Realization

Lokesh Agnihotri

# Version Table.

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Changes | Author |
| 1.0 | 25/06/2023 | Initial Draft | Lokesh Agnihotri |
|  |  |  |  |

# Abstract:

This abstract provides a comprehensive overview of the technical aspects involved in the development of the Pronunciation Trainer application for the Protectorate of Technology Tilburg. The project aims to assist MBO students in improving their English pronunciation through a user-friendly platform. This document outlines the key steps undertaken to realize the project, focusing on the front-end and back-end implementation, as well as the incorporation of data science and AI techniques to ensure accurate measurement of user speech input. Furthermore, the document highlights the importance of unit tests and their automation to guarantee the application's reliability.  
The front-end of the Pronunciation Trainer application plays a crucial role in providing an intuitive and engaging user experience. Through careful design considerations, the user interface was optimized for easy navigation and seamless interaction, enabling MBO students to effectively practice and refine their English pronunciation skills.

On the back end, a robust architecture was implemented to handle various functionalities of the Pronunciation Trainer application. A server-side programming language, such as Python, was chosen for its versatility and extensive support for data science and AI libraries. The back-end infrastructure was responsible for storing pronunciation exercises, and processing speech input for accurate analysis.   
Unit testing played a vital role in ensuring the reliability and stability of the Pronunciation Trainer application. Comprehensive test suites were developed to validate each component and functionality of the system. Automation of these tests using frameworks such as pytest allowed for continuous integration and deployment, enabling efficient bug detection and resolution.

A screen shot of a computer program

Description automatically generated with medium confidence  
The above code defines a function called getPronunciationAccuracy that calculates the pronunciation accuracy of a list of pairs containing real words and their transcribed counterparts. The function initializes variables to keep track of the total number of mismatches (total\_mismatches) and the total number of phonemes (number\_of\_phonemes). It also creates an empty list current\_words\_pronunciation\_accuracy to store the accuracy percentage for each word. The function then iterates over each pair in the input list. For each pair, it removes any punctuation from the real word and transcribed word, converts them to lowercase, and calculates the number of mismatches between the two using the edit\_distance\_python function. The number of mismatches is added to the total\_mismatches variable. The number of phonemes in the real word is determined by the length of the word without punctuation, and this value is added to the number\_of\_phonemes variable. The function calculates the pronunciation accuracy for the current word by subtracting the number of word mismatches from the number of phonemes in the word, dividing it by the number of phonemes in the word, and multiplying by 100 to get a percentage. This accuracy percentage is added to the current\_words\_pronunciation\_accuracy list. Finally, the function calculates the overall percentage of correct pronunciations by subtracting the total number of mismatches from the total number of phonemes, dividing it by the total number of phonemes, and multiplying by 100. This percentage is rounded using NumPy's round function. The function returns a tuple containing the rounded percentage of correct pronunciations and the list of pronunciation accuracy percentages for each word.

# Front End

The front end works on HTML for designing the view that the user will see and interact with, and CSS is used for the styling on how the page will look like. The HTML gives the basic architecture to layout of page, defining where will the button and textboxes be while the CSS defines how it would look like.

And JavaScript is added to the HTML, CSS to make the page dynamic, so that the page can interact with the backend and fetch and show the values returned by the backend. Also, it takes the user input to the backend for processing. The JavaScript makes the website dynamic.   
In the screenshot below once can see the basic structure of HTML containing the stylesheets and the Javascript links which will work together.   
A picture containing text, screenshot, display

Description automatically generated

Below is a screenshot of the CSS which user can see to understand how the page was designed to look.   
A screen shot of a computer screen

Description automatically generated with low confidence

For making the website Dynamic JavaScript was added to make the fetch and post requests to the backend.   
  
A screen shot of a computer program

Description automatically generated with low confidence

In the above screenshot one can see that upon clicking the button with id “next” this part of the JavaScript is triggered and a function is defined that performs the fetch request from the back end and updates the received values in the front end. The received text from the back end is stored in the reftext “which is where the user sees the next sentence they have to practice” and the sound received is send to the” next\_button\_audio” element for user to listen to and practice.

Similarly, when the button Pronunciation trainer is clicked the following part of code gets triggered and phonetics and actual text are updated in the text box.  
A screen shot of a computer program

Description automatically generated with low confidence

For the record button as well, when it is clicked it creates a recording from the microphone and sends it to the backend for processing as well as retrieves the recording from the back end and play it for the user.

A picture containing text, screenshot, software, multimedia software

Description automatically generated

Back End:

For the back-end flask was chose as it is a lightweight python framework. The backend generally involved in sending the data to the front end and processing the data from the front end to create the insights.

In the backend multiple python libraries are involved such as ing-to-ipa to find the phonetics of the reference text that user must speak. Along with speech-to-text which is used to convert the speech recorded form the user to the text. Pyttsc3 library was used to convert the text to speech so user can hear what they spoke as well as they can head the correct pronunciation of the phonetics from the reference text. Here is how it works:   
  
A screen shot of a computer program

Description automatically generated with low confidence

The necessary libraries and modules are imported to support the functionality of the back end. These include json for JSON manipulation, os for file and directory operations, random for generating random words, ast for literal evaluation of strings, pydub for audio file manipulation, eng\_to\_ipa for converting English text to International Phonetic Alphabet (IPA) notation, pyttsx3 for text-to-speech conversion, speech\_recognition for speech-to-text conversion, and Flask and related modules for web application development.

A screenshot of a computer program

Description automatically generated with medium confidence

The Flask application is initialized, and CORS (Cross-Origin Resource Sharing) is enabled to handle requests from different origins. The configuration is set to allow all headers. The file upload and storage paths are defined, along with a variable to store the reference text used for pronunciation comparison.

A picture containing text, screenshot, font

Description automatically generated

This route handles the main endpoint of the application and returns the rendered HTML template for the user interface.

A screen shot of a computer program

Description automatically generated with low confidence

These routes handle the retrieval of reference audio files. The first route without a specified file name retrieves a list of available audio files, while the second route with a specified file name retrieves a specific audio file. The audio file is read and returned as a response with appropriate headers.

A screen shot of a computer program

Description automatically generated with low confidence

This code snippet defines a function convert\_audio\_to\_text that converts an audio file to text using speech recognition. The audio file is first converted to PCM WAV format and then processed using the speech\_recognition library. The recognized text is returned as the output.

A picture containing text, screenshot, software

Description automatically generated

This function converts the given text to speech using the pyttsx3 library. It takes a text parameter representing the text to be converted and an audio\_name\_location parameter representing the desired name and location of the output audio file. The function initializes the speech synthesis engine, searches for a female voice, sets the voice property, saves the speech as an audio file, and runs the speech synthesis engine. The filename of the generated audio file is returned, and any errors that occur during the process are printed.

A picture containing text, screenshot, font

Description automatically generated

This function converts the given text to its corresponding International Phonetic Alphabet (IPA) representation using the eng\_to\_ipa library. It simply calls the convert function from the library and returns the converted text.

A picture containing text, screenshot, software

Description automatically generated

This function is a route handler for the "/pronunciation\_trainer" route. It handles a POST request and expects JSON data containing the text to be used for pronunciation training. It converts the text to IPA representation using the convert\_text\_to\_ipa function, saves the pronunciation audio file using the convert\_text\_to\_speech function, and constructs a JSON response containing the IPA representation and the filename of the audio file. The response is returned as a JSON string.

A picture containing text, screenshot, font

Description automatically generated

This function is a route handler for the "/next\_word" route. It handles a GET request and randomly selects a word from an English dictionary file. It converts the selected word to IPA representation, saves the pronunciation audio file, and constructs a JSON response containing the selected word, its IPA representation, and the filename of the audio file. The response is returned as a JSON string.

A screen shot of a computer program

Description automatically generated with medium confidence

This function is a route handler for the "/upload-audio" route. It handles both POST and GET requests. It checks if the audio file and reference text are present in the request. If not, appropriate error messages are returned. The audio file is saved, converted to text using the convert\_audio\_to\_text function, and the recognized text is printed. The reference text is retrieved from the request. Word error rate (WER) and character error rate (CER) are calculated between the reference text and the recognized text using the wer and cer functions from the accuracy module. The function also performs other operations related to matching words, calculating pronunciation accuracy, and constructs a JSON response containing various metrics and data. The response is returned as a JSON string.

A picture containing text, screenshot, font

Description automatically generated

This code block runs the Flask application when the script is executed directly, setting the host to "0.0.0.0" and the port to 5000, with debugging enabled.

# Calculating Accuracy by Comparing strings, words, and phenomes

The accuracy is calculated by implementing the levenshtein distance in dynamic programming in our code. Before proceeding forward, we need to first understand what exactly is levenshtein distance and how it works in real life:

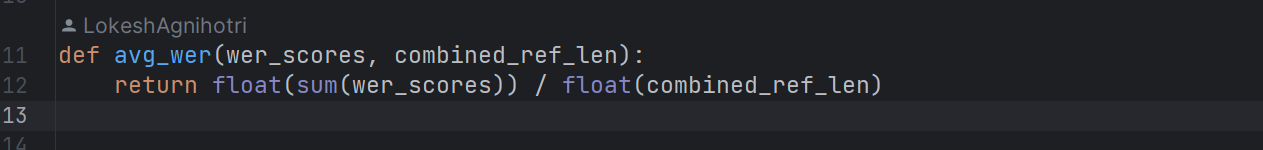
Levenshtein distance, also known as edit distance, is a measure of the difference or similarity between two strings. It calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another.  
The Levenshtein distance algorithm is named after Vladimir Levenshtein, who introduced it in 1965. It has various applications in fields such as computer science, computational linguistics, and bioinformatics.

Here's a step-by-step explanation of how the Levenshtein distance is calculated:

1. The algorithm takes two input strings, let's call them String A and String B.
2. It creates a matrix (often referred to as the "distance matrix") with dimensions (M+1) x (N+1), where M is the length of String A and N is the length of String B.
3. The first row of the matrix represents the characters of String A, and the first column represents the characters of String B.
4. The matrix is initialized as follows:
   1. The value at position (0,0) is set to 0.
   2. The values in the first row are set to their respective column indices (0 to N).
   3. The values in the first column are set to their respective row indices (0 to M).
5. Starting from position (1,1) of the matrix, each cell's value is calculated based on three operations:
   1. If the characters at the corresponding positions in String A and String B are the same, the value in the current cell is the same as the value in the diagonal cell above-left.
   2. If the characters are different, the value in the current cell is the minimum of the following three values:
   3. The value in the cell above plus 1 (representing deletion).
   4. The value in the cell to the left plus 1 (representing insertion).
   5. The value in the diagonal cell above-left plus 1 (representing substitution).
6. The algorithm iterates through all cells of the matrix, filling them in based on the defined rules.
7. After iterating through all the cells, the value in the bottom-right cell of the matrix represents the Levenshtein distance between String A and String B.

The Levenshtein distance is a useful metric for tasks such as spell checking, DNA sequence alignment, and fuzzy string matching. It quantifies the difference between two strings and provides a measure of similarity or dissimilarity based on the number of edits required to transform one string into another.

When applying it in code the accuracy.py file looks like this:



This function calculates the average word error rate (WER) given a list of WER scores and the combined length of the reference sentences. It adds up all the WER scores and divides the sum by the combined length of the reference sentences to obtain the average WER. The average WER is then returned.

A screen shot of a computer program

Description automatically generated with low confidence

This function calculates the Levenshtein distance between two sequences, ref and hyp. The Levenshtein distance represents the minimum number of insertions, deletions, and substitutions required to transform ref into hyp. It uses a dynamic programming approach to compute the distance.

The function initializes a distance matrix with appropriate dimensions and populates the first row and column with incremental values. Then, it iterates over the remaining cells of the matrix and calculates the minimum cost for each operation (deletion, insertion, or substitution). Finally, it returns the value in the bottom-right corner of the matrix, which represents the Levenshtein distance.

A picture containing text, screenshot, font

Description automatically generated

This function calculates the word error rate (WER) between a reference sentence and a hypothesis sentence. It takes the reference and hypothesis sentences as inputs, and optionally allows for case-insensitive comparison and specification of a delimiter to split the sentences into words.

First, if case-insensitive comparison is enabled, the reference and hypothesis sentences are converted to lowercase. Then, both sentences are split into word sequences using the specified delimiter. The function calls \_levenshtein\_distance to calculate the edit distance between the word sequences.  
Finally, it returns the edit distance and the number of words in the reference sentence.

A picture containing text, screenshot, font, software

Description automatically generated

This function calculates the character error rate (CER) between a reference string and a hypothesis string. It takes the reference and hypothesis strings as inputs and allows for optional case-insensitive comparison and removal of internal spaces. If case-insensitive comparison is enabled, the reference and hypothesis strings are converted to lowercase. If the removal of spaces is enabled, all space characters are removed from both strings. The function then calls \_levenshtein\_distance to calculate the edit distance between the character sequences. Finally, it returns the edit distance and the length of the reference string. These functions provide the foundation for evaluating the accuracy of transcriptions or text recognition systems by calculating the word error rate (WER) and character error rate (CER) metrics.

A picture containing text, screenshot, software, font

Description automatically generated

The function definition specifies the input parameters reference, hypothesis, ignore\_case, and delimiter. The ignore\_case parameter is a boolean indicating whether case sensitivity should be considered, and the delimiter parameter determines how the sentences should be split into words.

The function's docstring provides an overview of WER and its calculation. It mentions that WER is calculated based on the number of substituted, deleted, and inserted words, divided by the total number of words in the reference.

The word\_errors function is called to calculate the edit distance and the length of the reference sentence. The code checks if the reference length is zero and raises a ValueError if that is the case, as the WER calculation is not defined for an empty reference. Finally, the WER is calculated by dividing the edit distance by the reference length, and the result is returned. The WER is a common metric used in speech recognition and natural language processing tasks to evaluate the accuracy of speech or text recognition systems.

A picture containing text, screenshot, software, font

Description automatically generated

The function cer takes four parameters: reference (the reference sentence), hypothesis (the hypothesis sentence), ignore\_case (a flag indicating whether to consider case sensitivity), and remove\_space (a flag indicating whether to remove internal space characters). The char\_errors function returns two values: edit\_distance and ref\_len. The edit\_distance represents the number of character errors (substitutions, deletions, insertions) between the reference and hypothesis sentences, while ref\_len represents the length of the reference sentence. The code checks if ref\_len is equal to 0, which would indicate an invalid reference sentence with zero length. If that's the case, it raises a ValueError with the message "Length of reference should be greater than 0." If the reference length is valid, the code calculates the Character Error Rate (CER) by dividing the edit\_distance by ref\_len and assigns the result to the variable cer.

Finally, the function returns the calculated CER.

A picture containing text, screenshot, font

Description automatically generated

The above code defines a function called get\_resulting\_string that takes three input parameters: mapped\_indices, words\_estimated, and words\_real. The purpose of this function is to generate a mapped string based on the indices provided, using the estimated and real words as references. Inside the function, the code initializes empty lists to store the resulting mapped words and their corresponding indices. It also sets a constant variable for a special token to represent words that are not found in the mapping process. The code then iterates over each index of the real words. For each index, it checks if there are any corresponding positions in the mapped indices. If no position is found, it adds the word not found token and -1 to the lists and continues to the next index. If only one position is found, it adds the estimated word at that position and the corresponding index to the lists. In cases where multiple positions are found, the code calculates the error between each estimated word and the real word and selects the position with the lowest error. The estimated word with the lowest error and its position are added to the lists. Finally, the function returns the lists of mapped words and indices as a tuple.

A picture containing text, screenshot, font, software

Description automatically generated

This code defines a function called get\_word\_distance\_matrix that takes two input parameters: words\_estimated and words\_real, both of which are lists. The purpose of this function is to calculate a word distance matrix based on the estimated and real words. The code first determines the number of real words and the number of estimated words by getting the lengths of the respective lists. It then creates an empty matrix called word\_distance\_matrix with dimensions (number\_of\_estimated\_words + offset\_blank, number\_of\_real\_words). The offset\_blank variable is used to control the size of the matrix. Next, the code iterates over the indices of the estimated words using a for loop. Within this loop, it iterates over the indices of the real words using another for loop. For each combination of estimated and real word indices, it calculates the edit distance between the corresponding words using the edit\_distance\_python function. The resulting edit distance value is stored in the corresponding cell of the word\_distance\_matrix. After calculating the edit distances for all word combinations, the code checks if offset\_blank is equal to 1. If so, it enters another for loop that iterates over the indices of the real words. Within this loop, it assigns the length of each real word to the last row of the word\_distance\_matrix, specifically to the corresponding column. Finally, the function returns the resulting word\_distance\_matrix as a NumPy array.

A picture containing text, screenshot, font

Description automatically generated

The get\_best\_path\_from\_distance\_matrix function uses the Google OR-Tools CP-SAT solver to find the best path through a distance matrix. It creates variables to represent the order of estimated words and constraints to ensure the words are in ascending order. It also handles the matching of estimated and real words, calculates the total distance of the best path, and minimizes this distance. The function returns the indices of the estimated words in the best path.

A picture containing text, screenshot, software, display

Description automatically generated

The code snippet represents a function called edit\_distance\_python that calculates the edit distance (also known as the Levenshtein distance) between two input sequences seq1 and seq2. The edit distance is the minimum number of operations (insertions, deletions, and substitutions) required to transform seq1 into seq2. The function initializes a matrix of size (size\_x, size\_y) using NumPy, where size\_x is the length of seq1 plus one and size\_y is the length of seq2 plus one. The matrix is used to store the edit distance values for each prefix of seq1 and seq2. Next, the function initializes the first row and column of the matrix with increasing values from 0 to size\_x - 1 and size\_y - 1, respectively. These values represent the edit distances between the empty string and the prefixes of seq1 and seq2. The function then iterates through the remaining cells of the matrix, starting from index (1, 1). For each cell, it checks if the corresponding characters in seq1 and seq2 are equal. If they are equal, the current cell's value is set to the minimum of the following three values: the cell above plus 1, the cell diagonally above-left, and the cell to the left plus 1. If the characters are not equal, the current cell's value is set to the minimum of the cell above plus 1, the cell diagonally above-left plus 1, and the cell to the left plus 1. This step calculates the edit distance for each prefix of seq1 and seq2. Finally, the function prints the matrix (for debugging purposes) and returns the value at the bottom-right cell of the matrix, which represents the minimum edit distance between the two input sequences.

# Unit tests

The "test.py” and “test\_accuracy.py” files serves a practical purpose. It contains a set of unit tests for a Flask application. These tests are designed to verify the application's functionality, ensuring that it behaves correctly and produces the expected results. The code in "these files uses the unittest framework, a popular testing tool in Python. It begins by importing necessary modules, including unittest itself and any dependencies required for testing.

A screen shot of a computer program

Description automatically generated with medium confidence

The file defines a class called AppTestCase, which inherits from unittest.TestCase. This class acts as a container for individual test methods. Each test method is responsible for checking a specific aspect of the Flask application's behavior. To set up the testing environment, the setUp method is implemented. It creates an application context and sets up a test client, allowing simulated requests to be sent to the Flask application. The tearDown method is used to clean up the testing environment after each test method execution. It ensures that each test starts with a clean state, preventing any interference between tests. The test methods themselves start with the keyword test\_ and use assertions to check the expected behavior of the Flask application. These assertions verify properties such as the status code of the response, the content type, and the presence of specific data in the response. Overall, the "test.py" file plays a crucial role in testing the Flask application, helping to identify and fix any issues or bugs before deploying it. It ensures that the application functions as intended, providing confidence in its reliability and correctness.

# Conclusion

In conclusion, I have provided an overview of the Pronunciation Trainer application, explaining its development process from the front end to the back end. I have highlighted the importance of user-friendly design, data science techniques, and accurate measurement of user speech input. Additionally, I have emphasized the significance of unit tests and their automation in ensuring the application's reliability. By following these steps, we have created an effective tool to assist MBO students in improving their English pronunciation skills.