig: System architecture of the cloud application

Following are the components of Google Cloud, used in our project:

Google Load Balancer: This component works as a traffic filter which will be the only component visible to the public, for sending HTTP requests.

Google App Engine (GAE):

1. Front-end: A Flask application is run on the GAE, which together forms a Virtual Machine, which will be the UI to take in inputs from the user and give the outputs back accordingly.
2. Back-end: Again, a Flask server that runs as a listener on another GAE, which will lift all the workload of processing the inputs using the Natural Language Processing (NLP) model. These instances will be auto-scaled, according to the number of inputs given by the user.

Google Pub/Sub:

The front-end server sends the URLs of the transcripts, that are stored in the google cloud storage, to the Publisher. These URLs are subscribed by the Subscriber at the back-end GAE that would start running the model for the given transcript (URL). This would also help in decoupling the front-end from the back-end, so that asynchrony is maintained.

Google Cloud storage:

Input storage:

The inputs from the user will be uploaded to a google cloud storage's bucket and the blob's URL will be published to the publisher.

Output storage:

The images obtained by running the NLP model on the GAE will be stored in another bucket, in order to achieve persistent storage. These images will be picked up by the front-end GAE, which will render these images on the UI, to be viewed by the user.

VPC:

It acts like a network confinement for the application, so that only the authorized users can access the application and thus making it secure.

CLOUD SERVICES (APIs) USED:

Cloud storage:

This API will be used to create, access and delete user inputs and the outputs that are to be sent back to the UI. The 'gsutil' URLs of these objects will be used to engage the Pub/Sub API, so that the front-end and the back-end GAEs are decoupled.

Google Pub/Sub:

The Pub/Sub API is used as the bridge between the app-server and the web-server. The input images' 'gsutil' URLs will be used to publish to the Publisher and they will be subscribed by the Subscriber. This is done to ensure that both the servers (GAEs) used in the application work in an asynchronous fashion and are also decouple to enhance the performance of the system as a whole.

Google Cloud Speech:

This handy API is used to recognize the speech that will be recorded live by speaker/user. The output obtained from the API will be a transcript text, which will be analyzed by the NLP model in the back-end.

Google Cloud Translate:

This is used to translate the transcripts in different languages, given by the user, to English. These will also be fed to the NLP model in order to be processed and analyzed.

Role of Google App Engine (GAE):

Front-end:

For the front-end of our web-application, we use a Flask app-server that would take in inputs from the user and give the outputs back to them after the input transcripts/recordings are processed and analyzed. The App engine will ping the output bucket for every 10 seconds for each user, so that the outputs are rendered as soon as they are produced by the model.

Back-end:

The Flask application at the back-end has python code that runs the NLP model. It will be used to process the input and analyse the transcripts and send the output to the output bucket. This GAE will be auto-scaled based on the CPU utilization of the VM.

Autoscaling:

In our project, the bottleneck is the backend server that runs the NLP model. This is the case because of the load on the vCPU while pre-processing the transcripts and predicting the topic that the speaker speaks. This bottleneck is handled automatically by the GAE, due to the configuration, which scales the VMs up when the vCPU utilization is more than 80%. It also scales down when there are no requests for the VM to process.

Solution to the problem proposed:

Our application has the following features that address the solution to the problem discussed earlier;

1. UI to take speech transcripts or audio files from the user.
2. NLP model that pre-processes and analyses the speech:
   - It uses the subjectivity and polarity metrics of a speech and gives a sense of the sentiments of the speaker. It also uses a model called LDA (Latent Dirichlet Allocation), which has an accuracy rate of 75%, to give the right predictions about the topic the speaker speaks about.
3. Persistent storage that stores the results of the model.
4. Dashboard in the UI, which will reflect the output of the model to the user.

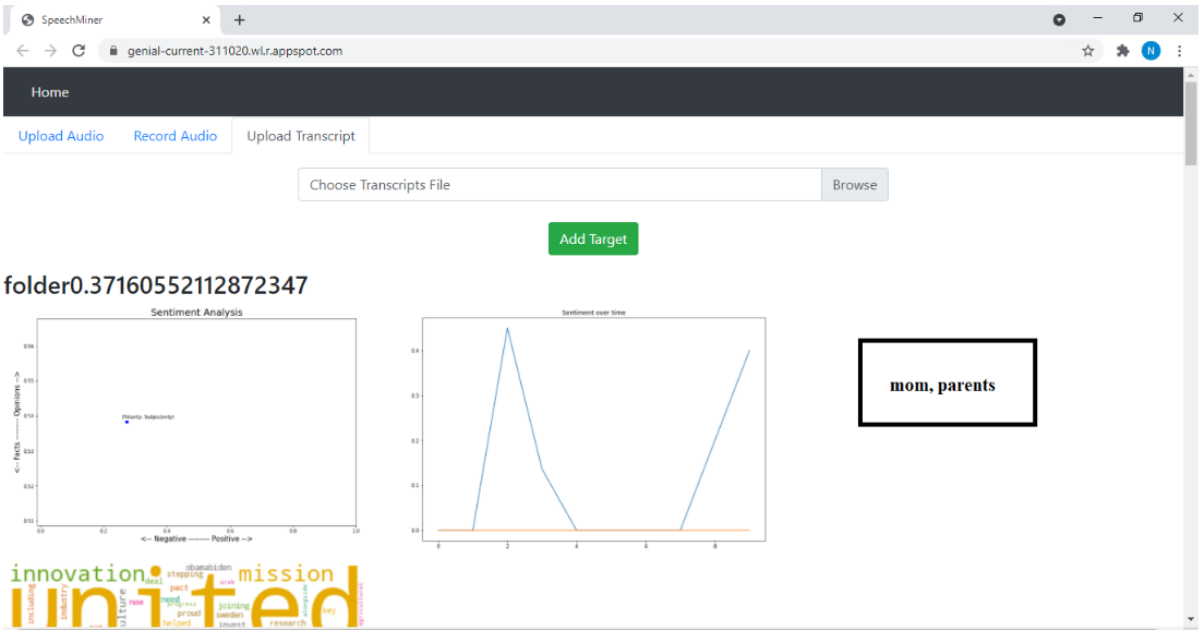
Why is our solution significantly better than SOTA (State-of-the-art)?

The state-of-the-art solution (SOTA) for topic modelling is 71%, whereas our model has an accuracy rate of 75%, which is 4% greater than the SOTA model. Given the challenges in handling unstructured data and that of finding a pattern or useful information from the same, we can say that our model performs significantly better than the SOTA model. Also, performance-wise, as we deploy the model on GAEs, we can leverage the advantages of autoscaling our application as well.

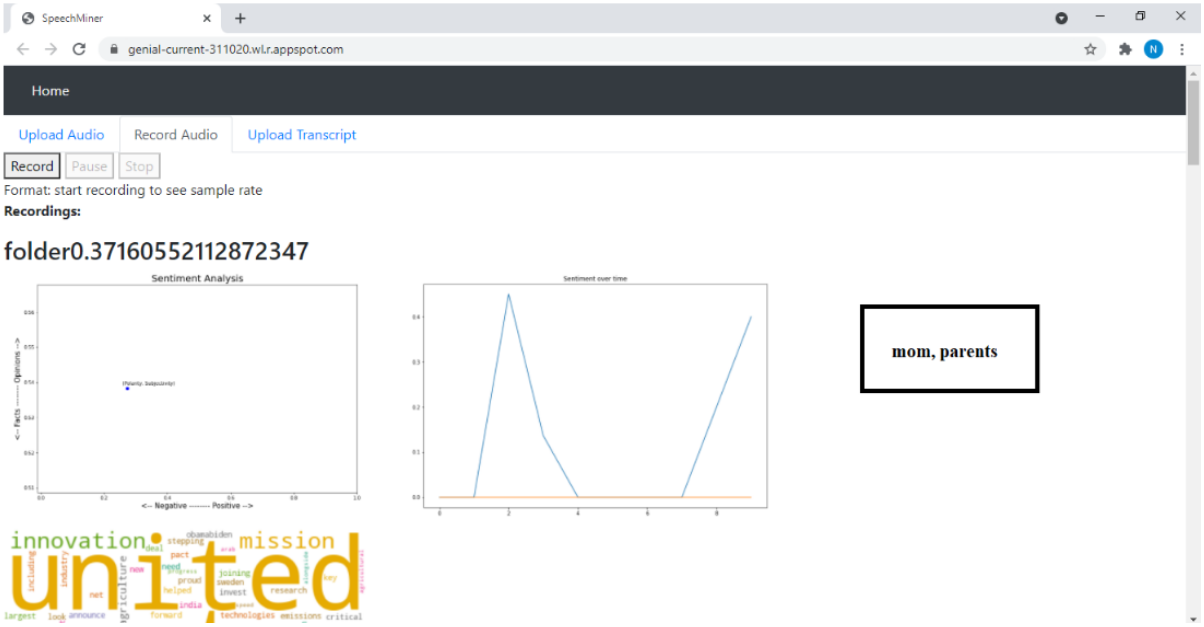
4. Testing and evaluation

Testing functionality:

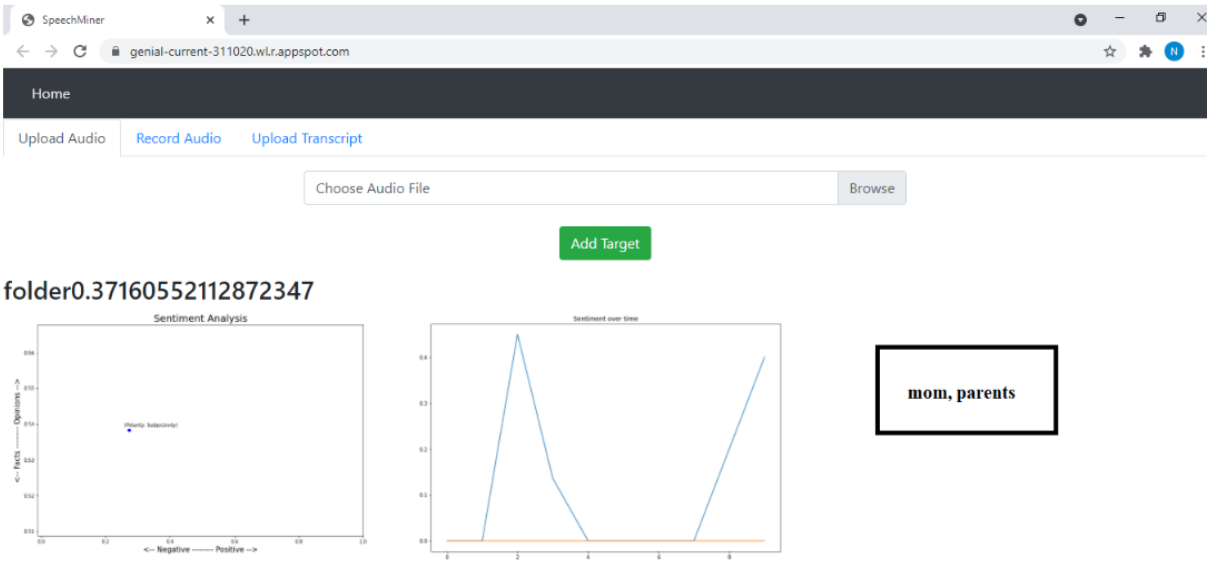
1. Input transcripts:



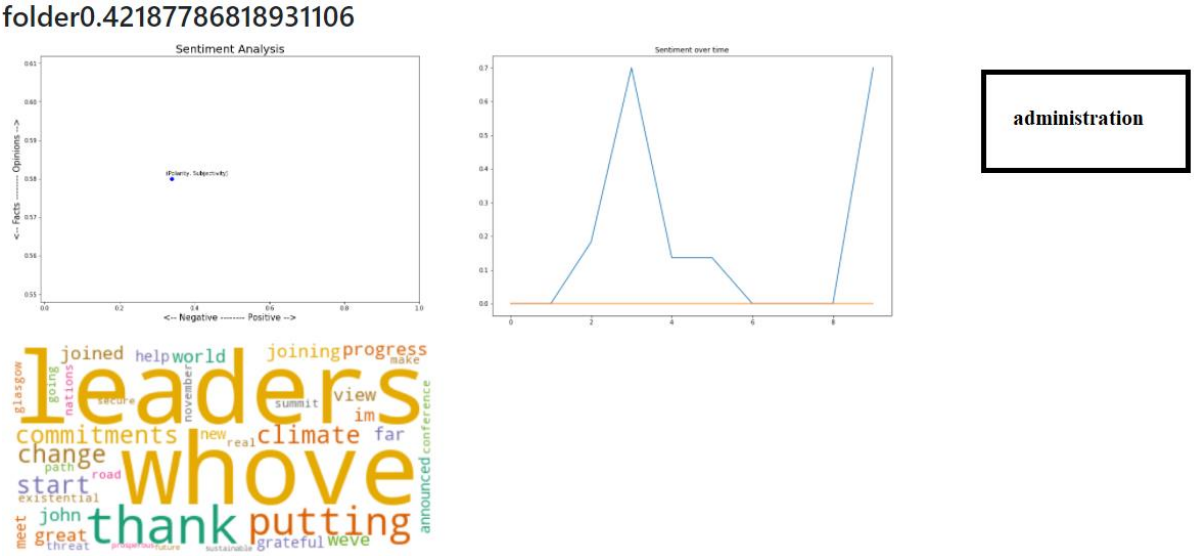
2. Input live recording:



3. Input previously recorded file:



4. Check if we get all the four output images on the UI:



In the above figure (clockwise), we can see the overall sentiment of the speaker, sentiments over time, topic the speaker talked about the most and also the 'wordcloud' that depicts the important words used by the speaker.

Thus, we tested our app to see if all of its features were running correctly and we found out that it did.

Testing autoscaling:

1. For typical number of input files (5 files):
   As a first test case, we checked if our application was autoscaling for a typical number of input files, 5. The application worked as expected, in terms of autoscaling, which is shown in fig. 1. Here, we can see that the number of instances went up from 2 to 7 and then back to 2 within a duration of 10 minutes.

Fig. 1

2. For large number of inputs (30 files):

We tested the autoscaling feature of our cloud application by giving a greater number of input files, to see if it scaled up and down as expected. Our application did implement autoscaling well, in this case as well, as shown in fig. 2. Here we can see that the number of instances were scaled from 2 to 8 and then back down to 2, in a span of 30 minutes, due to larger number of input files.

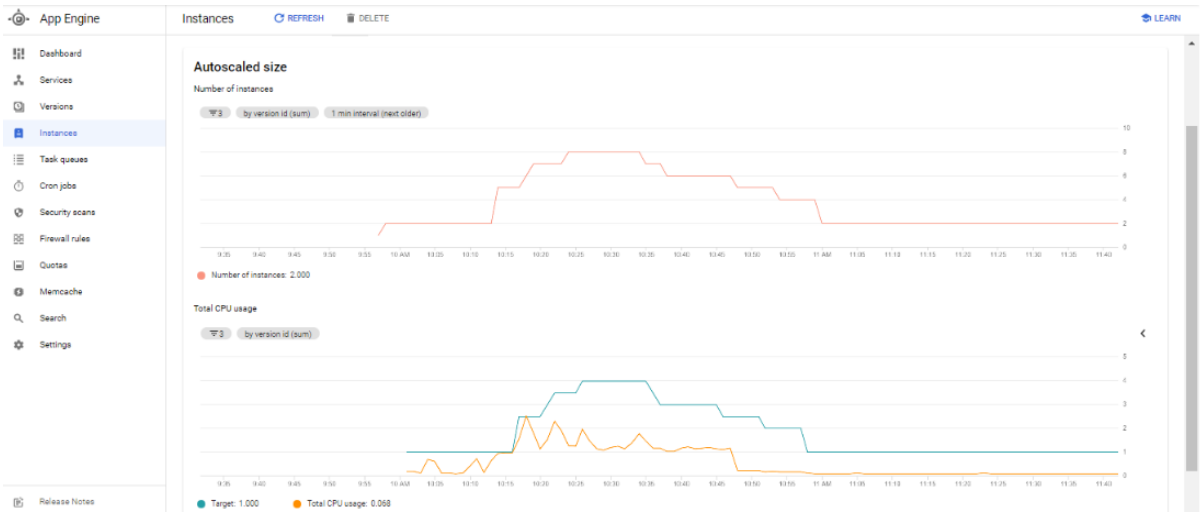


Fig. 2

5. Code

Functionality of every program of our application:

1. App-server:

    1. Static: Home_page.js
    It has the JavaScript code that runs to take the input from the user and return the output from the output cloud-bucket to the user after the inputs are processed.

    2. Templates: HomePage.html
    It has the HTML file that renders the template for the UI.

    3. main.py:
    The Flask code that runs as the front-end instance on the Google App Engine (GAE).

    4. app.yaml:
    Configuration file to run the front-end server on the GAE.

    5. credentials_file.json:
    JSON file that consists of the access credentials for the developer to use the cloud APIs.

    6. Publish.py:
    It contains the python code that helps in publishing the 'gsutil URL' of the input files to the Publisher.

    7. speech_to_text.py:
    This python file consists of the code that would convert the input recordings into transcripts and translate text files in other languages into English, using google APIs like 'speech' and 'translate'.

    8. requirements.txt:
    This file has all the required libraries that will be installed automatically when run on gcloud.

2. Web-server:

    1. Static: Home_page.js
    It has the JavaScript code that runs as a listener to fetch input text files URLs and feed it to the NLP model running in the backend.

    2. Templates: HomePage.html
    It has the HTML file that renders the template for the UI (Just to convey that the backend is running).

    3. main.py:
    The Flask code that runs as the back-end instance on the Google App Engine (GAE) to run the NLP model.

4. app.yaml:
   Configuration file to run the back-end server on the GAE.

5. credentials_file.json:
   JSON file that consists of the access credentials for the developer to use the cloud APIs.

6. subscribe.py:
   It contains the python code that helps in subscribing the 'gsutil URL' of the input files from the publisher to feed to the Python code that runs the NLP model.

7. speech_to_text.py:
   This python file consists of the code that would convert the input recordings into transcripts and translate text files in other languages into English, using google APIs like 'speech' and 'translate'.

8. requirements.txt:
   This file has all the required libraries that will be installed automatically when run on gcloud.

How to install and run the application on Google Cloud Platform?

We have used Python 3.9 as the runtime for the application.
For the front-end Flask app to run and deploy on GAE;

1. Open the 'FlaskProject' folder in an IDE and type the following command:
   - gcloud app deploy

2. To open the app in a web browser, type;
   - gcloud app browse
     and click the URL that shows up.

For the back-end Flask app to run and deploy on GAE;

3. Open the 'NLP_model' folder in an IDE and type the following command:
   - gcloud app deploy

4. To open the app in a web browser, type;
   - gcloud app browse
     and click the URL that shows up.

Thus, we can have both, the app-server and web-server instances up and running for the user to give the inputs through the UI and get back the appropriate outputs.

6. Conclusions:

We have developed a project that can process and analyze speeches, in any language, and achieve scalability and asynchrony among the instances of the application. We have implemented an NLP model

that is 75% accurate, which is more accurate than the SOTA model, which is 71% accurate. The dashboard in the UI can give the user rich patterns about the speech input by him/her.

In this project, we learnt about;

1. The usage of the Platform-as-a-Service (PaaS) model of Google Cloud Platform, by using the Google App Engine (GAE).
2. The way we can store blobs in Google Cloud Storage and achieve persistency.
3. Pub/Sub module of GCP that would help decouple web-server from the app-server and introduce asynchrony in the application.
4. Configuring of apps running on instances, so that they can auto-scale themselves automatically according to the CPU utilization percentages.
5. Different APIs like 'speech' and 'translate', that can easily be used with any application and make it sophisticated.

Ways we can improve our application;

1. We can improve our application by training the NLP model more, so that its accuracy increases.
2. We can further improve it by building a more readable dashboard by integrating the application with sophisticated tools like Tableau.
3. Also, there can be additional features added to the application, so that it gives rise to a complete-package application, like an interviewee-analytics software, where can give the user a more complete picture.

7 Individual Contributions

7.0.1 Adersh Ganesh

The project aims at building an elastic web-tier application built on top of the Google App engine using languages like Python, Html and JavaScript. The project takes in Audio as input and analyses the sentiment of the speakers and describes the topic of the speech. Furthermore, The ML Model predicts the sentiment of different time intervals and displays the Speech graph in the Front end. The below mentioned are my contributions to the phases of the project.

1. Design

The Design phase of the project includes deciding which Google App Engine Service to use, how each individual component should be connected to each other, and which framework needs to be used for the project. We collaborated as a team and came up with architecture and design that provides maximum scalability and independence for the modules.

2. Front End

We collaborated as a team and designed the user interface of the front end and I implemented the user interface design. We have used Html and JavaScript for the front end and Bootstrap V4 library for the implemented CSS and Attributes. The live recording of the audio at the UI has been done using the Recorder JS, source JavaScript file taken from multiple sources like Github and StackOverflow has been

modified according to our application.

3. Web-Tier.
The Html has to be rendered from the Backend and has to be routed to an address. I implemented the rendering template of the Html and handled the Input file (either in the format of live recording or transcripts uploading or file uploading) in the backend. I also wrote the implementation of Speech to Text recognition using the Google API, since the audio files need to be converted into transcripts since the NLP model takes only the text as the input. Our application is not constricted to a single language and I implemented the translation of non-English language to English using the 'translate' API of Google Cloud.

References:

[1] Programming Google App Engine with Python [Book] by O'Reilly.
[2] Learning document embeddings along with their uncertainties Edit social preview; Santosh Kesiraju, Oldřich Plchot, Lukáš Burget, Suryakanth V. Gangashetty.
[3] Neural Variational Inference for Text Processing Edit social preview; Yishu Miao, Lei Yu, Phil Blunsom.