A Project Report

on

ML POWERED TEXT AUTO-COMPLETION AND GENERATION

Submitted in partial fulfillment of the requirements

for the award of degree of

BACHELOR OF TECHNOLOGY

in

Information Technology

by

J. Nikhitha (19WH1A1207)
P. Pooja Sri (19WH1A1216)
G. Pavithra (19WH1A1218)
Ch. Sai Meghana (19WH1A1246)

Under the esteemed guidance of
Dr. P. Kayal
Associate Professor

Department of Information Technology

BVRIT HYDERABAD College of Engineering for Women

Rajiv Gandhi Nagar, Nizampet Road, Bachupally, Hyderabad – 500090

(Affiliated to Jawaharlal Nehru Technological University, Hyderabad)

(NAAC ‘A’ Grade & NBA Accredited- ECE, EEE, CSE, IT)

June, 2023

DECLARATION

We hereby declare that the work presented in this project entitled **"ML Powered Text Auto-Completion and Generation"** submitted towards completion of the project in IV year II sem of B.Tech IT at "BVRIT HYDERABAD College of Engineering for Women", Hyderabad is an authentic record of our original work carried out under the esteemed guidance of Dr. P. Kayal, Associate Professor, Department of Information Technology.

J. Nikhitha (19WH1A1207)

P. Pooja Sri (19WH1A1216)

G. Pavithra (19WH1A1218)

Ch. Sai Meghana (19WH1A1246)

BVRIT HYDERABAD

College of Engineering for Women

Rajiv Gandhi Nagar, Nizampet Road, Bachupally, Hyderabad – 500090

(Affiliated to Jawaharlal Nehru Technological University Hyderabad)

(NAAC 'A' Grade & NBA Accredited- ECE, EEE, CSE & IT)

CERTIFICATE

This is to certify that the Project report on **"ML Powered Text Auto-completion and Generation"** is a bonafide work carried out by **J.Nikhitha (19WH1A1207), P.Pooja Sri (19WH1A1216), G.Pavithra (19WH1A1218)** and **Ch.Sai Meghana (19WH1A1246)** in the partial fulfillment for the award of B.Tech degree in **Information Technology, BVRIT HYDERABAD College of Engineering for Women, Bachupally, Hyderabad** affiliated to Jawaharlal Nehru Technological University, Hyderabad under my guidance and supervision.

The results embodied in the project work have not been submitted to any other university or institute for the award of any degree or diploma.

Internal Guide	**Head of the Department**
Dr. P. Kayal	**Dr. Aruna Rao S L**
Associate Professor	**Professor & HoD**
Department of IT	**Department of IT**

External Examiner

ACKNOWLEDGEMENT

We would like to express our profound gratitude and thanks to **Dr. K. V. N. Sunitha, Principal, BVRIT HYDERABAD College of Engineering for Women** for providing the working facilities in the college.

Our sincere thanks and gratitude to **Dr. Aruna Rao S L, Professor & Head, Department of IT, BVRIT HYDERABAD College of Engineering for Women** for all the timely support, constant guidance and valuable suggestions during the period of our project.

We are extremely thankful and indebted to our internal guide, **Dr. P. Kayal, Associate Professor, Department of IT, BVRIT HYDERABAD College of Engineering for Women** for her constant guidance, encouragement and moral support throughout the project.

Finally, we would also like to thank our Project Coordinators **Dr. P. Kayal, Associate Professor and Ms. K. Niraja, Assistant Professor**, all the faculty and staff of the Department of IT who helped us directly or indirectly, parents and friends for their cooperation in completing the project work.

J. Nikhitha (19WH1A1207)

P. Pooja Sri (19WH1A1216)

G. Pavithra (19WH1A1218)

Ch. Sai Meghana (19WH1A1246)

ABSTRACT

Natural Language Processing (NLP) is a sub-field of artificial intelligence that assists computers with understanding human language. It combines the power of linguistics and computer science to contemplate the guidelines and structure of language and make intelligent systems fit for comprehension, breaking down, and separating significance from text and speech. Using this concept, the project proposes a system which helps in minimizing the human effort by providing the features like text auto-completion and generation. Auto-completion refers to the completion of a word, or a phrase, as we start typing in a document. The prediction is based upon the selection of the most likely word from a set of frequently used words. The text prediction task consists of editing text with the minimum number of keystrokes feasible. This method is suggesting words that the user intended to write, and the system predicts the next word related to the previous work. Text generation is a feature which is an enhanced version of Gmail's Smart Reply. This feature helps in generating a template of the mail by classifying the subject line. The purpose of our intelligent system is to help differently ab-led people by increase their typing speed, as well as to help them decrease the number of keystrokes needed in order to complete a word or a sentence. The proposed system is also very useful for multi domain professionals, who writes text, particularly people–such as medical doctors–who frequently use long, hard-to-spell terminology that may be technical or medical in nature.

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	**Meaning**
ML	Machine Learning
NLP	Natural Language Processing
OCC	One Click Chat
LSTM	Long short-term memory
RNN	Recurrent Neural Network
LM	Language Model
OOV	Out-of-Vocabulary
LTM	Long Term Memory
STM	Short Term Memory

CONTENTS

1. INTRODUCTION

In our daily life we need various means of communication. Now we have fax, telex, text message, mobile phone etc. But E-mail has brought about a revolutionary change in our communication system. All we need for this system is an internet connection and a personal computer or smartphone. We can instantly communicate with someone on the other side of the globe. There are separate mailboxes in this system. So acute privacy can be maintained in this system. It is also very cheap and easy. So the students also can use Email to communicate with their friends at home and abroad. As it is used all over the world, and it is most modern communication system. As email is the widely used means of communication across the world to make this more user friendly we are proposing a system with two features, Text Auto-Completion and Text Generation. It is crucial that there be an instant messaging option in our busy lives because it will help consumers save a lot of time and effort.

Auto-complete is a user interface function in which an application predicts a word or phrase that the user needs to type without the user having to type it entirely. In modern applications, word completion/auto-complete/auto-suggest is a popular user interface feature. Its aim is to predict what the user wants to type and add sections of the text automatically. By providing available options, the aim is to speed up typing, assist those with typing problems, correct/prevent spelling errors, and promote information retrieval. Witten and Daraghs' work on the Reactive Keyboard from 1983 may be the earliest example of the concept. Several other methods have been identified since then, but the basic concept has remained the same. Word processors, programming editors, desktop applications, HTML form elements on websites, web applications (Google Suggest, web based e-mail clients), mobile phone interfaces, Unix terminals, and so on all have the feature. Auto-complete systems can be extremely useful or extremely irritating, depending on the scenario and implementation. It's worth noting that auto-complete works well in limited contexts and is nearly useless in arbitrary natural text entry. Many mobile devices have a mechanism that allows for faster typing by using several functions of the keys and leveraging the language's redundancy.

Text generation is the task of generating text with the goal of appearing indistinguishable to human-written text. This task is more formally known as ”natural language generationïn the literature.

Text generation is a sub field of natural language processing. It leverages knowledge in computational linguistics and artificial intelligence to automatically generate natural language texts, which can satisfy certain communicative requirements. Text generation is a field that has been developing since the 1970s and is regarded as a subsection of NLP (Natural Language Processing). Developing deep learning models for text generation is an ongoing process in the field of NLP. The user needs to be filling the subject line in the mail and according to the subject, the model generates the body of the mail. Text Auto-completion and Text Generation both work collectively and make the user experience more convenient while using the mail. These features are responsible for instant messaging through the mail. The auto-complete completes the upcoming words from predicting and text generation is responsible for generating the body of the mail through the subject line.

1.1. Objective

To build a model that helps in decreasing the number of keystrokes needed in order to complete a word or a sentence. It mainly decreases the time consumption of the user and also our intelligent system helps differently ab-led people to increase their typing speed.

1.2. Problem Definition

With the extending popularity of social media platforms people are connecting with each other more and more nowadays. As a result, the use of instant messaging systems is also increasing day by day. The need for such a system is mainly useful, when the user is busy or not in a position to reply. Our system takes the Subject line as input and generates a template for the body.

1.3. Modules

1. Data Pre-Processing
2. Building the Model
3. Training the Model
4. Testing the Model

2. LITERATURE SURVEY

Mia Xu Chen, Benjamin N Lee, Gagan Bansal, Yuan Cao, Shuyuan Zhang, Justin Lu, Jackie Tsay, Yinan Wang, Andrew M. Dai, Zhifeng Chen, Timothy Sohn, and Yonghui Wu proposed Smart Compose [1], a novel system for generating interactive, real-time suggestions in Gmail that assists users in writing mails by reducing repetitive typing. In the design and deployment of such a large-scale and complicated system, we faced several challenges including model selection, performance evaluation, serving and other practical issues. At the core of Smart Compose is a large-scale neural language model. We leveraged state-of-the-art machine learning techniques for language model training which enabled high-quality suggestion prediction, and constructed novel serving infrastructure for high-throughput and real-time inference. Experimental results show the effectiveness of our proposed system design and deployment approach. This system is currently being served in Gmail.

Email use continues to grow even as other methods of interpersonal communication, such as instant messaging, social networking and chat are seeing strong adoption. In 2017, the total number of business and consumer emails sent and received per day will reach 269 billion, and is expected to continue to grow at an average annual rate of 4.4% over the next four years, reaching 319.6 billion by the end of 2021. Improving the user experience by simplifying the writing process is a top priority of major email service providers like Gmail. To fulfil this goal, previously Gmail introduced Smart Reply, a system for automatically generating short replies in response to incoming e-mail messages. While such a feature significantly reduces user response time, the suggestions are constrained to short phrases and appear only once in the composition process. Even with an initial suggestion, drafting longer messages can still be a time-consuming process, and is arguably one in which the user most needs accurate and frequent suggestions. So here we will try to build a Smart Compose providing real-time, interactive suggestions to help users compose messages quickly and with confidence. Smart Compose helps by cutting back on repetitive idiomatic writing via providing immediate context-dependent suggestions.

A powerful neural language model trained on a large amount of email data is used to build this Smart Compose to make instant predictions as user types. To provide high-quality suggestions and smooth user experience, we need to properly handle a variety of issues including the model evaluation and large-scale inference, which we detail below. To build smart compose we are using Enron Email Data-set, Which contains approximately 500,000 emails generated by employees of the Enron Corporation. It was obtained by the Federal Energy Regulatory Commission during its investigation of Enron's collapse.

Yue Weng, Huaixiu Zheng, Franziska Bell, and Gokhan Tur introduced Uber's smart reply system: one-click-chat (OCC) [2], which is a key enhanced feature on top of the Uber in-app chat system. It enables driver-partners to quickly respond to rider messages using smart replies. The smart replies are dynamically selected according to conversation content using machine learning algorithms. Our system consists of two major components: intent detection and reply retrieval, which are very different from standard smart reply systems where the task is to directly predict a reply. It is designed specifically for mobile applications with short and non-canonical messages. Reply retrieval utilizes pairings between intent and reply based on their popularity in chat messages as derived from historical data. For intent detection, a set of embedding and classification techniques are experimented with, and we choose to deploy a solution using unsupervised distributed embedding and nearest-neighbor classifier. It has the advantage of only requiring a small amount of labeled training data, simplicity in developing and deploying to production, and fast inference during serving and hence highly scalable. At the same time, it performs comparably with deep learning architectures such as word-level convolutional neural network. Overall, the system achieves a high accuracy of 76% on intent detection. Currently, the system is deployed in production for English-speaking countries and 71% of in-app communications between riders and driver-partners adopted the smart replies to speedup the communication process.

Uber's ride-sharing business connects driver partners to riders who need to transport around cities. App provides navigation and many other innovative technology and features that assist driver partners with finding riders at the pick-up locations. Uber's in-app chat , a real-time in-app messaging platform launched in early 2017, is one of them. Before having the functionality to chat within the app, communication between customers occurred outside of the mobile app experience using third-party technologies. As resulted in safety concerns, higher operational costs, fewer completed trips, and most importantly, limits the company's ability to understand and resolve challenges both riders and driver partners were having while using the app. Although the newly added chat feature has solved many of these problems by bringing the chat experience into the app, it still requires driver-partners to type messages while driving, which is a huge safety concern. According to a public study, compared to regular driving, accident risk is about 2.2 times higher when talking on a hand-held cell phone and 6.1 times higher when texting . Before, to provide a safe and smooth in-app chat experience for driver-partners, we developed One-Click Chat (OCC), a smart reply system that allows driver-partners to respond to messages using smart replies selected dynamically according to the conversation context.

Rajeev Gupta, Ranganath Kondapally, Chakrapani Ravi Kiran proposed a personalized chatbot [3] which is designed as a digital chatting assistant to the user. The key characteristic of a personalized chatbot is that it should have a consistent personality with the corresponding user. It can talk the same way as the user when it is delegated to respond to others' messages. Many methods have been proposed to assign a personality to dialogue chatbots, but most of them utilize explicit user profiles, including several persona descriptions or key-value-based personal information. In a practical scenario, however, users might be reluctant to write detailed persona descriptions, and obtaining a large number of explicit user profiles requires tremendous manual labor. To tackle the problem, it presents a retrieval-based personalized chatbot model, namely IMPChat, to learn an implicit user profile from the user's dialogue history. It argues that the implicit user profile is superior to the explicit user profile regarding accessibility and flexibility. IMPChat aims to learn an implicit user profile through modeling user's personalized language style and personalized preferences separately. To learn a user's personalized language style, They elaborately build language models from shallow to deep using the user's historical responses; To model a user's personalized preferences, It explores the conditional relations underneath each post-response pair of the user. The personalized preferences are dynamic and context-aware: we assign higher weights to those historical pairs that are topically related to the current query when aggregating the personalized preferences. It matches each response candidate with the personalized language style and personalized preference, respectively, and fuse the two matching signals to determine the final ranking score and conducts comprehensive experiments on two large datasets, and the results show that our method outperforms all baseline models.

Shao T., Chen H., Chen W. introduced a personalized query auto-completion (QAC) system using a recurrent neural network language model [4]. The system aims to enhance the search engine experience by generating personalized query suggestions based on a user's previous queries. By leveraging an adaptable language model, the system effectively handles rare and unseen query prefixes through online updating of user information. The study demonstrates that the personalized predictions generated by the model outperform a baseline approach that does not utilize user information. The proposed approach combines the benefits of personalization and language modeling to improve the efficiency and effectiveness of query auto-completion, thereby enhancing the overall search engine feature.

Aaron Jaech and Mari Ostendorf proposed a hybrid query auto-completion (QAC) model called FS-QAC [5], which combines semantic similarity and query frequency to improve the prediction of user query intention. The traditional QAC methods often overlook the importance of semantic relevance between queries, which can significantly impact search intentions. By employing the word2vec method to measure the semantic similarity, the proposed FS-QAC model demonstrates enhanced performance in predicting user query intention and assisting in formulating accurate queries. Experimental results show that the optimal hybrid model achieves a 7.54 % improvement in terms of Mean Reciprocal Rank (MRR) compared to a state-of-the-art baseline using public AOL query logs. The study also examines the influence of prefix length on model performance. Overall, the contributions of this research include the introduction of semantic similarity feature in QAC systems using word2vec, the development of the hybrid FS-QAC model, and the evaluation of its effectiveness against existing methods.

3. SYSTEM ANALYSIS & DESIGN

To implement the project we used VS Code and Google Colab. After referring to the research papers the following are the finalized technologies to design the project. Below mentioned is the System Design of the project explained in detail.

3.1. Architecture Design

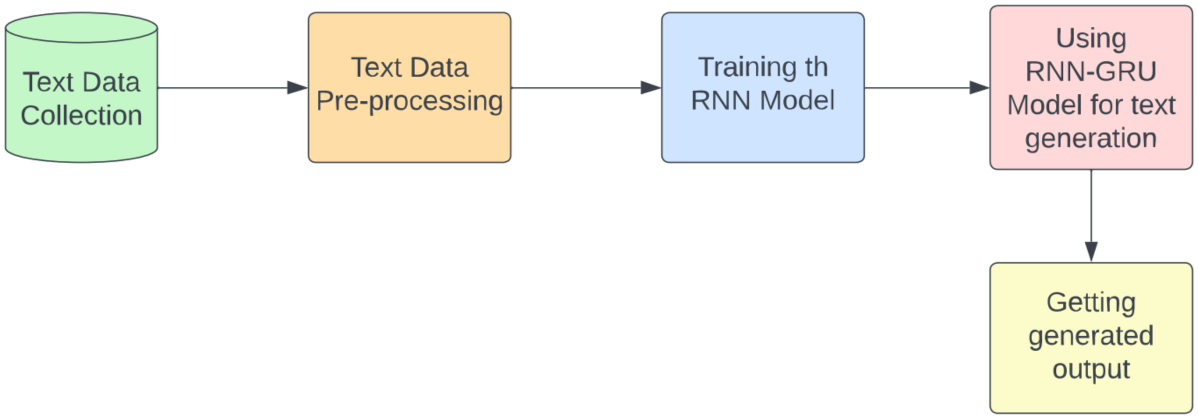


Figure 3.1.1: Project Flow for Training Text Generation Module

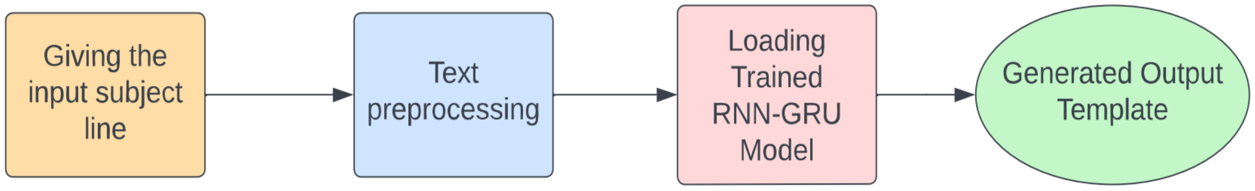

Giving the input subject line
Text preprocessing
Loading Trained RNN-GRU Model
Generated Output Template

Figure 3.1.2: Project Flow for Testing Text Generation Module

The diagrams 3.1.1 and 3.1.2 depict the project flow of the text generation module. We have first collected the email templates on various classes which include Leave, work-related, marketing, job applications, education, travel and personal templates. The data is next pre-processed, where the special characters are removed. After that we tokenize the data for training. After that we construct the model, which is made of an embedding layer, a GRU layer, a dense output layer. The model is then trained with 100-epoch iteration count and a categorical cross-entropy loss function. The resultant template is then generated.

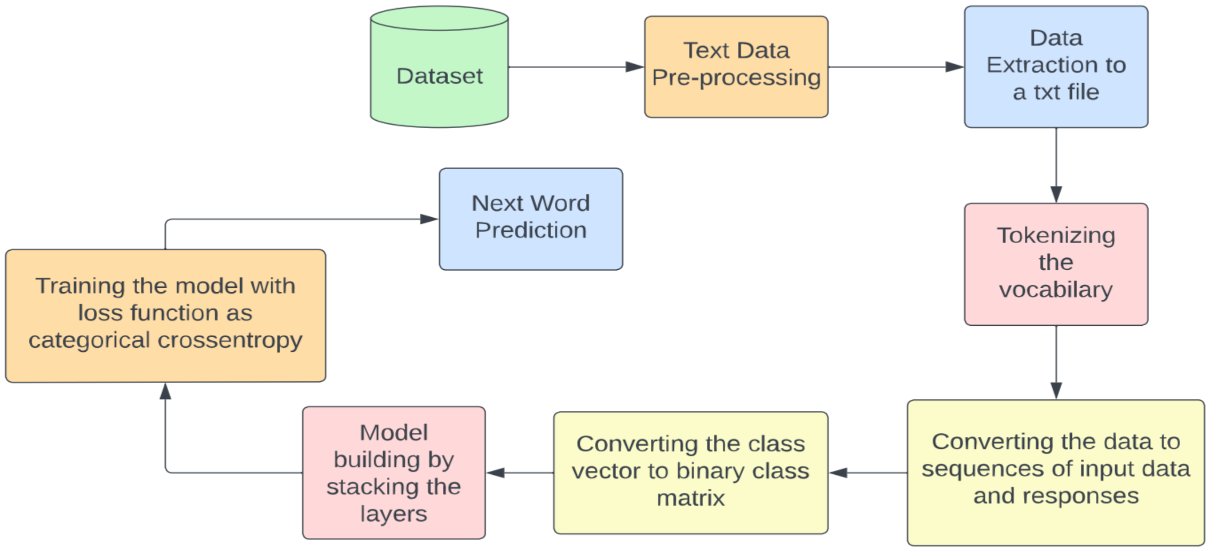

Dataset
Text Data Pre-processing
Data Extraction to a txt file
Tokenizing the vocabilary
Converting the data to sequences of input data and responses
Converting the class vector to binary class matrix
Model building by stacking the layers
Training the model with loss function as categorical crossentropy
Next Word Prediction

Figure 3.1.3: Project Flow for Text Auto-Completion Module

The diagram above shows the model's flow. The Enron Email dataset is first pre-processed, and the data is then scraped and saved to a text file. Tokenizing the vocabulary follows. The data is next transformed into sequences and then to binary class matrices. After that, we construct the model, which is made up of an embedding layer, an encoder LSTM layer, a decoder LSTM layer, a dense output layer, and a softmax layer. The model is then trained with a 35-epoch iteration count and a categorical cross-entropy loss function. The following word is then predicted.

3.2 Technologies

3.2.1. Keras

Keras is an open-source high-level Neural Network library, which is written in Python is capable enough to run on Theano, TensorFlow, or CNTK. It was developed by one of the Google engineers, Francois Chollet. It is made user-friendly, extensible, and modular for facilitating faster experimentation with deep neural networks. It not only supports Convolutional Networks and Recurrent Networks individually but also their combination and LSTM which we were using.

3.2.2. Tensorflow

TensorFlow is an end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML, and developers easily build and deploy ML-powered applications.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence Research organization to conduct machine learning and deep neural networks research. The system is general enough to be applicable in a wide variety of other domains, as well.

3.2.3. Pickle

The pickle module implements binary protocols for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling," or "flattening"; however, to avoid confusion, the terms used here are "pickling" and "unpickling".

3.2.4. SSL

SSL module provides access to Transport Layer Security (often known as “Secure Sockets Layer”) encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, macOS, and probably additional platforms, as long as OpenSSL is installed on that platform.

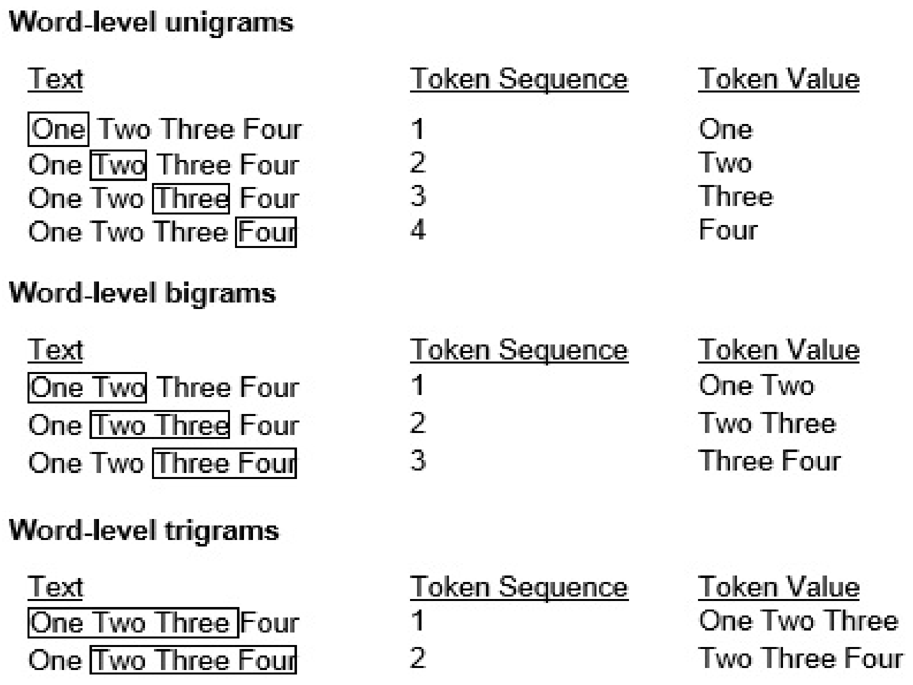
3.2.5. NLP

Natural language processing (NLP) is the ability of a computer program to understand human language as it is spoken and written – referred to as natural language. It is a component of artificial intelligence (AI). NLP enables computers to understand natural language as humans do. Whether the language is spoken or written, natural language processing uses artificial intelligence to take real-world input, process it, and make sense of it in a way a computer can understand. Just as humans have different sensors – such as ears to hear and eyes to see – computers have programs to read and microphones to collect audio. And just as humans have a brain to process that input, computers have a program to process their respective inputs. At some point in processing, the input is converted to code that the computer can understand.

3.2.6. N-grams

N-gram is a sequence of the N-words in the modeling of NLP. Consider an example of the statement for modeling. “I love reading history books and watching documentaries”. In one-gram or unigram, there is a one-word sequence. As for the above statement, in one gram it can be “I”, “love”, “history”, “books”, “and”, “watching”, “documentaries”. In two-gram or the bi-gram, there is the two-word sequence i.e. “I love”, “love reading”, or “history books”. In the three-gram or the tri-gram, there are three words sequences i.e. “I love reading”, “history books,” or “and watching documentaries”.

An issue when using n-gram language models are out-of-vocabulary (OOV) words. They are encountered in computational linguistics and natural language processing when the input includes words which were not present in a system's dictionary or database during its preparation. By default, when a language model is estimated, the entire observed vocabulary is used. In some cases, it may be necessary to estimate the language model with a specific fixed vocabulary. In such a scenario, the n-grams in the corpus that contain an out-of-vocabulary word are ignored. The n-gram probabilities are smoothed over all the words in the vocabulary even if they were not observed.



Word-level unigrams

Text	Token Sequence	Token Value
One Two Three Four	1	One
One Two Three Four	2	Two
One Two Three Four	3	Three
One Two Three Four	4	Four

Word-level bigrams

Text	Token Sequence	Token Value
One Two Three Four	1	One Two
One Two Three Four	2	Two Three
One Two Three Four	3	Three Four

Word-level trigrams

Text	Token Sequence	Token Value
One Two Three Four	1	One Two Three
One Two Three Four	2	Two Three Four

Figure 3.2.6: N-Grams

3.2.7. LSTM

Long Short-Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997) and were refined and popularized by many people in the following work. They work tremendously well on a large variety of problems and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn. All recurrent neural networks have the form of a chain of repeating modules of neural networks. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

LSTMs deal with both Long Term Memory (LTM) and Short Term Memory (STM) and for making the calculations simple and effective it uses the concept of gates.

Forget Gate: LTM goes to forget gate and it forgets information that is not useful.

Learn Gate: Event (current input) and STM are combined together so that necessary information that we have recently learned from STM can be applied to the current input.

Remember Gate: LTM information that we haven't forget and STM and Event are combined together in Remember gate which works as updated LTM.

Use Gate: This gate also uses LTM, STM, and Event to predict the output of the current event which works as an updated STM.

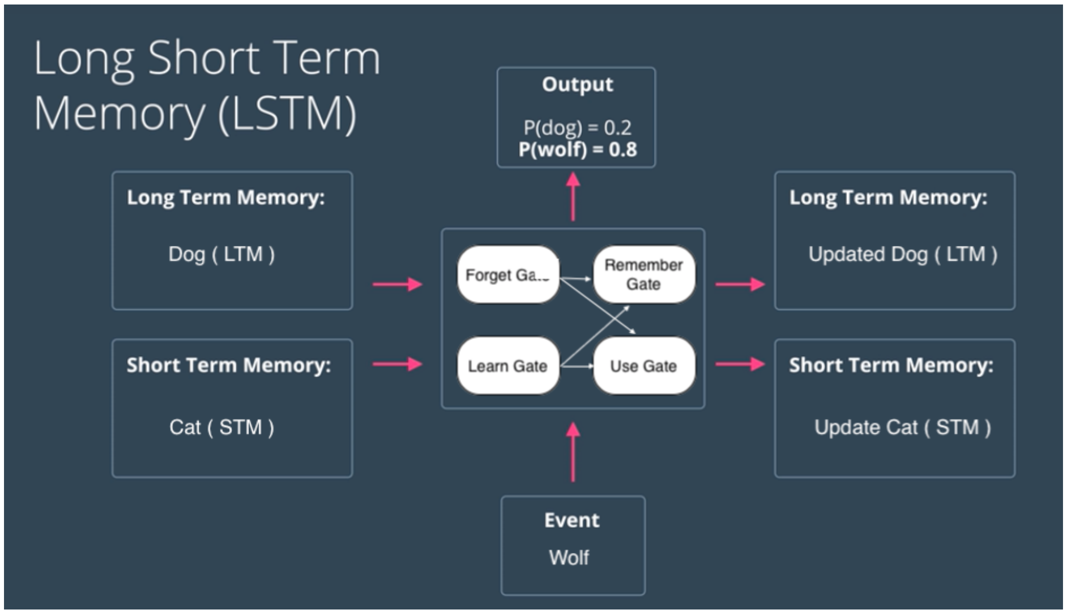


Figure 3.2.7: LSTM

3.2.8. GRU

GRU or Gated recurrent unit is an advancement of the standard RNN i.e recurrent neural network. GRUs are very similar to Long Short Term Memory(LSTM). Just like LSTM, GRU uses gates to control the flow of information. They are relatively new as compared to LSTM. This is the reason they offer some improvement over LSTM and have simpler architecture.

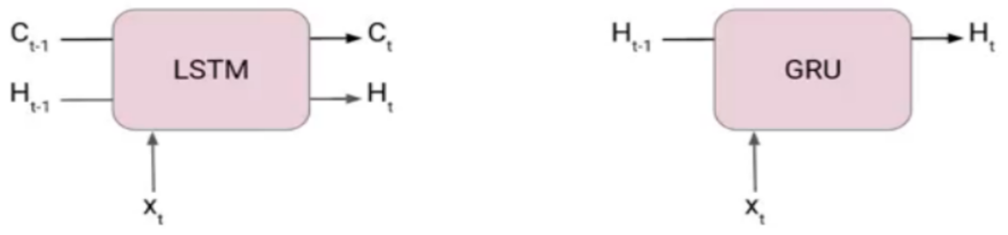


Figure 3.2.8.1: LSTM vs GRU Gate

Another Interesting thing about GRU is that, unlike LSTM, it does not have a separate cell state (Ct). It only has a hidden state(Ht). Due to the simpler architecture, GRUs are faster to train.

Architecture of GRU

Here we have a GRU cell which more or less similar to an LSTM cell or RNN cell.

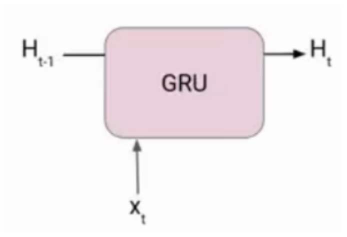


Figure 3.2.8.2: GRU Gate

At each timestamp t, it takes an input Xt and the hidden state Ht-1 from the previous timestamp t-1. Later it outputs a new hidden state Ht which again passed to the next timestamp.
Now there are primarily two gates in a GRU as opposed to three gates in an LSTM cell. The first gate is the Reset gate and the other one is the update gate.

Reset Gate (Short term memory): The Reset Gate is responsible for the short-term memory of the network i.e the hidden state (Ht). Here is the equation of the Reset gate.

$$r_t = \sigma (x_t * U_r + H_{t-1} * W_r)$$

LSTM gate equation it is very similar to that. The value of rt will range from 0 to 1 because of the sigmoid function. Here Ur and Wr are weight matrices for the reset gate.

Update Gate (Long Term memory): Similarly, we have an Update gate for long-term memory and the equation of the gate is shown below.

$$u_t = \sigma (x_t * U_u + H_{t-1} * W_u)$$

The only difference is of weight metrics i.e Uu and Wu.

3.2.9. Flask

Flask is a web framework, it's a Python module that lets you develop web applications easily. It's has a small and easy-to-extend core: it's a microframework that doesn't include an ORM (Object Relational Manager) or such features.

3.3. UML Diagram

UML is an acronym that stands for Unified Modeling Language. Simply put, UML is a modern approach to modeling and documenting software. In fact, it's one of the most popular business process modeling techniques. It is based on diagrammatic representations of software components. A UML diagram with the purpose of visually representing a system along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system.

3.3.1. Sequence Diagram

A sequence diagram is the most commonly used interaction diagram. Interaction diagram is used to show the interactive behaviour of a system. Since visualizing the interactions in a system can be a cumbersome task, we use different types of interaction diagrams to capture various features and aspects of interaction in a system. A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

One of the primary uses of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement. Use cases are often refined into one or more sequence diagrams. In addition to their use in designing new systems, sequence diagrams can be used to document how objects in an existing system currently interact. This documentation is very useful when transitioning a system to another person or organization.

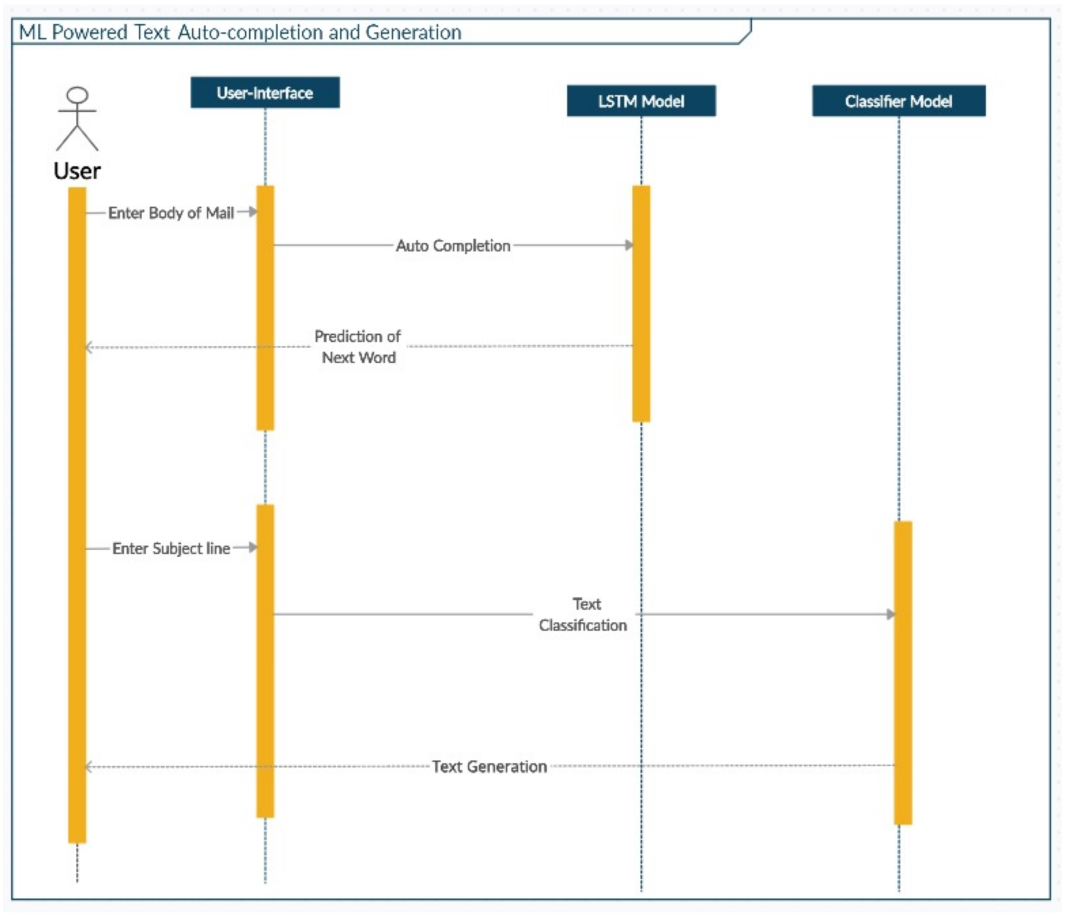


Figure 3.3.1: Sequence Diagram

In our model, when the user starts typing in the body section of the mail compose, he activates the auto-completion function which generates the next word and if the user does start composing from the subject line, then the text classification is done and the body of the mail would be generated for editing.

3.3.2. Use Case Diagram

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system. The following picture depicts the Use case diagram of our model, ML Powered Text Auto-completion and Generation.

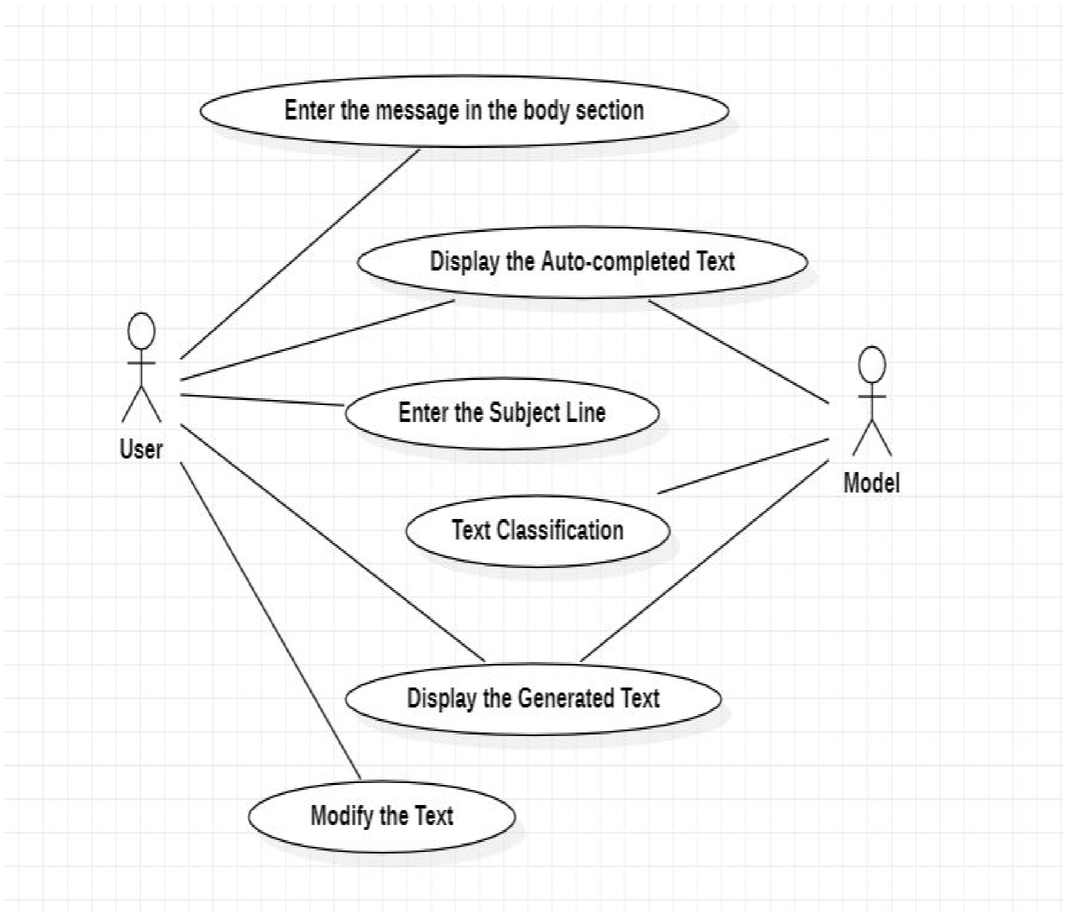

Enter the message in the body section
Display the Auto-completed Text
User
Enter the Subject Line
Model
Text Classification
Display the Generated Text
Modify the Text

Figure 3.3.2: Use Case Diagram

3.4. Software and Hardware Requirements

Hardware

i) Operating System: Windows 10

ii) Processor - Intel Core i5

iii) Memory(RAM) - 8 GB

iv) Hard disk: 1TB

Software

i) Python 3.5 and later versions

ii) IDE: Google Colab, VS Code

4. Modules

Modules are important to have a precise overview of the development of the project process so that while execution, clarity of the next step is maintained. Our project has the following modules

1. Data Pre-processing
2. Building the model
3. Training the model
4. Testing the model

1. Data Pre-processing

Data pre-processing is the process of transforming raw data into an understandable format. It is an important step in data mining as we cannot work with raw data. The quality of the data should be checked before applying machine learning algorithms. The main agenda is the model to be accurate and precise in predictions and should be able to easily interpret the data's features.

During the pre-processing, the data is cleaned and transformed so that the model can understand the data. In our project, we are using the Enron dataset.The Enron email dataset contains approximately 500,000 emails generated by employees of the Enron Corporation. It was obtained by the Federal Energy Regulatory Commission during its investigation of Enron's collapse. The Enron Email Corpus is one of the biggest email data sources in the world. Almost half a million files are spread over 2.5 GB. Now for pre-processing, we create a simple pipeline function that are:

1. Splits the datasets by the \n character
2. Remove leading and trailing spaces
3. Drop empty sentences.
4. Tokenize sentences using nltk.word_tokenize

We pre-process the text data to make it feedable to our neural network. We use recurrent neural networks, it is the smart way to deal with text pre-processing is typically to use an embedding layer that translates words into vectors. However, text embedding is unsuitable for this task since our goal is to build a character-level text generation model. In other words, our model is not going to generate word predictions; instead, it will spit out a character each prediction cycle. Therefore, we will use an alternative technique, namely mapping each character to an integer value. This isn't as elegant as text embedding or even one-hot encoding but for a character-level analysis, it should work fine. The preprocess_split function takes a string text data as input and returns a list of training data, each of length max_len, sampled every step characters. It also returns the training labels and a hash table mapping characters to their respective integer encoding.

2. Building the model

To build the model we have used the Sequential model of keras library. A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor. You can create a Sequential model by passing a list of layers to the Sequential constructor. We can also create a Sequential model incrementally via the add() method.

Our model has 5 layers. They are

1. Embedding layer which takes 3 arguments, the input dimension which is the vocabulary size, the output dimension of size of the vector space in which words will be embedded and input length that is 3 words.
2. Encoder LSTM Layer with number of units equal to dimensions of the output space and return_sequences = true implies that the next layer is an LSTM layer.
3. Decoder LSTM Layer where return_sequences = false.
4. Dense layer which takes the number of units and activation function as arguments. Here, activation function = relu which is rectified linear unit that converts -ve values to zero and does nothing with the +ve values.
5. Dense layer which has number of units as it's vocabulary size, as the output needs to be in the vocabulary. Here, activation function = soft max which scales the numbers into probabilities.

3. Training the model

We trained the model using loss and optimizer functions where the loss function is categorical_crossentropy which is used for multi-class classification task that is used when we have several possible outputs and optimizer is adam which is to update the weights iteratively based on training data and learning rate = 0.001 which indicates the amount that the weights are updated during training.

We used ModelCheckpoint callback in conjunction with training using model.fit() to save the model. And we have fixed the argument savesbestonly as true which only saves the best model.

While fitting the model we set the epoch count as 35 which indicates the no. of iterations over the entire data and batch size as 64 which is the number of training samples in one forward/backward pass

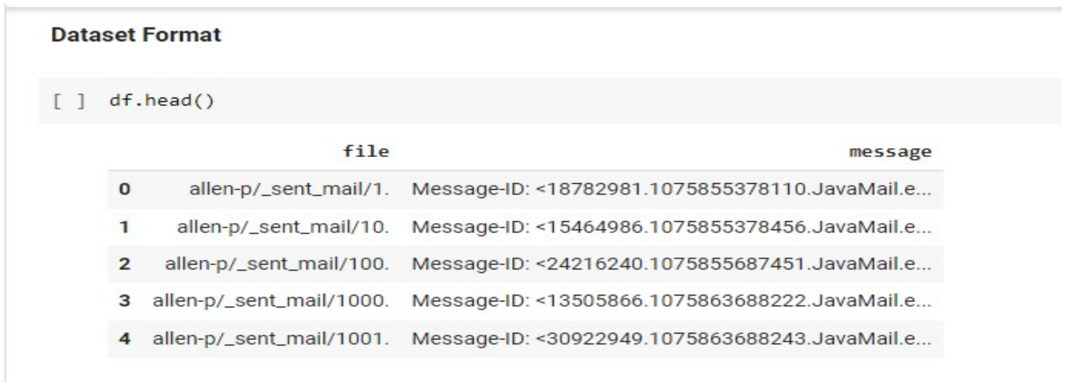
4. Testing the model

To test the model, we require 3 arguments, the trained model, tokenizer and the input text which is 3 words. Then the model first converts the text to sequence and then from sequence to array and applies the argmax function which returns the indices of the maximum value that corresponds to the next word.

5. Implementation

As part of the implementation of the project, we implemented and built 2 features, Text Auto-completion Model and Text Generation model. For building the Text Auto-Completion model, we used LSTM and N-Grams models. LSTM stands for long short-term memory, it is a type of recurrent neural network (RNN). LSTMs are predominately used to learn, process, and classify sequential data because these networks can learn long-term dependencies between time steps of data. Common LSTM applications include sentiment analysis, language modeling, speech recognition, and video analysis. N-Gram models are Statistical (Probabilistic) Language models that aim to assign probabilities to a given sequence of words. An N-gram model is built by counting how often word sequences occur in corpus text and then estimating the probabilities. An N-gram model is one type of a Language Model (LM), which is about finding the probability distribution over word sequences. For building the Text Generation model, we have used GRU model. GRU stands for Gated Recurrent Unit, it is a type of recurrent neural network (RNN). This is much similar to LSTM, but due to its simple architecture it works faster.

5 Code

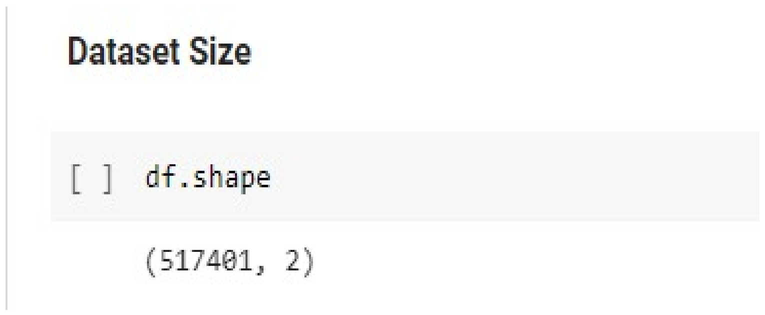


Dataset Format

```
[ ] df.head()
```

	file	message
0	allen-p/_sent_mail/1.	Message-ID: <18782981.1075855378110.JavaMail.e...
1	allen-p/_sent_mail/10.	Message-ID: <15464986.1075855378456.JavaMail.e...
2	allen-p/_sent_mail/100.	Message-ID: <24216240.1075855687451.JavaMail.e...
3	allen-p/_sent_mail/1000.	Message-ID: <13505866.1075863688222.JavaMail.e...
4	allen-p/_sent_mail/1001.	Message-ID: <30922949.1075863688243.JavaMail.e...

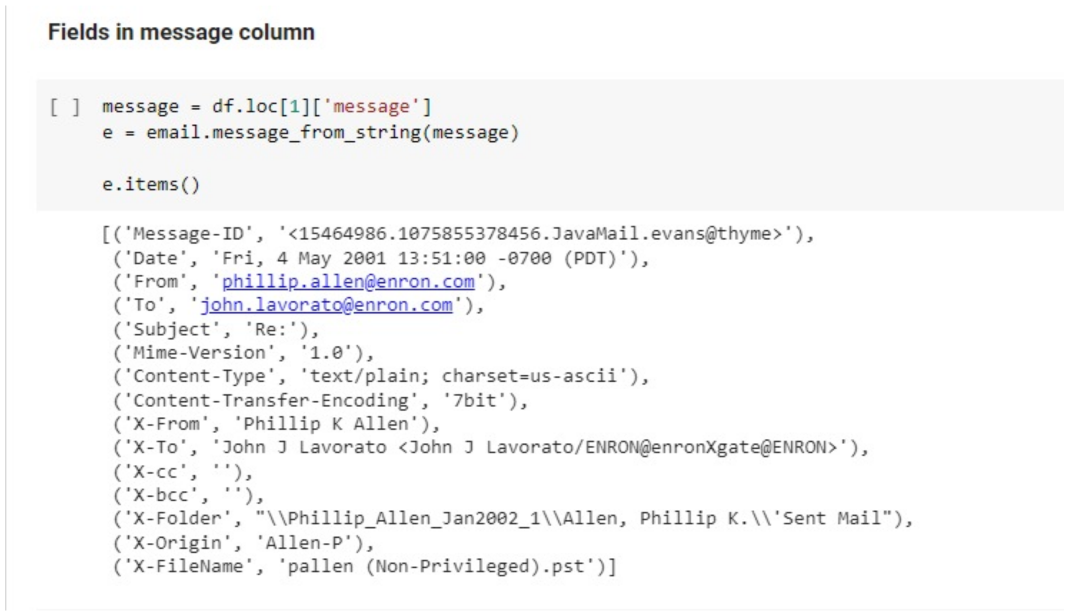
Figure 5.1: Dataset Format used for Text Auto-Completion



Dataset Size

```
[ ]  df.shape

     (517401, 2)
```

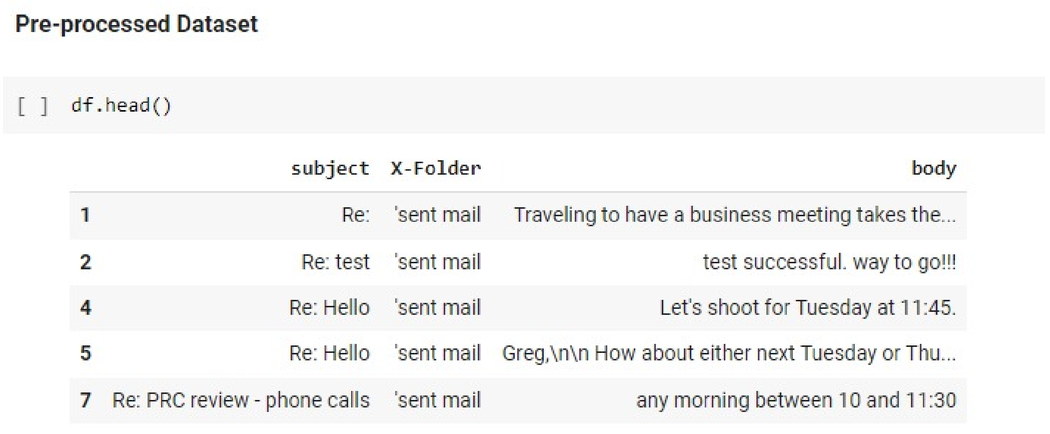
Figure 5.2: Dataset Shape



Fields in message column

```
[ ]  message = df.loc[1]['message']
     e = email.message_from_string(message)

     e.items()

     [('Message-ID', '<15464986.1075855378456.JavaMail.evans@thyme>'),
      ('Date', 'Fri, 4 May 2001 13:51:00 -0700 (PDT)'),
      ('From', 'phillip.allen@enron.com'),
      ('To', 'john.lavorato@enron.com'),
      ('Subject', 'Re:'),
      ('Mime-Version', '1.0'),
      ('Content-Type', 'text/plain; charset=us-ascii'),
      ('Content-Transfer-Encoding', '7bit'),
      ('X-From', 'Phillip K Allen'),
      ('X-To', 'John J Lavorato <John J Lavorato/ENRON@enronXgate@ENRON>'),
      ('X-cc', ''),
      ('X-bcc', ''),
      ('X-Folder', "\\Phillip_Allen_Jan2002_1\\Allen, Phillip K.\\'Sent Mail"),
      ('X-Origin', 'Allen-P'),
      ('X-FileName', 'pallen (Non-Privileged).pst')]
```

Figure 5.3: Actual Data Format

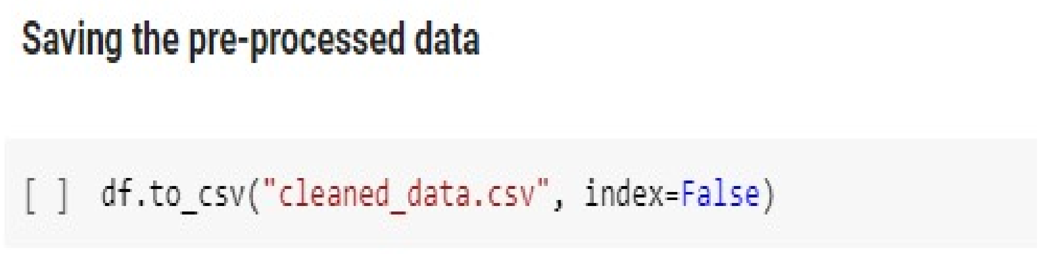


Pre-processed Dataset

```
[ ]  df.head()
```

	subject	X-Folder	body
1	Re:	'sent mail	Traveling to have a business meeting takes the...
2	Re: test	'sent mail	test successful. way to go!!!
4	Re: Hello	'sent mail	Let's shoot for Tuesday at 11:45.
5	Re: Hello	'sent mail	Greg,\n\n How about either next Tuesday or Thu...
7	Re: PRC review - phone calls	'sent mail	any morning between 10 and 11:30

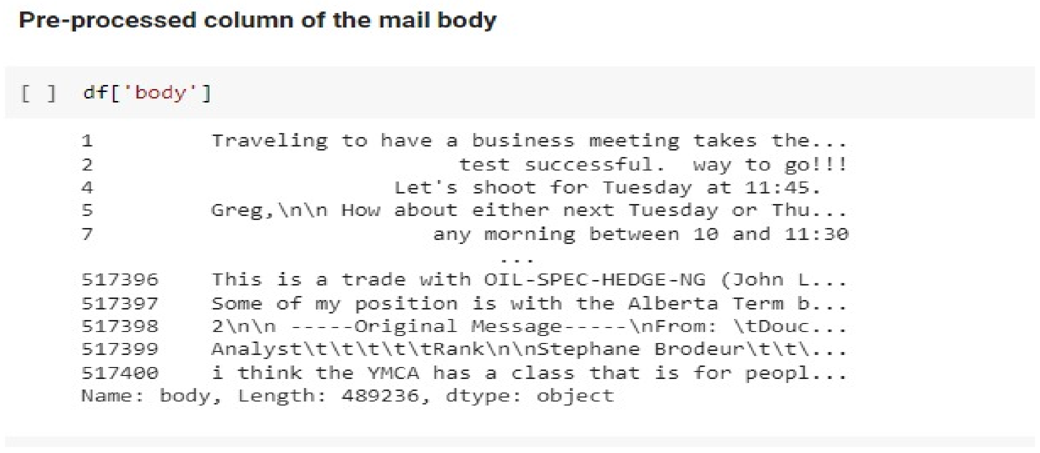
Figure 5.4: Dataset after pre-processing



Saving the pre-processed data

```
[ ]  df.to_csv("cleaned_data.csv", index=False)
```

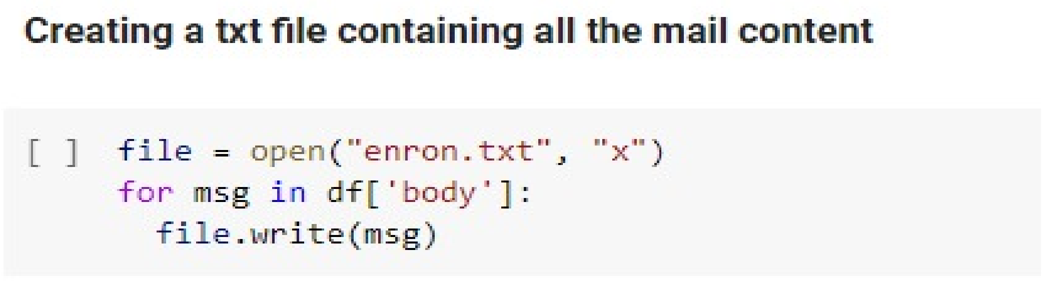
Figure 5.5: Saving the pre-processed data



Pre-processed column of the mail body

```
[ ]  df['body']

     1         Traveling to have a business meeting takes the...
     2                            test successful.  way to go!!!
     4                      Let's shoot for Tuesday at 11:45.
     5         Greg,\n\n How about either next Tuesday or Thu...
     7                           any morning between 10 and 11:30
                                          ...
     517396    This is a trade with OIL-SPEC-HEDGE-NG (John L...
     517397    Some of my position is with the Alberta Term b...
     517398    2\n\n -----Original Message-----\nFrom: \tDouc...
     517399    Analyst\t\t\t\t\tRank\n\nStephane Brodeur\t\t\...
     517400    i think the YMCA has a class that is for peopl...
     Name: body, Length: 489236, dtype: object
```

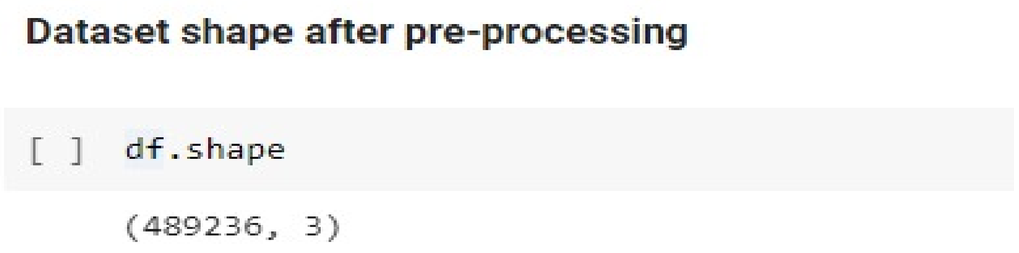
Figure 5.6: Mail body contents column



Creating a txt file containing all the mail content

```
[ ]  file = open("enron.txt", "x")
     for msg in df['body']:
       file.write(msg)
```

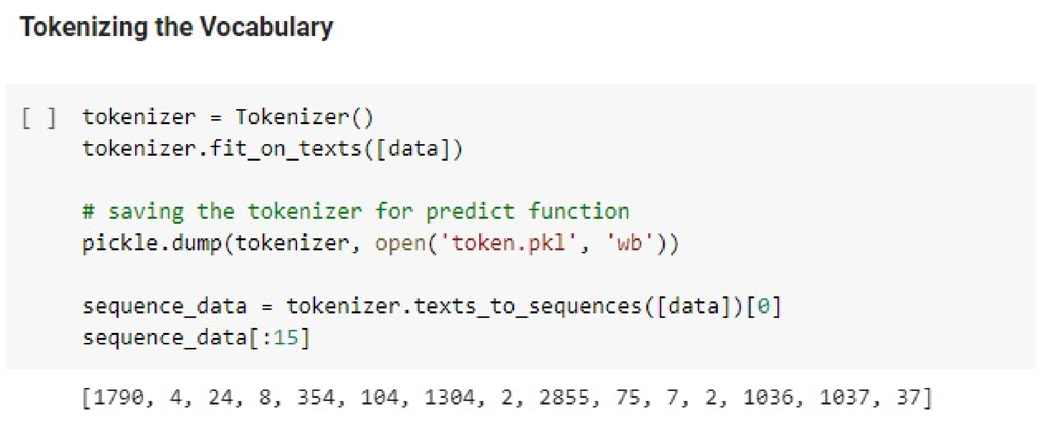
Figure 5.7: Storing the mail content in a txt file



Dataset shape after pre-processing

```
[ ]  df.shape

     (489236, 3)
```

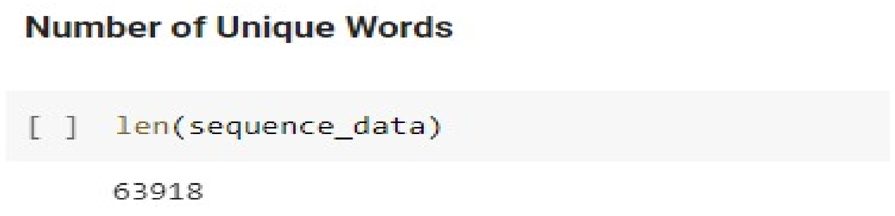
Figure 5.8: Dataset shape after pre-processing



Tokenizing the Vocabulary

```
[ ]  tokenizer = Tokenizer()
     tokenizer.fit_on_texts([data])

     # saving the tokenizer for predict function
     pickle.dump(tokenizer, open('token.pkl', 'wb'))

     sequence_data = tokenizer.texts_to_sequences([data])[0]
     sequence_data[:15]

     [1790, 4, 24, 8, 354, 104, 1304, 2, 2855, 75, 7, 2, 1036, 1037, 37]
```

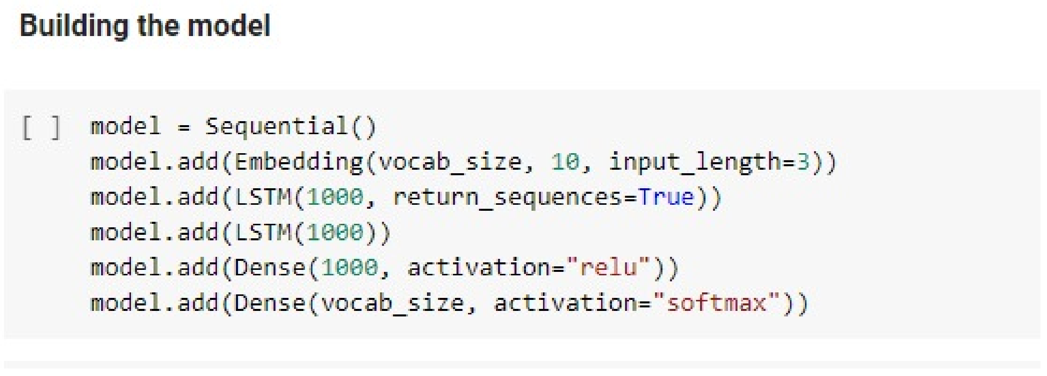
Figure 5.9: Tokenizing the vocabulary



Number of Unique Words

```
[ ]  len(sequence_data)

     63918
```

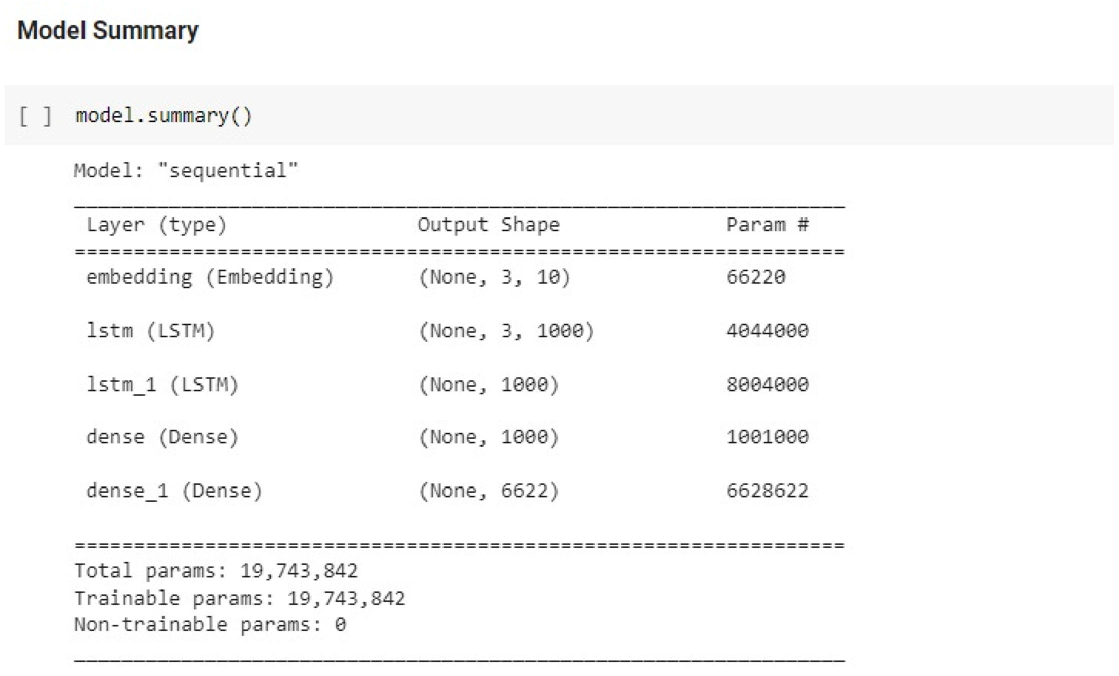
Figure 5.10: Number of unique words



Building the model

```
[ ]  model = Sequential()
     model.add(Embedding(vocab_size, 10, input_length=3))
     model.add(LSTM(1000, return_sequences=True))
     model.add(LSTM(1000))
     model.add(Dense(1000, activation="relu"))
     model.add(Dense(vocab_size, activation="softmax"))
```

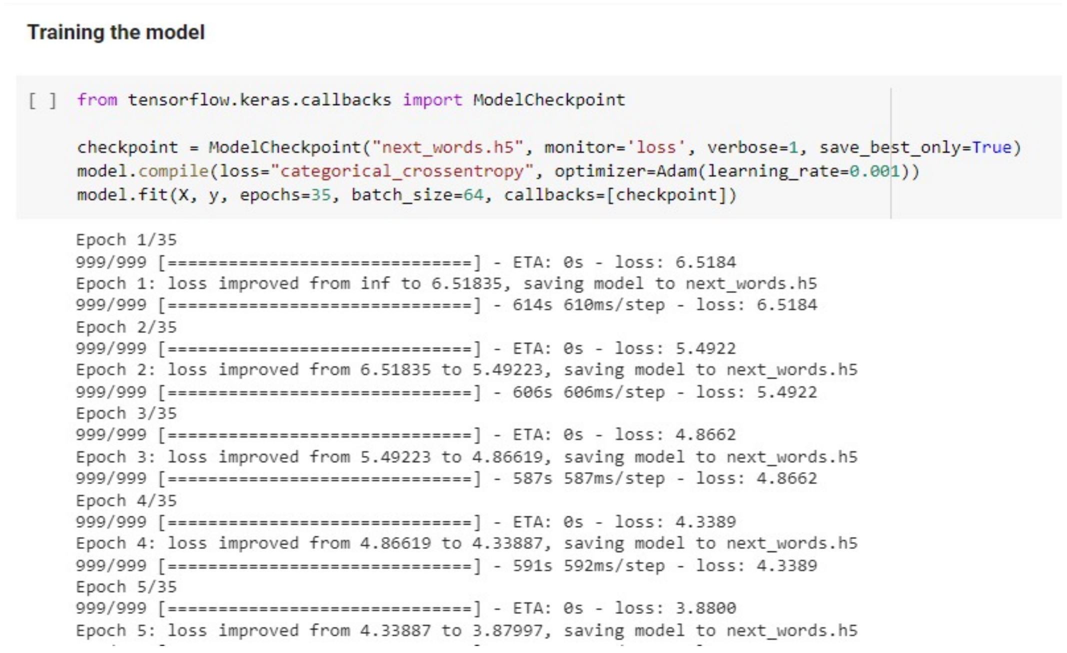
Figure 5.11: Building the model



Model Summary

```
[ ]  model.summary()

     Model: "sequential"
     _________________________________________________________________
      Layer (type)                Output Shape              Param #
     =================================================================
      embedding (Embedding)       (None, 3, 10)             66220

      lstm (LSTM)                 (None, 3, 1000)           4044000

      lstm_1 (LSTM)               (None, 1000)              8004000

      dense (Dense)               (None, 1000)              1001000

      dense_1 (Dense)             (None, 6622)              6628622

     =================================================================
     Total params: 19,743,842
     Trainable params: 19,743,842
     Non-trainable params: 0
     _________________________________________________________________
```

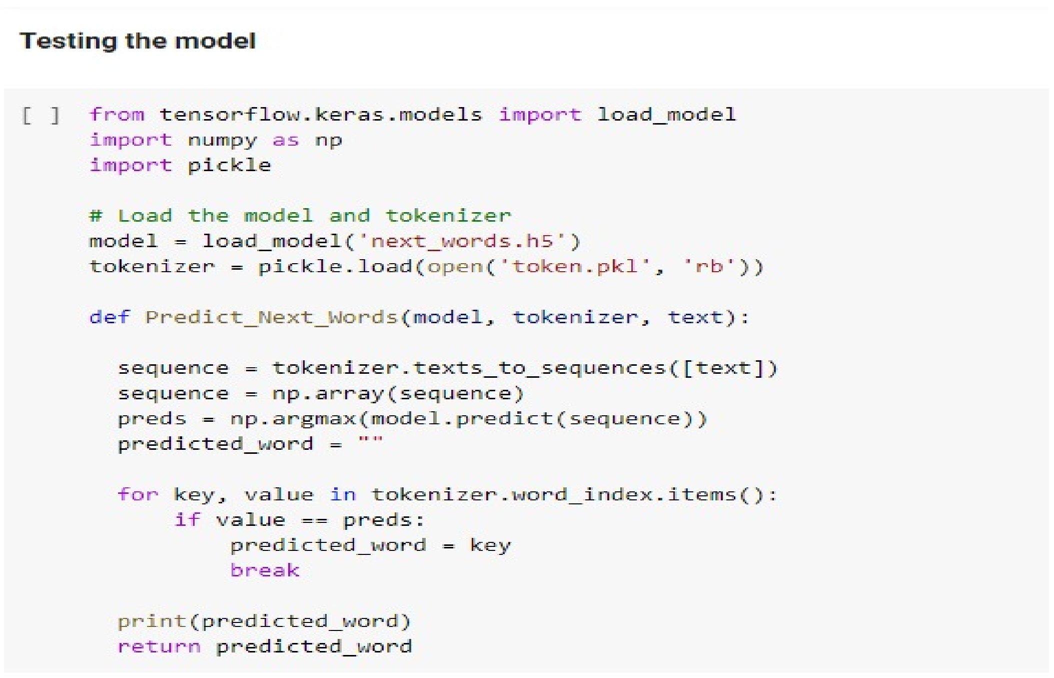
Figure 5.12: Model Summary



Training the model

```
[ ]  from tensorflow.keras.callbacks import ModelCheckpoint

     checkpoint = ModelCheckpoint("next_words.h5", monitor='loss', verbose=1, save_best_only=True)
     model.compile(loss="categorical_crossentropy", optimizer=Adam(learning_rate=0.001))
     model.fit(X, y, epochs=35, batch_size=64, callbacks=[checkpoint])

     Epoch 1/35
     999/999 [==============================] - ETA: 0s - loss: 6.5184
     Epoch 1: loss improved from inf to 6.51835, saving model to next_words.h5
     999/999 [==============================] - 614s 610ms/step - loss: 6.5184
     Epoch 2/35
     999/999 [==============================] - ETA: 0s - loss: 5.4922
     Epoch 2: loss improved from 6.51835 to 5.49223, saving model to next_words.h5
     999/999 [==============================] - 606s 606ms/step - loss: 5.4922
     Epoch 3/35
     999/999 [==============================] - ETA: 0s - loss: 4.8662
     Epoch 3: loss improved from 5.49223 to 4.86619, saving model to next_words.h5
     999/999 [==============================] - 587s 587ms/step - loss: 4.8662
     Epoch 4/35
     999/999 [==============================] - ETA: 0s - loss: 4.3389
     Epoch 4: loss improved from 4.86619 to 4.33887, saving model to next_words.h5
     999/999 [==============================] - 591s 592ms/step - loss: 4.3389
     Epoch 5/35
     999/999 [==============================] - ETA: 0s - loss: 3.8800
     Epoch 5: loss improved from 4.33887 to 3.87997, saving model to next_words.h5
```

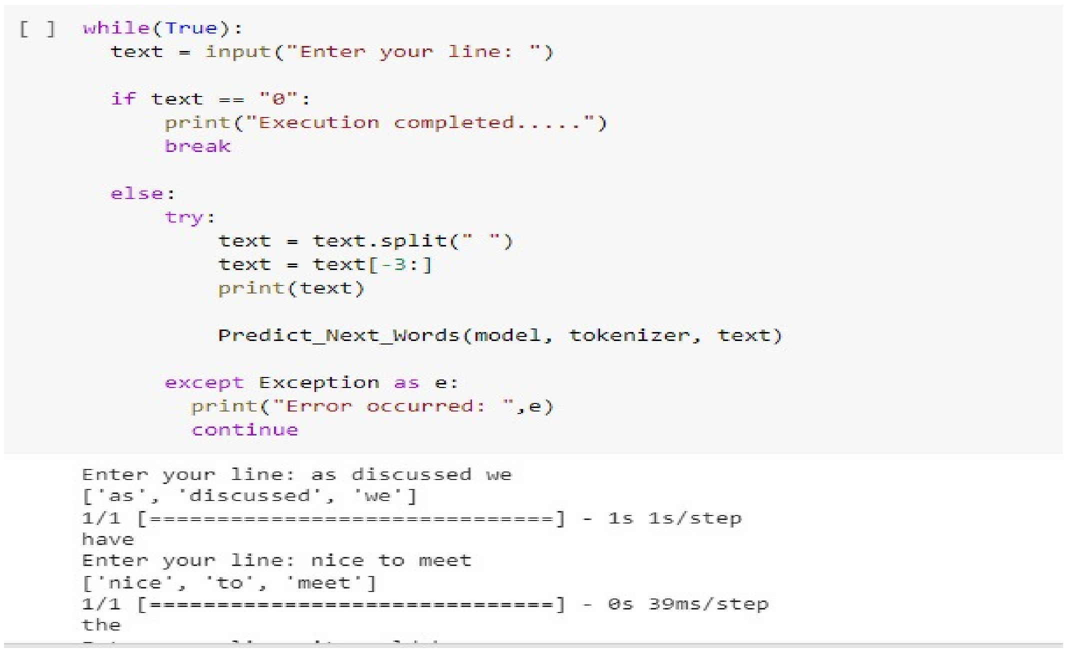
Figure 5.13: Training the model



Testing the model

```
[ ]  from tensorflow.keras.models import load_model
     import numpy as np
     import pickle

     # Load the model and tokenizer
     model = load_model('next_words.h5')
     tokenizer = pickle.load(open('token.pkl', 'rb'))

     def Predict_Next_Words(model, tokenizer, text):

       sequence = tokenizer.texts_to_sequences([text])
       sequence = np.array(sequence)
       preds = np.argmax(model.predict(sequence))
       predicted_word = ""

       for key, value in tokenizer.word_index.items():
           if value == preds:
               predicted_word = key
               break

       print(predicted_word)
       return predicted_word
```

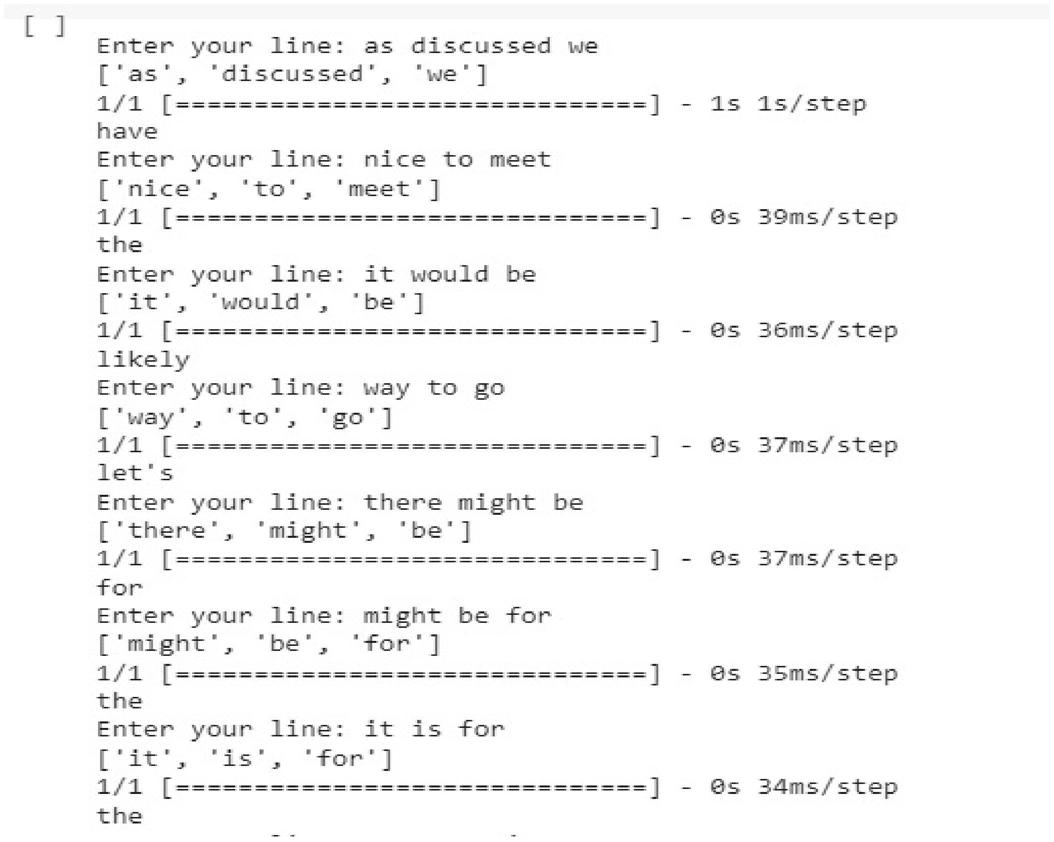
Figure 5.14: Next Word Predictor



```
[ ]  while(True):
       text = input("Enter your line: ")

       if text == "0":
           print("Execution completed.....")
           break

       else:
           try:
               text = text.split(" ")
               text = text[-3:]
               print(text)

               Predict_Next_Words(model, tokenizer, text)

           except Exception as e:
             print("Error occurred: ",e)
             continue
```

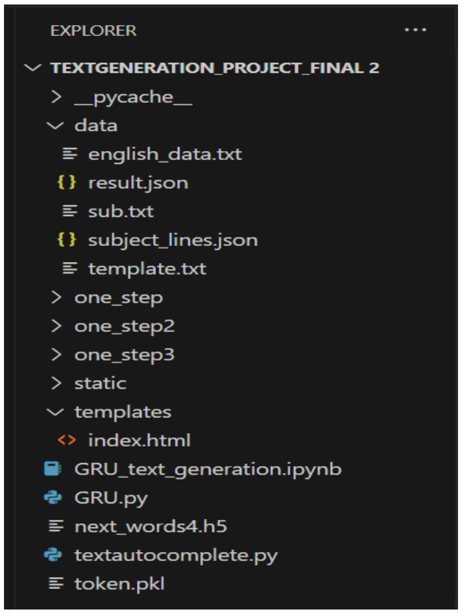
```
Enter your line: as discussed we
['as', 'discussed', 'we']
1/1 [==============================] - 1s 1s/step
have
Enter your line: nice to meet
['nice', 'to', 'meet']
1/1 [==============================] - 0s 39ms/step
the
```

Figure 5.15: Testing the model



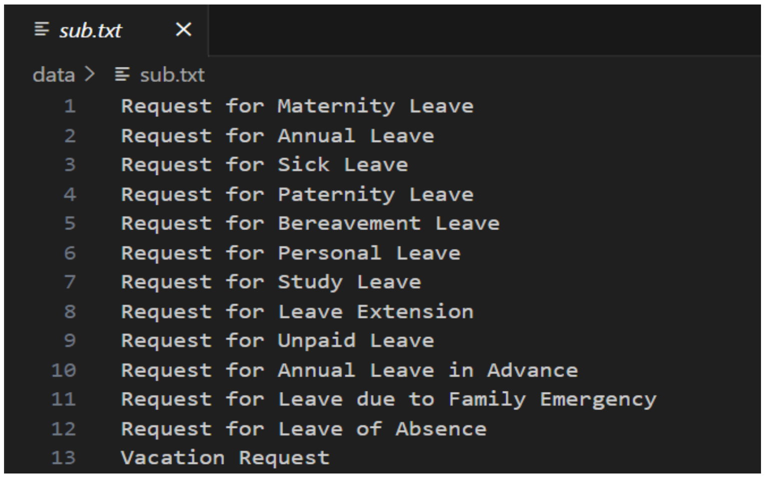
```
[ ]
     Enter your line: as discussed we
     ['as', 'discussed', 'we']
     1/1 [==============================] - 1s 1s/step
     have
     Enter your line: nice to meet
     ['nice', 'to', 'meet']
     1/1 [==============================] - 0s 39ms/step
     the
     Enter your line: it would be
     ['it', 'would', 'be']
     1/1 [==============================] - 0s 36ms/step
     likely
     Enter your line: way to go
     ['way', 'to', 'go']
     1/1 [==============================] - 0s 37ms/step
     let's
     Enter your line: there might be
     ['there', 'might', 'be']
     1/1 [==============================] - 0s 37ms/step
     for
     Enter your line: might be for
     ['might', 'be', 'for']
     1/1 [==============================] - 0s 35ms/step
     the
     Enter your line: it is for
     ['it', 'is', 'for']
     1/1 [==============================] - 0s 34ms/step
     the
```

Figure 5.16: Next Word Prediction



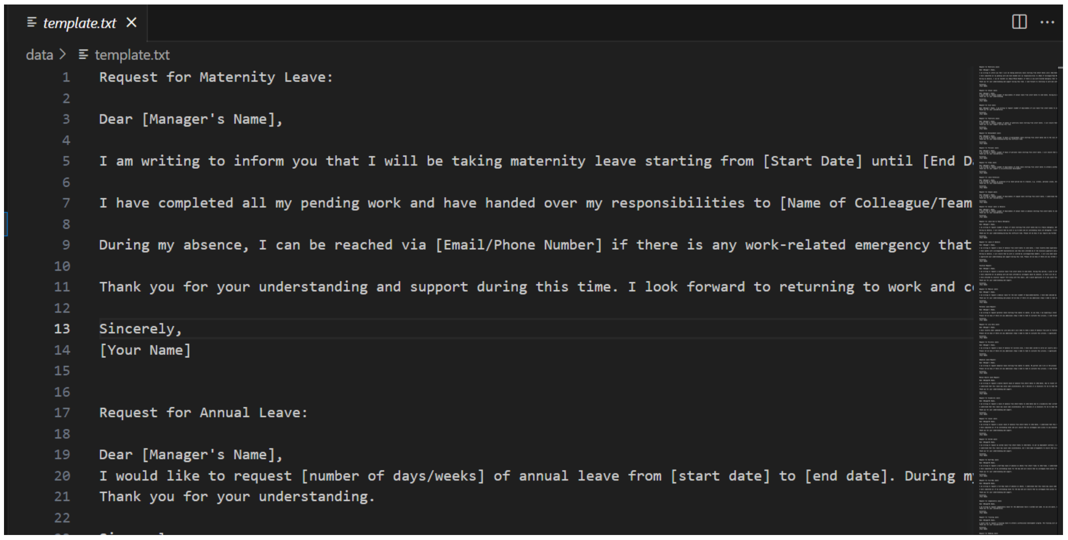
```
EXPLORER
TEXTGENERATION_PROJECT_FINAL 2
  __pycache__
  data
    english_data.txt
    result.json
    sub.txt
    subject_lines.json
    template.txt
  one_step
  one_step2
  one_step3
  static
  templates
    index.html
  GRU_text_generation.ipynb
  GRU.py
  next_words4.h5
  textautocomplete.py
  token.pkl
```

Figure 5.17: Project Structure



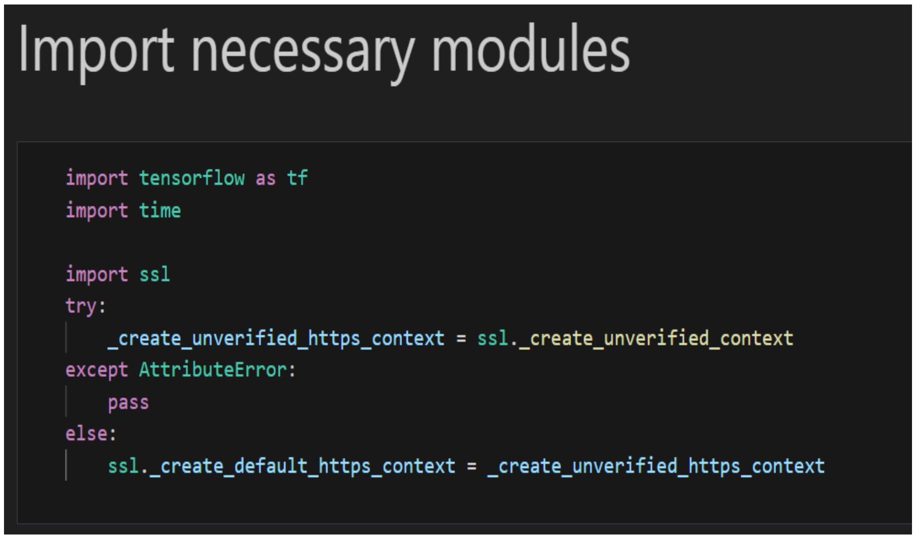
```
sub.txt
data > sub.txt
Request for Maternity Leave
Request for Annual Leave
Request for Sick Leave
Request for Paternity Leave
Request for Bereavement Leave
Request for Personal Leave
Request for Study Leave
Request for Leave Extension
Request for Unpaid Leave
Request for Annual Leave in Advance
Request for Leave due to Family Emergency
Request for Leave of Absence
Vacation Request
```

Figure 5.18: Subject Lines Format



```
template.txt ×
data > template.txt
Request for Maternity Leave:

Dear [Manager's Name],

I am writing to inform you that I will be taking maternity leave starting from [Start Date] until [End D

I have completed all my pending work and have handed over my responsibilities to [Name of Colleague/Team

During my absence, I can be reached via [Email/Phone Number] if there is any work-related emergency that

Thank you for your understanding and support during this time. I look forward to returning to work and c

Sincerely,
[Your Name]


Request for Annual Leave:

Dear [Manager's Name],
I would like to request [number of days/weeks] of annual leave from [start date] to [end date]. During m
Thank you for your understanding.

```

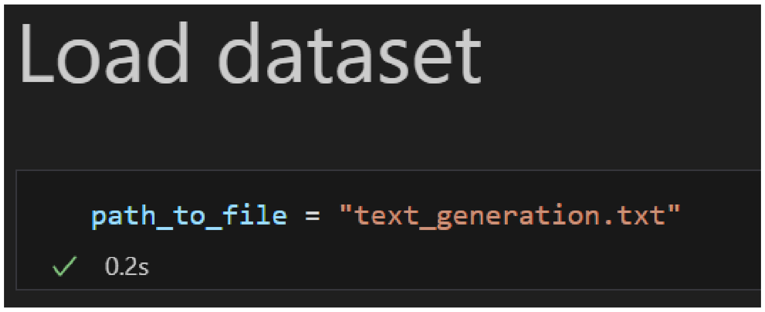
Figure 5.19: Email Template Format



Import necessary modules

```python
import tensorflow as tf
import time

import ssl
try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context
```

Figure 5.20: Importing Modules

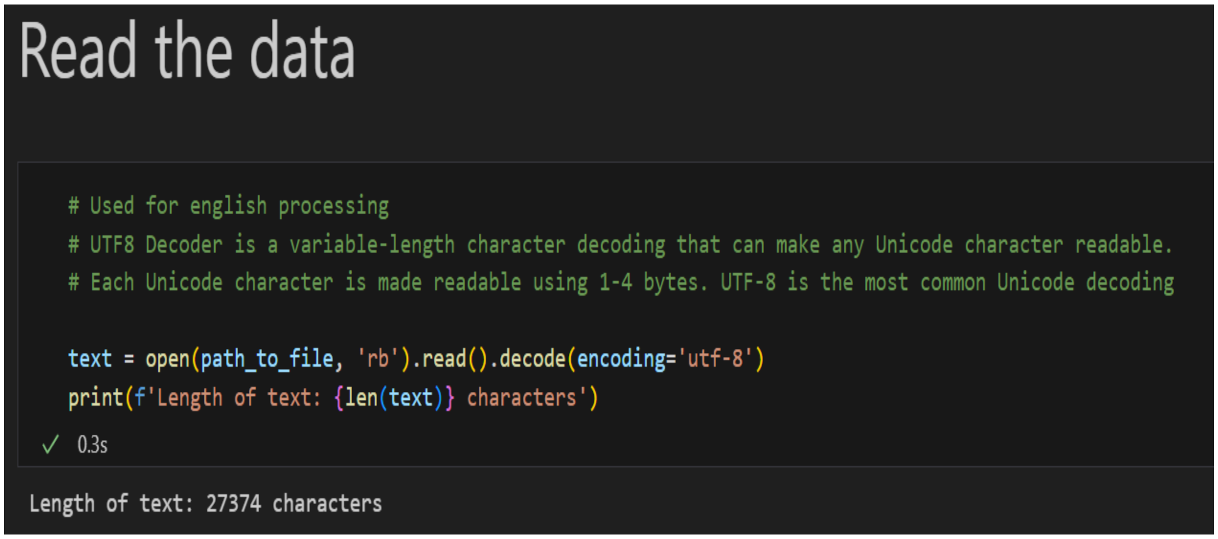


Load dataset

```
path_to_file = "text_generation.txt"
```

```
✓ 0.2s
```

Figure 5.21: Loading the Dataset



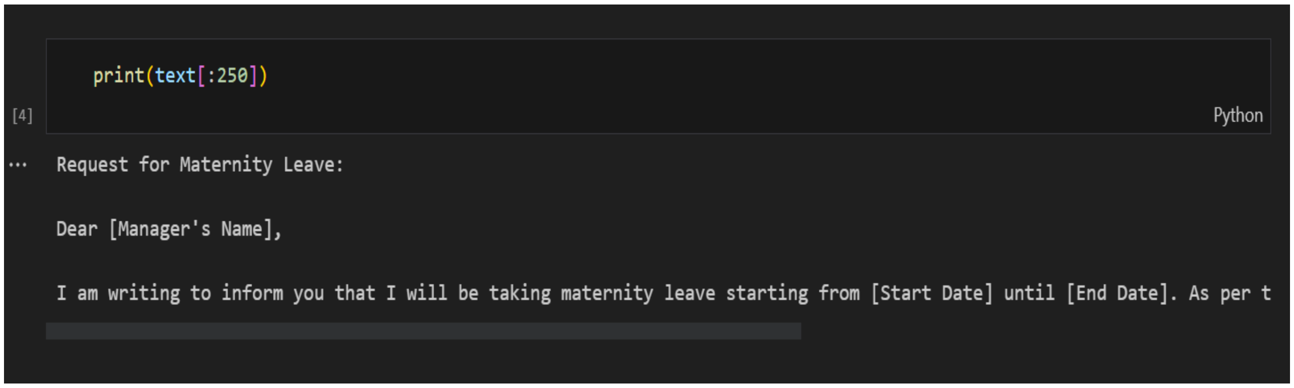
Read the data

```
# Used for english processing
# UTF8 Decoder is a variable-length character decoding that can make any Unicode character readable.
# Each Unicode character is made readable using 1-4 bytes. UTF-8 is the most common Unicode decoding

text = open(path_to_file, 'rb').read().decode(encoding='utf-8')
print(f'Length of text: {len(text)} characters')
```

```
✓ 0.3s
```

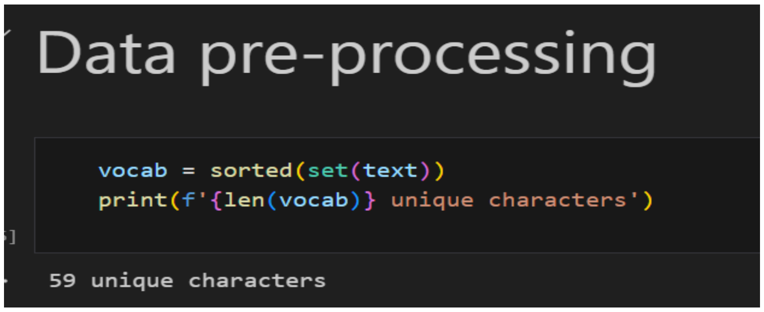
```
Length of text: 27374 characters
```

Figure 5.22: Reading the Dataset



```
[4]  print(text[:250])
                                                                    Python

···  Request for Maternity Leave:

     Dear [Manager's Name],

     I am writing to inform you that I will be taking maternity leave starting from [Start Date] until [End Date]. As per t
```

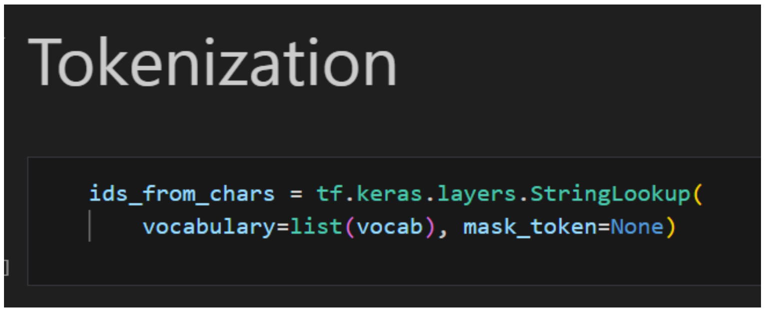
Figure 5.23: Printing the Dataset



Data pre-processing

```
vocab = sorted(set(text))
print(f'{len(vocab)} unique characters')

59 unique characters
```

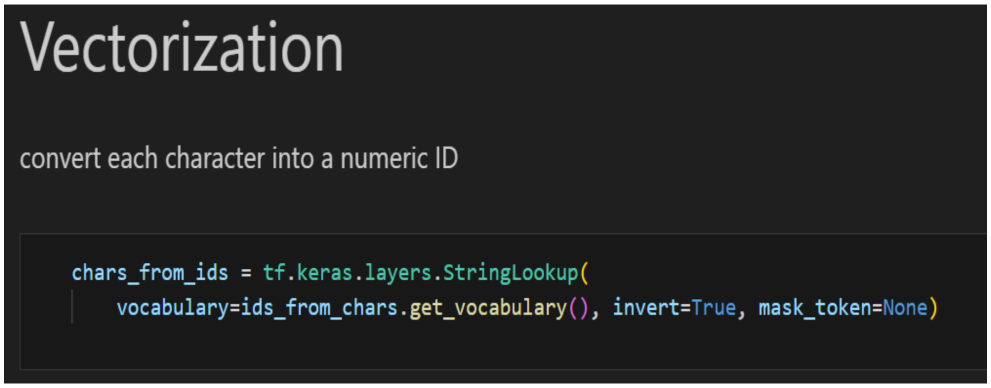
Figure 5.24: Length of Vocabulary



Tokenization

```
ids_from_chars = tf.keras.layers.StringLookup(
    vocabulary=list(vocab), mask_token=None)
```

Figure 5.25: Tokenizing the Vocabulary

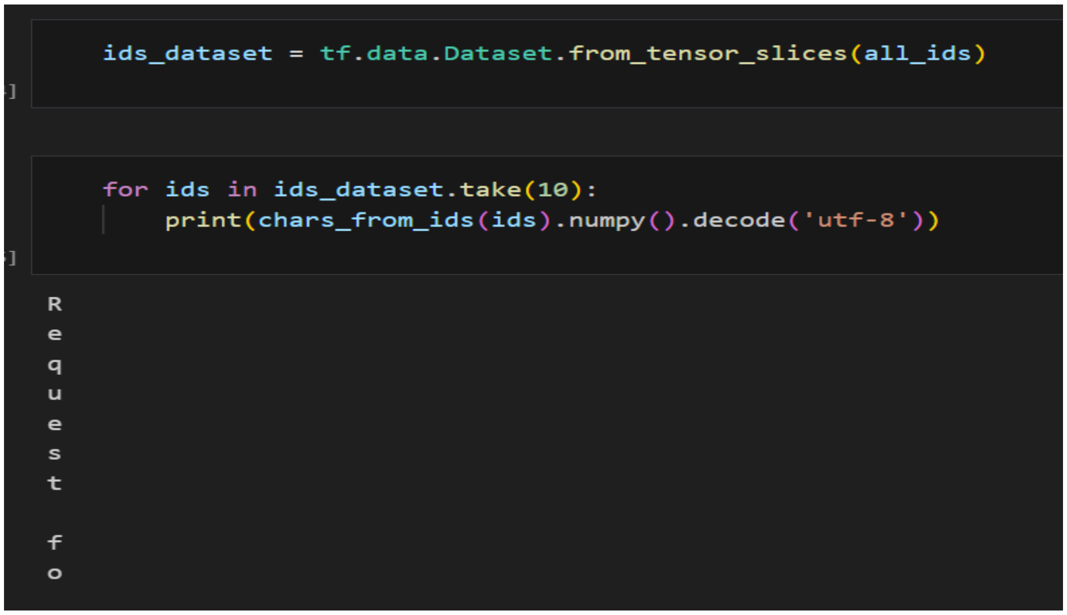


Vectorization

convert each character into a numeric ID

```
chars_from_ids = tf.keras.layers.StringLookup(
    vocabulary=ids_from_chars.get_vocabulary(), invert=True, mask_token=None)
```

Figure 5.26: Vectorizing the Vocabulary

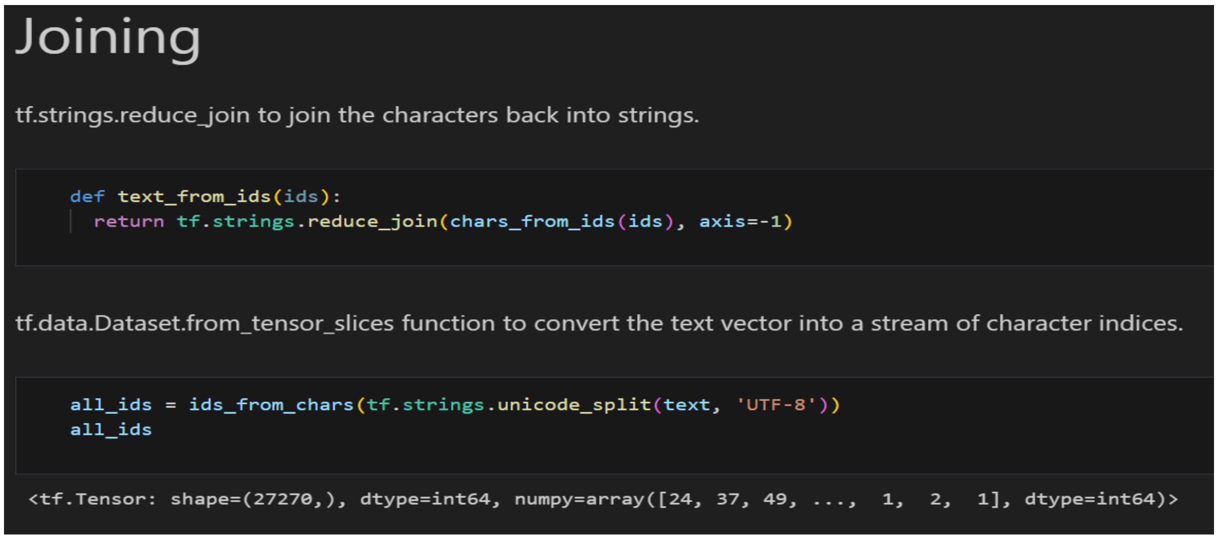


```
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
```

```
for ids in ids_dataset.take(10):
    print(chars_from_ids(ids).numpy().decode('utf-8'))
```

```
R
e
q
u
e
s
t

f
o
```

Figure 5.27: Get Characters from ids



Joining

tf.strings.reduce_join to join the characters back into strings.

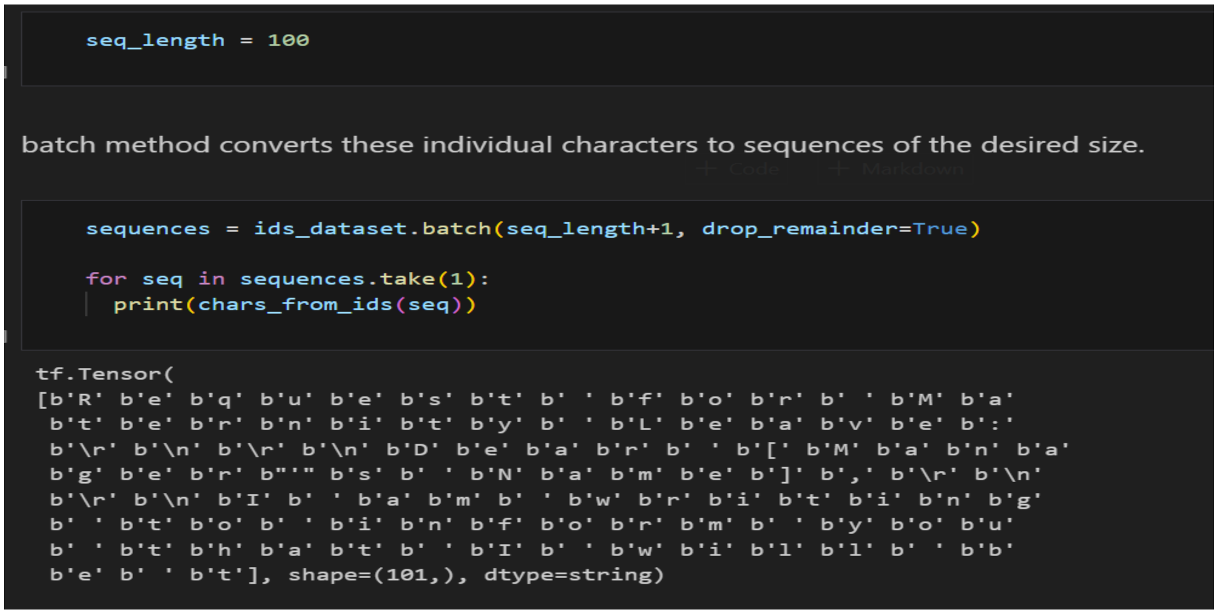
```
def text_from_ids(ids):
  return tf.strings.reduce_join(chars_from_ids(ids), axis=-1)
```

tf.data.Dataset.from_tensor_slices function to convert the text vector into a stream of character indices.

```
all_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
all_ids
```

```
<tf.Tensor: shape=(27270,), dtype=int64, numpy=array([24, 37, 49, ...,  1,  2,  1], dtype=int64)>
```

Figure 5.28: Joining the characters



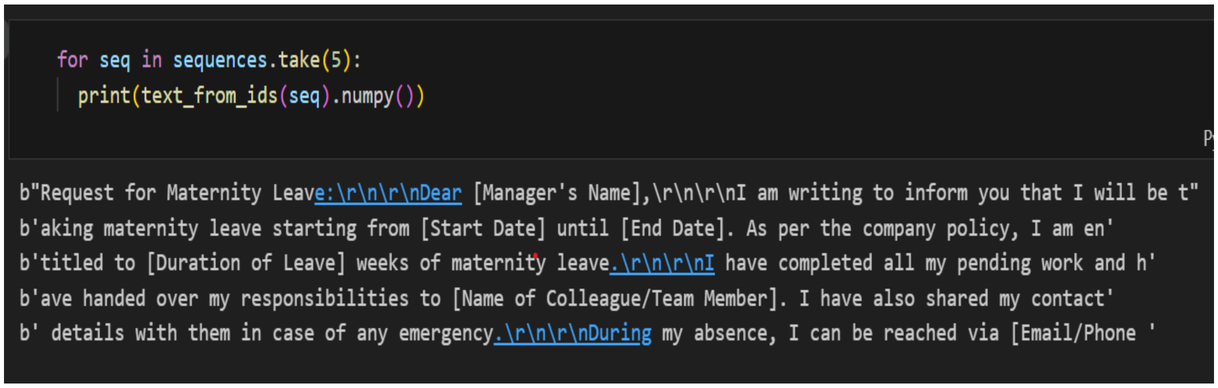
```
seq_length = 100
```

batch method converts these individual characters to sequences of the desired size.

```
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True)

for seq in sequences.take(1):
  print(chars_from_ids(seq))
```

```
tf.Tensor(
[b'R' b'e' b'q' b'u' b'e' b's' b't' b' ' b'f' b'o' b'r' b' ' b'M' b'a'
 b't' b'e' b'r' b'n' b'i' b't' b'y' b' ' b'L' b'e' b'a' b'v' b'e' b':'
 b'\r' b'\n' b'\r' b'\n' b'D' b'e' b'a' b'r' b' ' b'[' b'M' b'a' b'n' b'a'
 b'g' b'e' b'r' b"'" b's' b' ' b'N' b'a' b'm' b'e' b']' b',' b'\r' b'\n'
 b'\r' b'\n' b'I' b' ' b'a' b'm' b' ' b'w' b'r' b'i' b't' b'i' b'n' b'g'
 b' ' b't' b'o' b' ' b'i' b'n' b'f' b'o' b'r' b'm' b' ' b'y' b'o' b'u'
 b' ' b't' b'h' b'a' b't' b' ' b'I' b' ' b'w' b'i' b'l' b'l' b' ' b'b'
 b'e' b' ' b't'], shape=(101,), dtype=string)
```

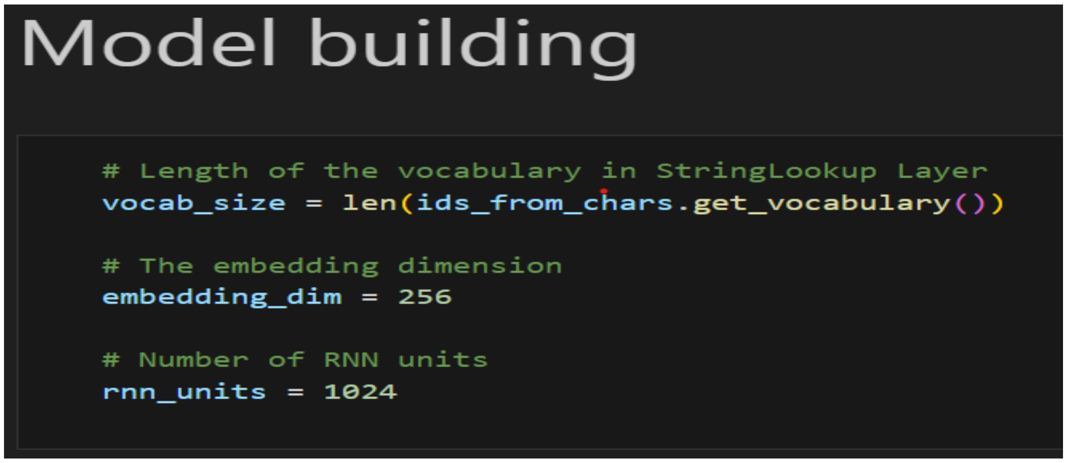
Figure 5.29: Creating Batches



```
for seq in sequences.take(5):
  print(text_from_ids(seq).numpy())
```

```
b"Request for Maternity Leave:\r\n\r\nDear [Manager's Name],\r\n\r\nI am writing to inform you that I will be t"
b'aking maternity leave starting from [Start Date] until [End Date]. As per the company policy, I am en'
b'titled to [Duration of Leave] weeks of maternity leave.\r\n\r\nI have completed all my pending work and h'
b'ave handed over my responsibilities to [Name of Colleague/Team Member]. I have also shared my contact'
b' details with them in case of any emergency.\r\n\r\nDuring my absence, I can be reached via [Email/Phone '
```

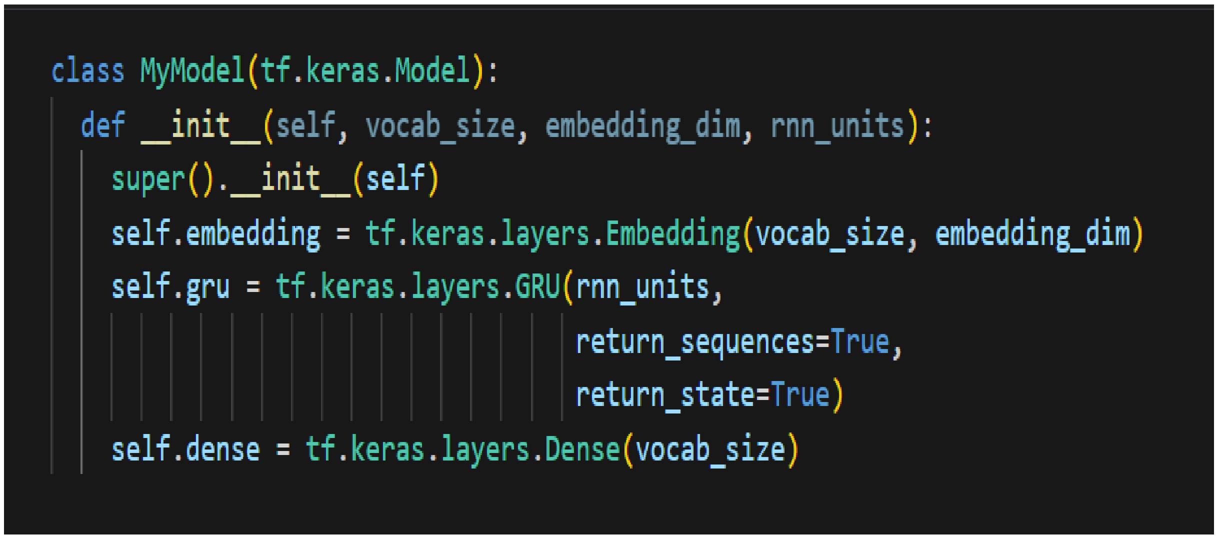
Figure 5.30: Sequences



Model building

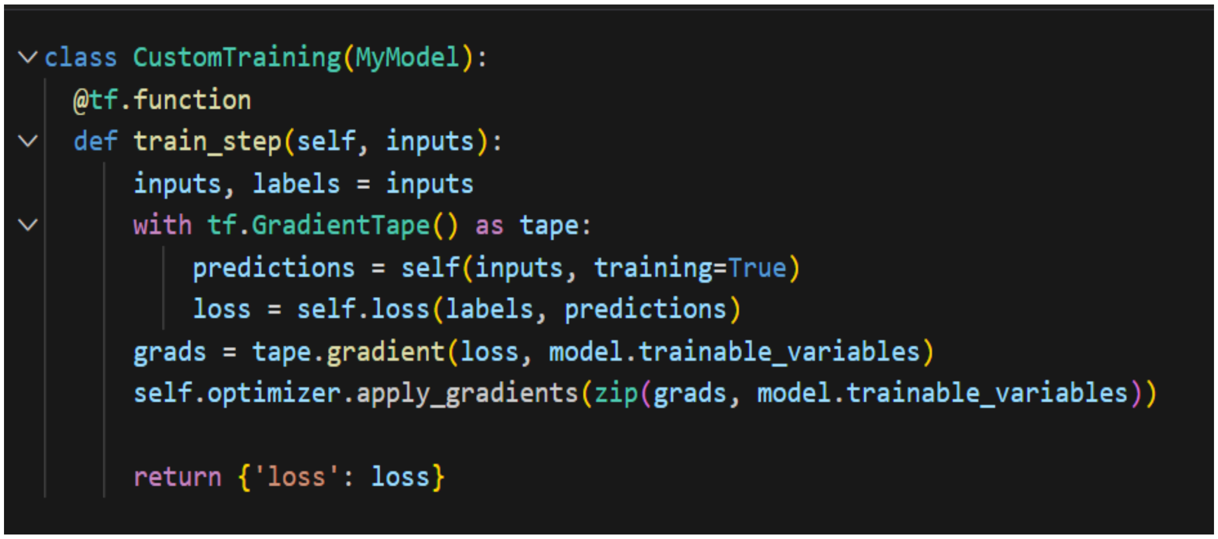
```
# Length of the vocabulary in StringLookup Layer
vocab_size = len(ids_from_chars.get_vocabulary())

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

Figure 5.31: Model Building Params



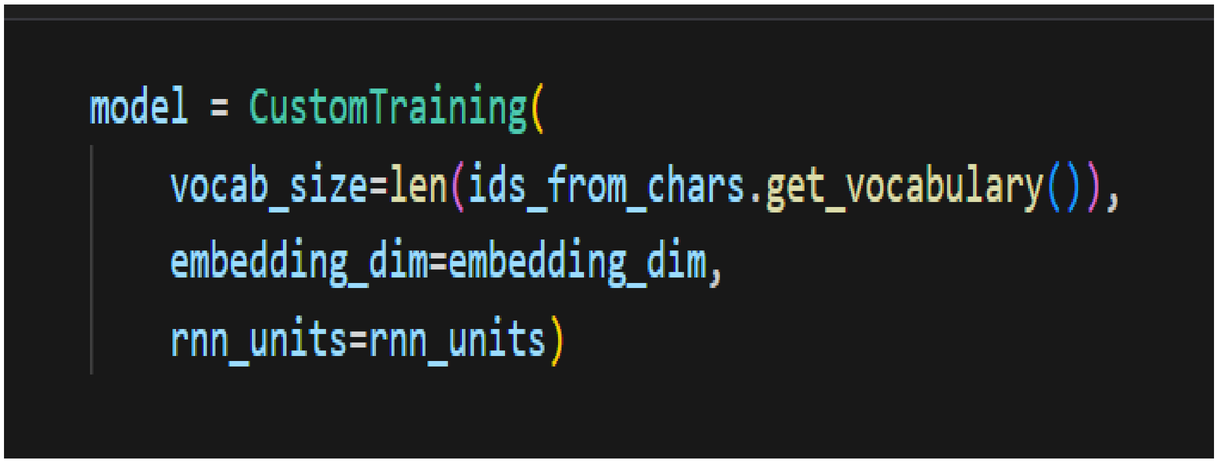
```python
class MyModel(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, rnn_units):
    super().__init__(self)
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(rnn_units,
                                   return_sequences=True,
                                   return_state=True)
    self.dense = tf.keras.layers.Dense(vocab_size)
```

Figure 5.32: Model Class



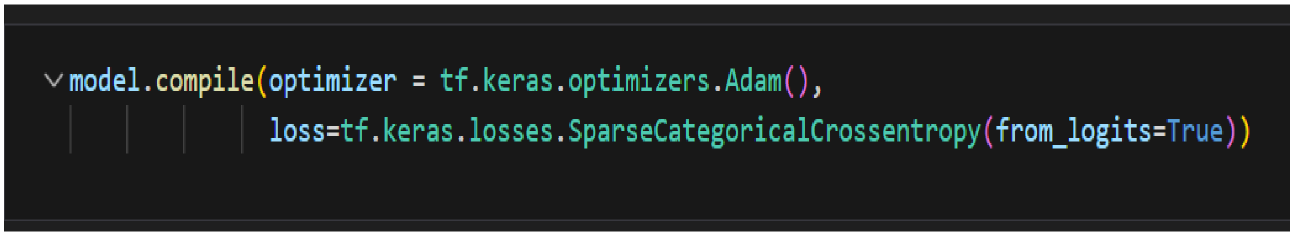
```python
class CustomTraining(MyModel):
  @tf.function
  def train_step(self, inputs):
      inputs, labels = inputs
      with tf.GradientTape() as tape:
          predictions = self(inputs, training=True)
          loss = self.loss(labels, predictions)
      grads = tape.gradient(loss, model.trainable_variables)
      self.optimizer.apply_gradients(zip(grads, model.trainable_variables))

      return {'loss': loss}
```

Figure 5.33: Custom Training Model



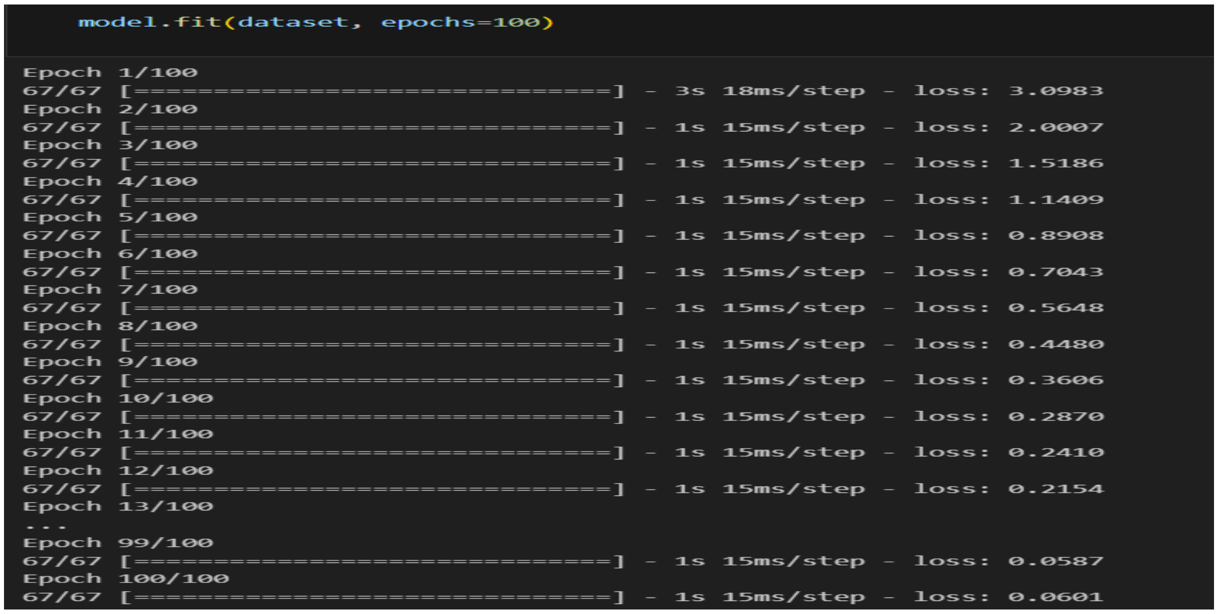
```python
model = CustomTraining(
    vocab_size=len(ids_from_chars.get_vocabulary()),
    embedding_dim=embedding_dim,
    rnn_units=rnn_units)
```

Figure 5.34: Initialize the Model



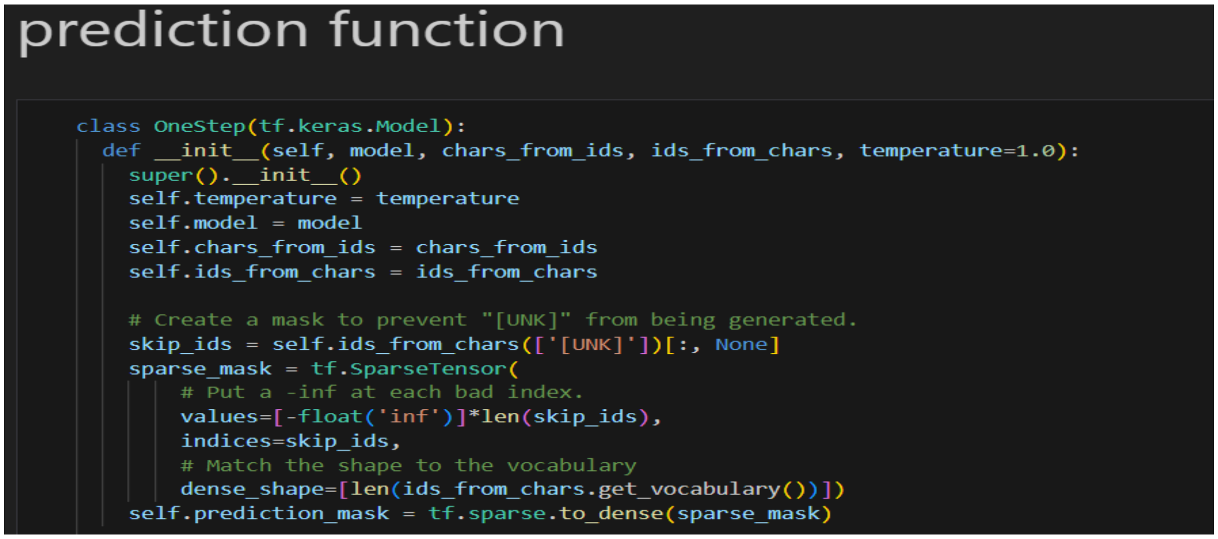
```python
model.compile(optimizer = tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

Figure 5.35: Compiling the model



```
model.fit(dataset, epochs=100)

Epoch 1/100
67/67 [==============================] - 3s 18ms/step - loss: 3.0983
Epoch 2/100
67/67 [==============================] - 1s 15ms/step - loss: 2.0007
Epoch 3/100
67/67 [==============================] - 1s 15ms/step - loss: 1.5186
Epoch 4/100
67/67 [==============================] - 1s 15ms/step - loss: 1.1409
Epoch 5/100
67/67 [==============================] - 1s 15ms/step - loss: 0.8908
Epoch 6/100
67/67 [==============================] - 1s 15ms/step - loss: 0.7043
Epoch 7/100
67/67 [==============================] - 1s 15ms/step - loss: 0.5648
Epoch 8/100
67/67 [==============================] - 1s 15ms/step - loss: 0.4480
Epoch 9/100
67/67 [==============================] - 1s 15ms/step - loss: 0.3606
Epoch 10/100
67/67 [==============================] - 1s 15ms/step - loss: 0.2870
Epoch 11/100
67/67 [==============================] - 1s 15ms/step - loss: 0.2410
Epoch 12/100
67/67 [==============================] - 1s 15ms/step - loss: 0.2154
Epoch 13/100
...
Epoch 99/100
67/67 [==============================] - 1s 15ms/step - loss: 0.0587
Epoch 100/100
67/67 [==============================] - 1s 15ms/step - loss: 0.0601
```

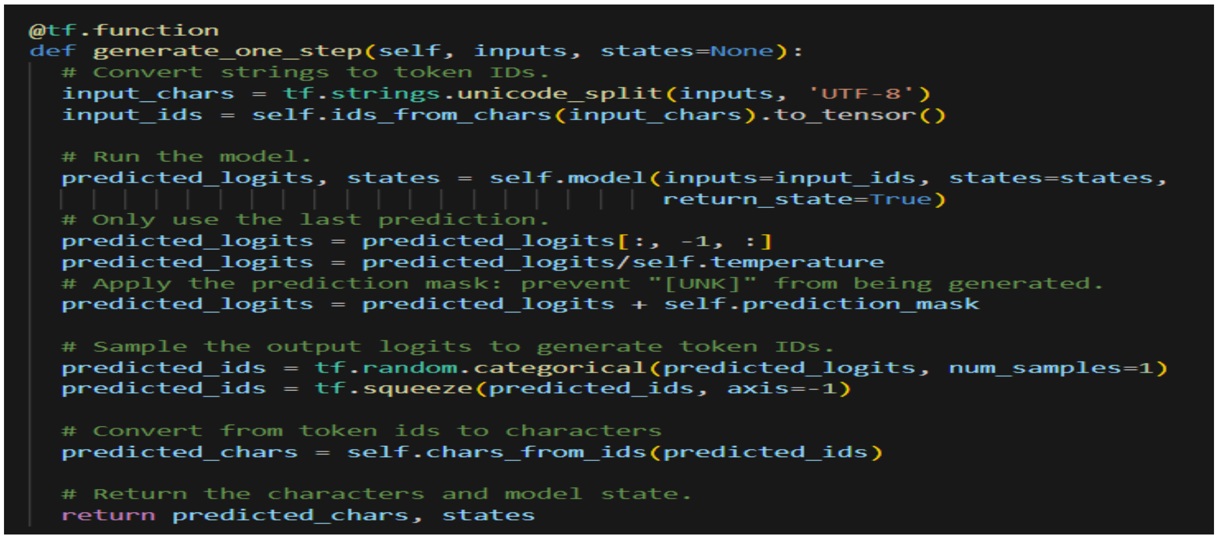
Figure 5.36: Fitting the Model



prediction function

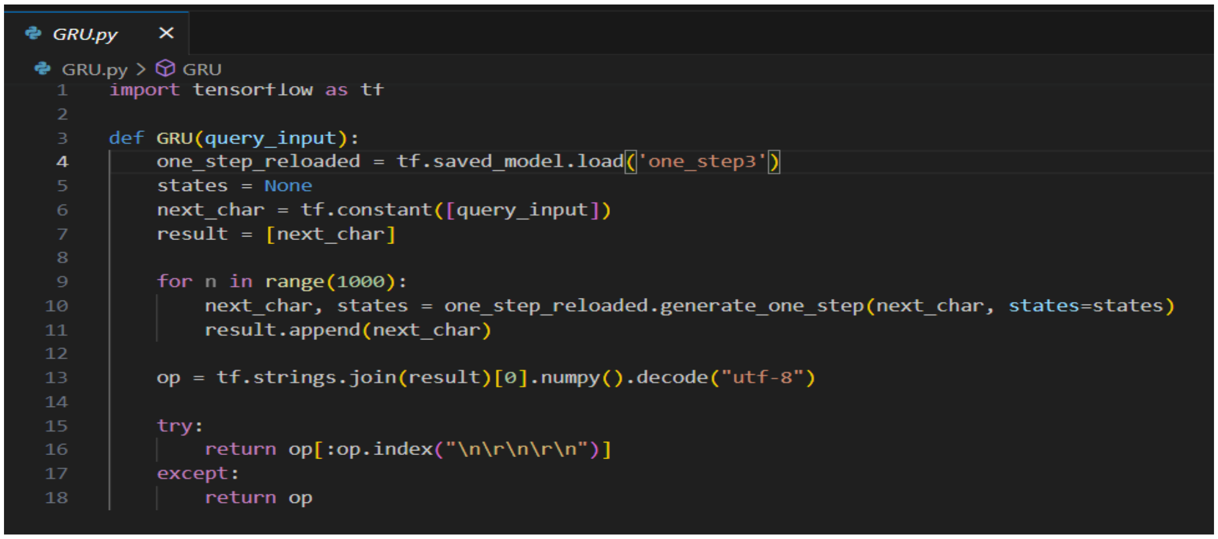
```
class OneStep(tf.keras.Model):
  def __init__(self, model, chars_from_ids, ids_from_chars, temperature=1.0):
    super().__init__()
    self.temperature = temperature
    self.model = model
    self.chars_from_ids = chars_from_ids
    self.ids_from_chars = ids_from_chars

    # Create a mask to prevent "[UNK]" from being generated.
    skip_ids = self.ids_from_chars(['[UNK]'])[:, None]
    sparse_mask = tf.SparseTensor(
        # Put a -inf at each bad index.
        values=[-float('inf')]*len(skip_ids),
        indices=skip_ids,
        # Match the shape to the vocabulary
        dense_shape=[len(ids_from_chars.get_vocabulary())])
    self.prediction_mask = tf.sparse.to_dense(sparse_mask)
```

Figure 5.37: Predict Function



```
@tf.function
def generate_one_step(self, inputs, states=None):
  # Convert strings to token IDs.
  input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
  input_ids = self.ids_from_chars(input_chars).to_tensor()

  # Run the model.
  predicted_logits, states = self.model(inputs=input_ids, states=states,
                                        return_state=True)
  # Only use the last prediction.
  predicted_logits = predicted_logits[:, -1, :]
  predicted_logits = predicted_logits/self.temperature
  # Apply the prediction mask: prevent "[UNK]" from being generated.
  predicted_logits = predicted_logits + self.prediction_mask

  # Sample the output logits to generate token IDs.
  predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
  predicted_ids = tf.squeeze(predicted_ids, axis=-1)

  # Convert from token ids to characters
  predicted_chars = self.chars_from_ids(predicted_ids)

  # Return the characters and model state.
  return predicted_chars, states
```

Figure 5.38: Predict Function

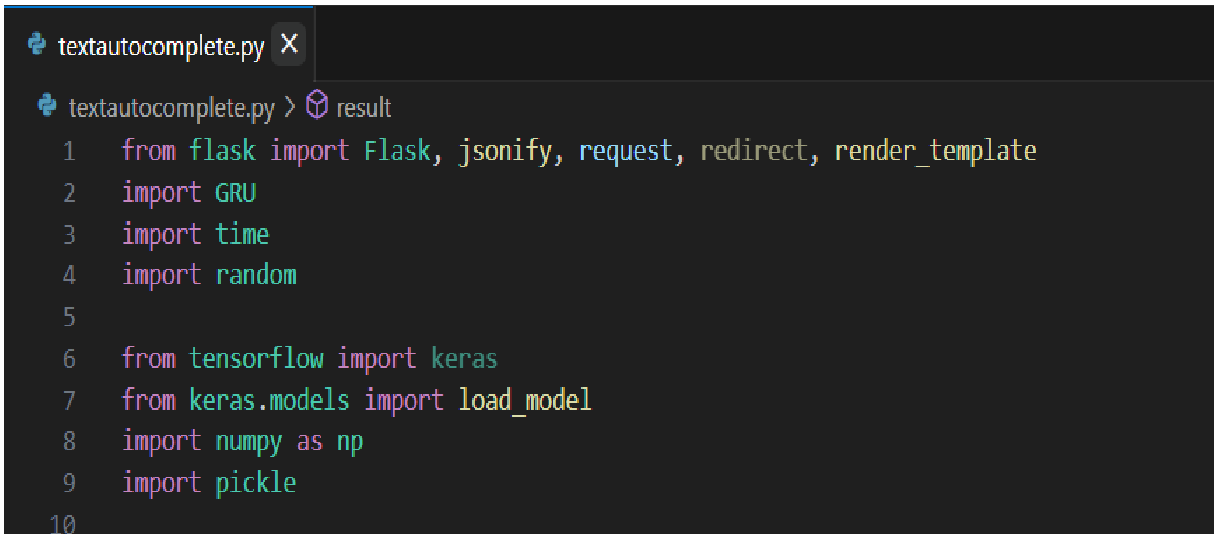


GRU.py ×

GRU.py > GRU

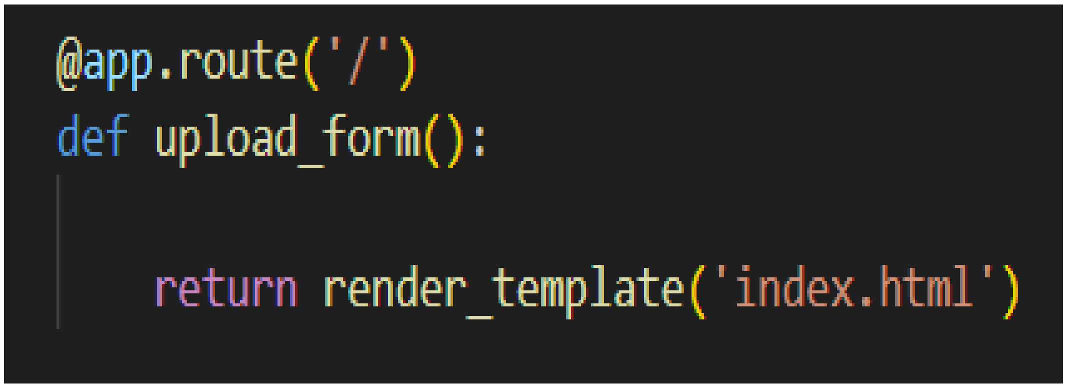
```
import tensorflow as tf

def GRU(query_input):
    one_step_reloaded = tf.saved_model.load('one_step3')
    states = None
    next_char = tf.constant([query_input])
    result = [next_char]

    for n in range(1000):
        next_char, states = one_step_reloaded.generate_one_step(next_char, states=states)
        result.append(next_char)

    op = tf.strings.join(result)[0].numpy().decode("utf-8")

    try:
        return op[:op.index("\n\r\n\r\n")]
    except:
        return op
```

Figure 5.39: Calling Result



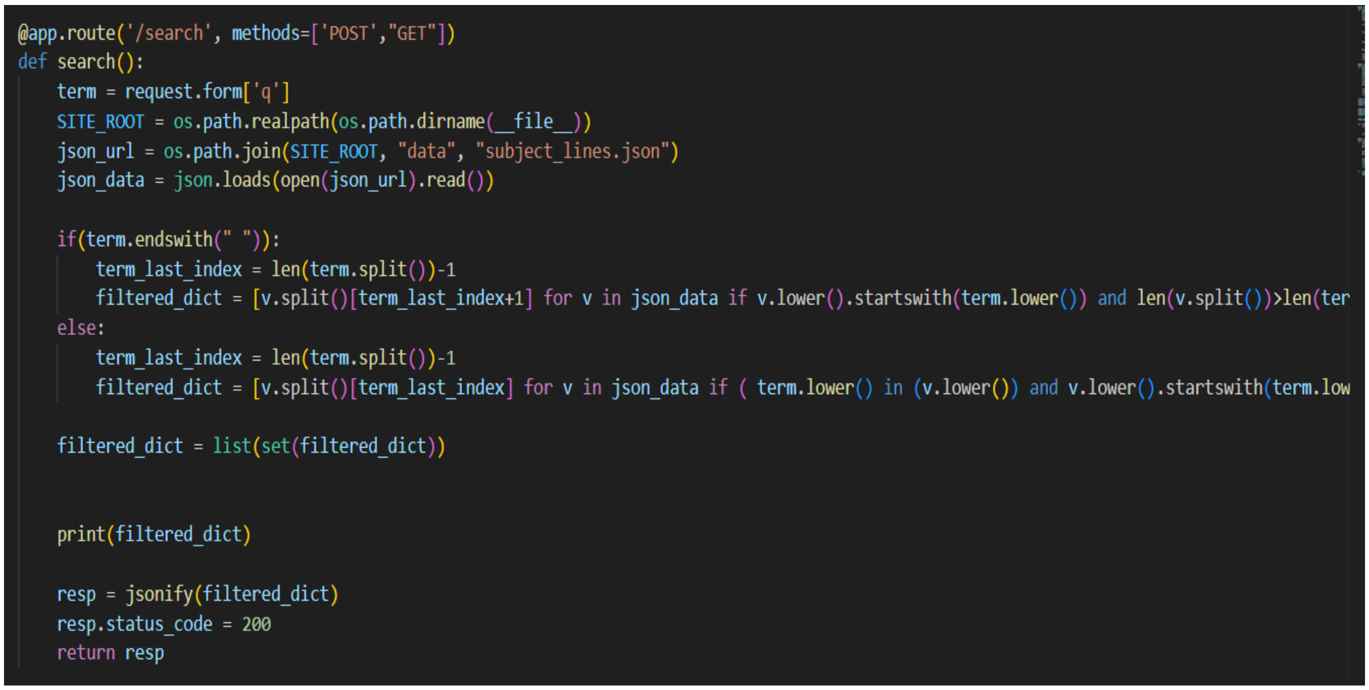
```python
from flask import Flask, jsonify, request, redirect, render_template
import GRU
import time
import random

from tensorflow import keras
from keras.models import load_model
import numpy as np
import pickle
```

Figure 5.40: Necessary imports for Main Function



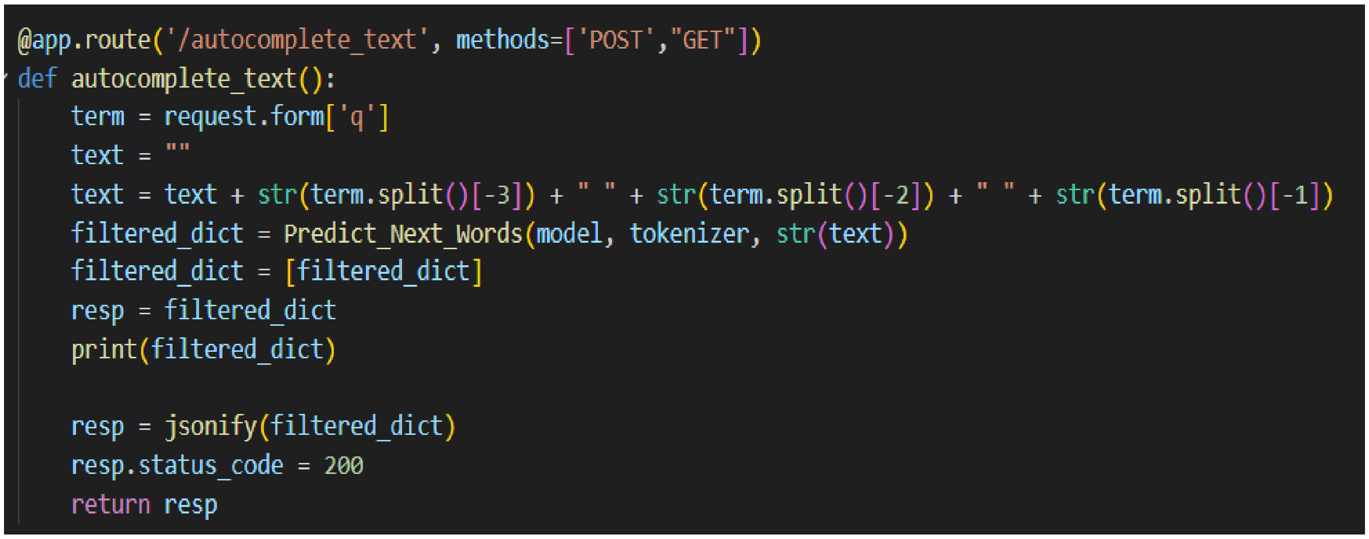
```python
@app.route('/')
def upload_form():

    return render_template('index.html')
```

Figure 5.41: Rendering the html template on visiting the url



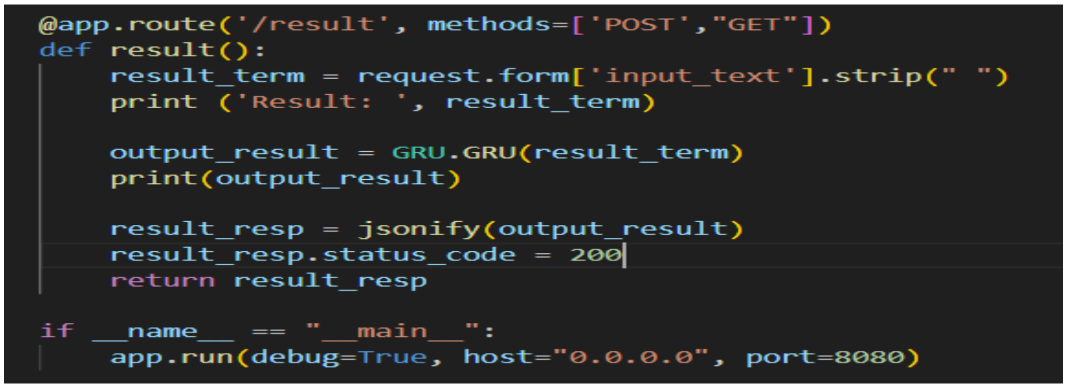
```python
@app.route('/search', methods=['POST',"GET"])
def search():
    term = request.form['q']
    SITE_ROOT = os.path.realpath(os.path.dirname(__file__))
    json_url = os.path.join(SITE_ROOT, "data", "subject_lines.json")
    json_data = json.loads(open(json_url).read())

    if(term.endswith(" ")):
        term_last_index = len(term.split())-1
        filtered_dict = [v.split()[term_last_index+1] for v in json_data if v.lower().startswith(term.lower()) and len(v.split())>len(ter
    else:
        term_last_index = len(term.split())-1
        filtered_dict = [v.split()[term_last_index] for v in json_data if ( term.lower() in (v.lower()) and v.lower().startswith(term.low

    filtered_dict = list(set(filtered_dict))


    print(filtered_dict)

    resp = jsonify(filtered_dict)
    resp.status_code = 200
    return resp
```

Figure 5.42: Auto Suggestion in Search Box



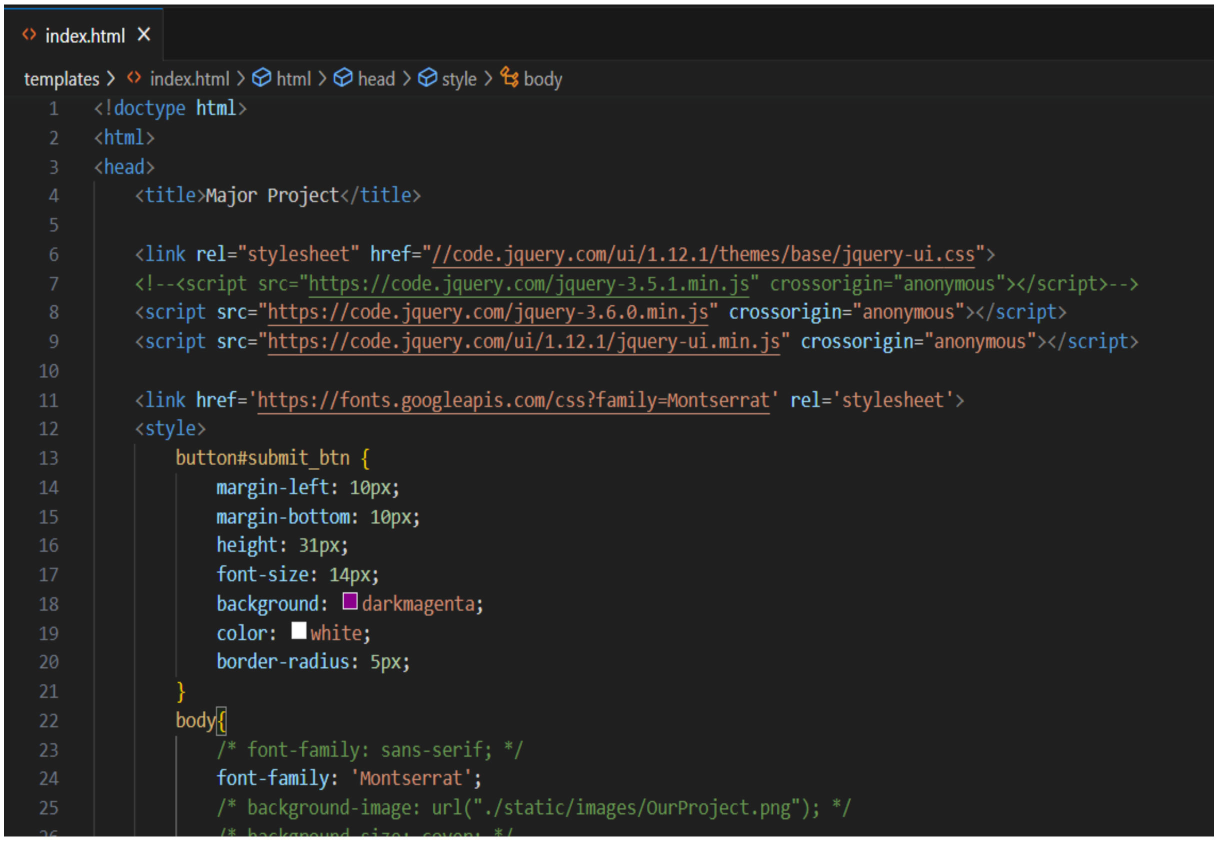
```python
@app.route('/autocomplete_text', methods=['POST',"GET"])
def autocomplete_text():
    term = request.form['q']
    text = ""
    text = text + str(term.split()[-3]) + " " + str(term.split()[-2]) + " " + str(term.split()[-1])
    filtered_dict = Predict_Next_Words(model, tokenizer, str(text))
    filtered_dict = [filtered_dict]
    resp = filtered_dict
    print(filtered_dict)

    resp = jsonify(filtered_dict)
    resp.status_code = 200
    return resp
```

Figure 5.43: Predicting Next Word in the Template



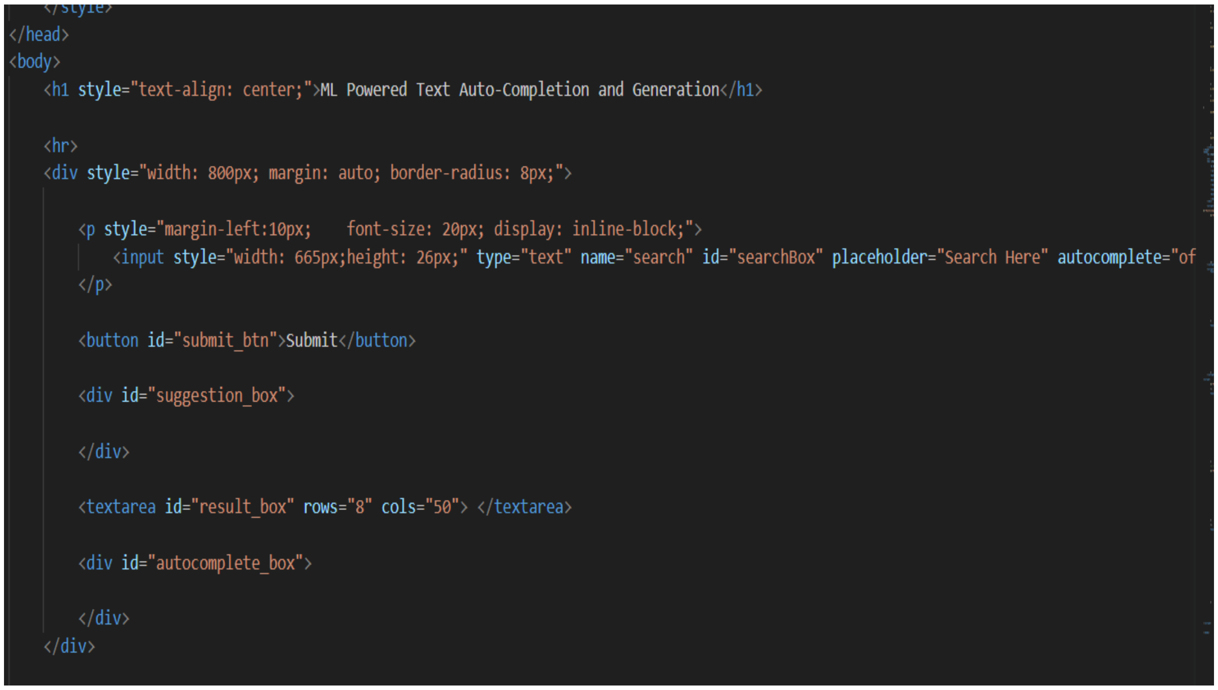
```
@app.route('/result', methods=['POST',"GET"])
def result():
    result_term = request.form['input_text'].strip(" ")
    print ('Result: ', result_term)

    output_result = GRU.GRU(result_term)
    print(output_result)

    result_resp = jsonify(output_result)
    result_resp.status_code = 200
    return result_resp

if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8080)
```

Figure 5.44: Generating Template



```
<!doctype html>
<html>
<head>
    <title>Major Project</title>

    <link rel="stylesheet" href="//code.jquery.com/ui/1.12.1/themes/base/jquery-ui.css">
    <!--<script src="https://code.jquery.com/jquery-3.5.1.min.js" crossorigin="anonymous"></script>-->
    <script src="https://code.jquery.com/jquery-3.6.0.min.js" crossorigin="anonymous"></script>
    <script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js" crossorigin="anonymous"></script>

    <link href='https://fonts.googleapis.com/css?family=Montserrat' rel='stylesheet'>
    <style>
        button#submit_btn {
            margin-left: 10px;
            margin-bottom: 10px;
            height: 31px;
            font-size: 14px;
            background: darkmagenta;
            color: white;
            border-radius: 5px;
        }
        body{
            /* font-family: sans-serif; */
            font-family: 'Montserrat';
            /* background-image: url("./static/images/OurProject.png"); */
```

Figure 5.45: HTML Template



```
    </style>
</head>
<body>
    <h1 style="text-align: center;">ML Powered Text Auto-Completion and Generation</h1>

    <hr>
    <div style="width: 800px; margin: auto; border-radius: 8px;">

        <p style="margin-left:10px;    font-size: 20px; display: inline-block;">
            <input style="width: 665px;height: 26px;" type="text" name="search" id="searchBox" placeholder="Search Here" autocomplete="of
        </p>

        <button id="submit_btn">Submit</button>

        <div id="suggestion_box">

        </div>

        <textarea id="result_box" rows="8" cols="50"> </textarea>

        <div id="autocomplete_box">

        </div>
    </div>
```

Figure 5.46: HTML Template

6. Result and Discussion

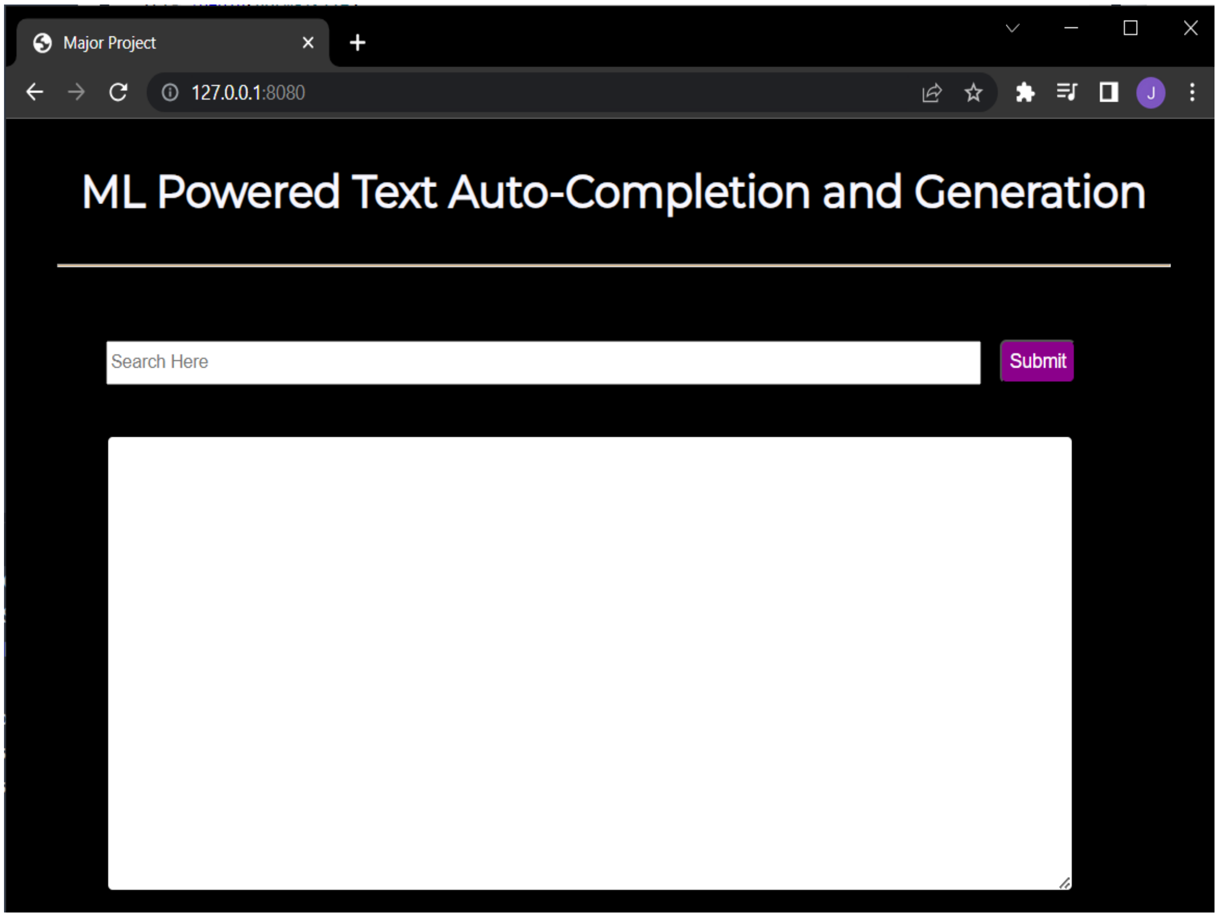


Figure 6.1: User Interface using Flask

The proposed UI is built using the most popular Python web application framework, Flask. The UI has 4 major components, which includes the search box, the auto-complete suggestion box Fig.6.2, result box Fig.6.3, and the Next Word Predictor Fig.6.4. When a user starts typing the input in the search box, the auto-complete model gets triggered and returns the suggestions as shown in Fig.6.2. Once the input is typed and the submit button is hit, the text-generation model renders the template in the result box. This template generated can also be edited. The proposed system also has a feature of next work prediction, where in when the user starts composing the mail in the result box, the next work is predicted and displayed as a suggestion to be used next as shown in Fig.6.4.

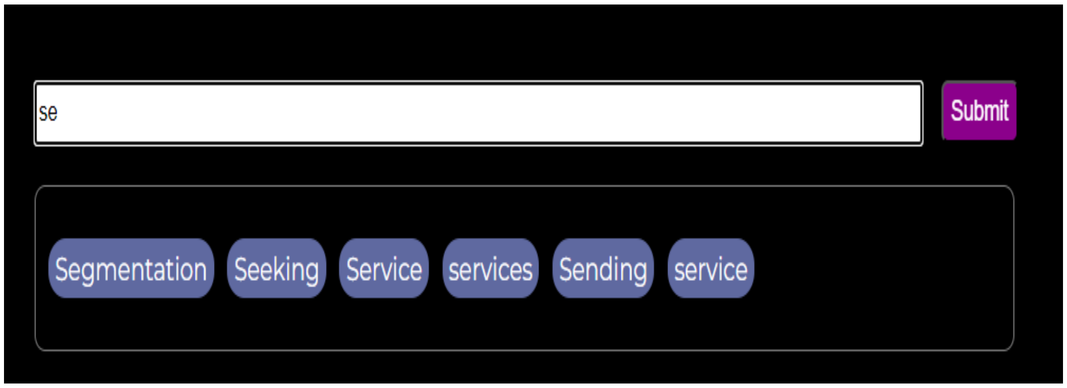
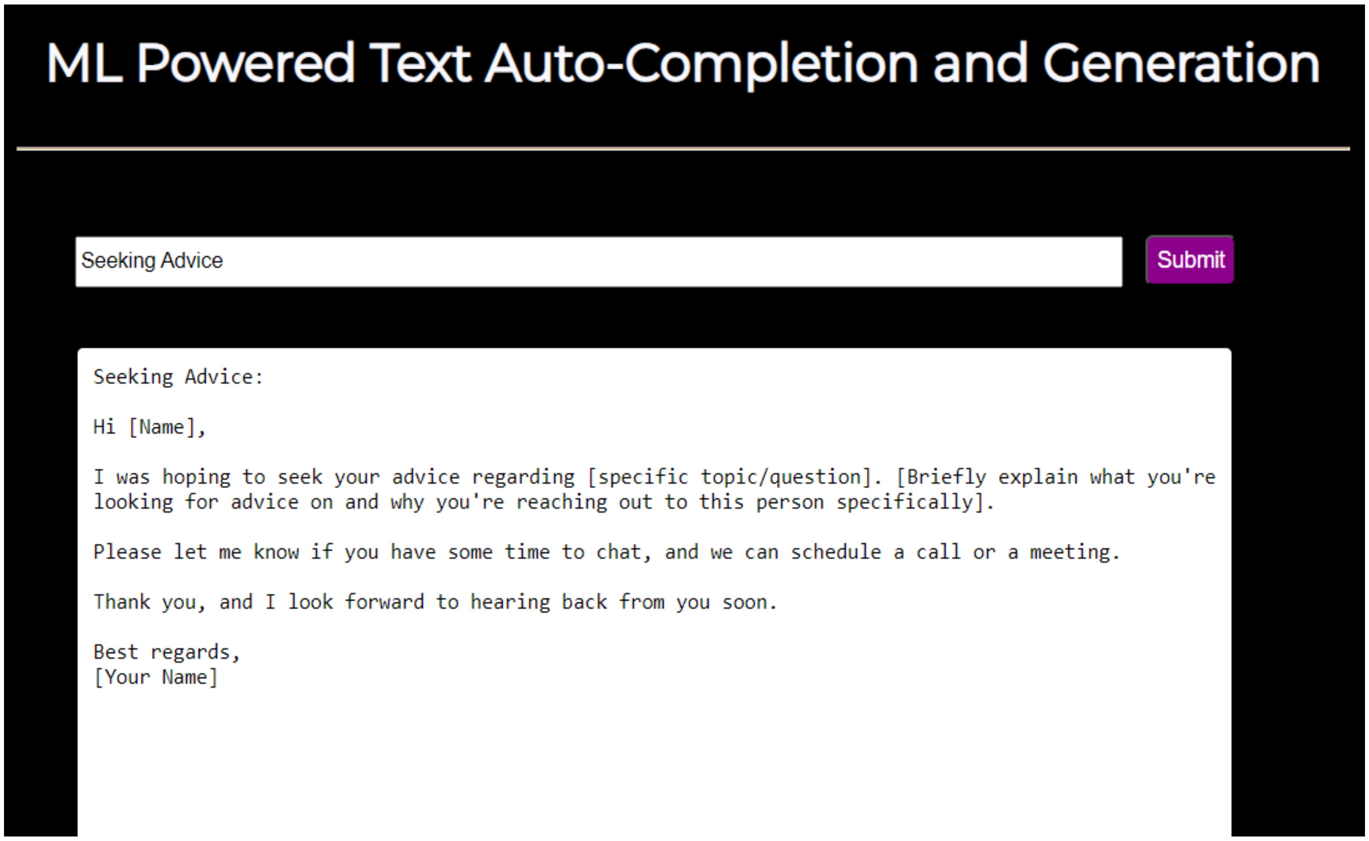
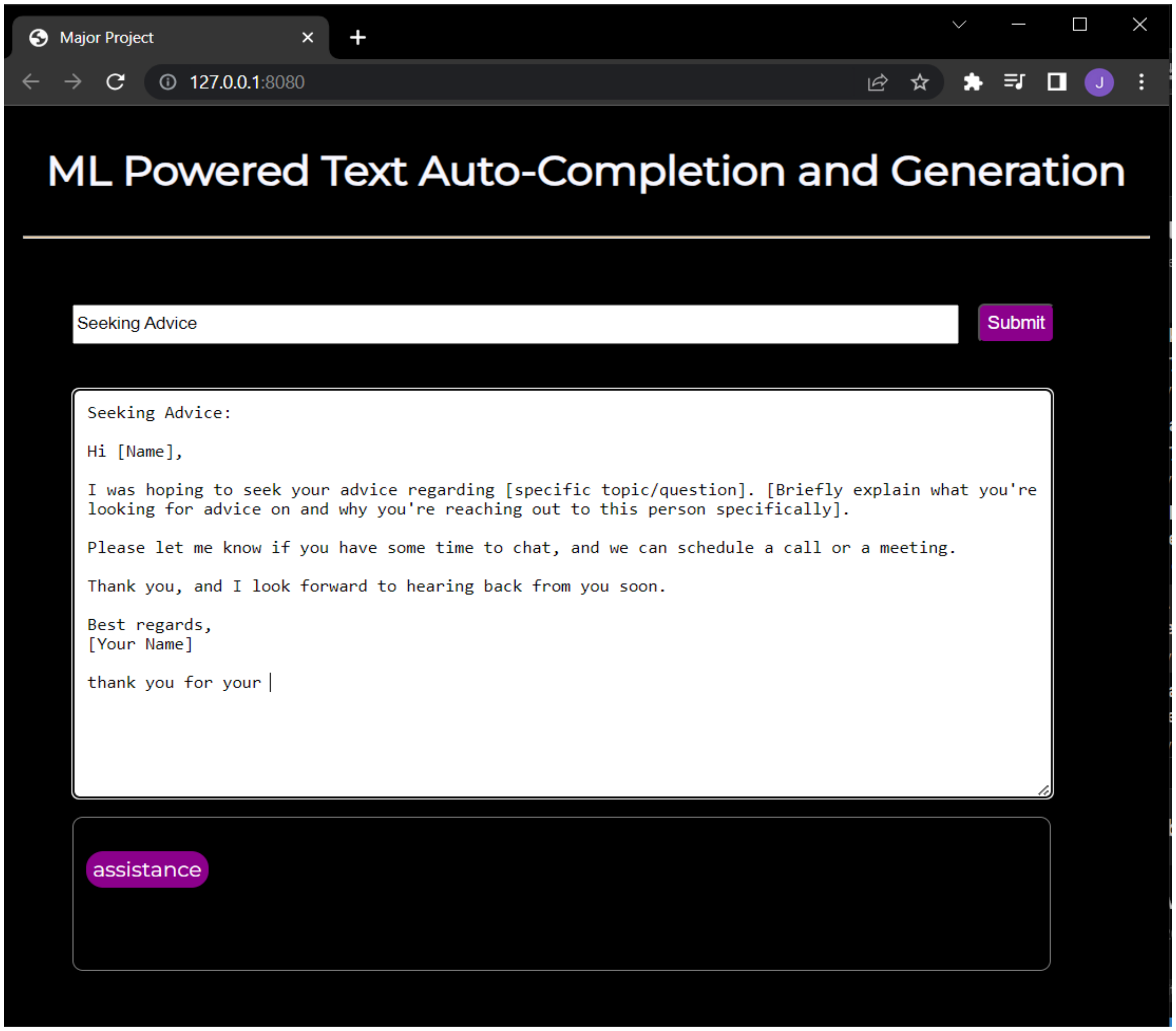


Figure 6.2: Suggestions for Subject Line Input



```
Seeking Advice:

Hi [Name],

I was hoping to seek your advice regarding [specific topic/question]. [Briefly explain what you're
looking for advice on and why you're reaching out to this person specifically].

Please let me know if you have some time to chat, and we can schedule a call or a meeting.

Thank you, and I look forward to hearing back from you soon.

Best regards,
[Your Name]
```

Figure 6.3: Template Generation



```
Seeking Advice:

Hi [Name],

I was hoping to seek your advice regarding [specific topic/question]. [Briefly explain what you're
looking for advice on and why you're reaching out to this person specifically].

Please let me know if you have some time to chat, and we can schedule a call or a meeting.

Thank you, and I look forward to hearing back from you soon.

Best regards,
[Your Name]

thank you for your
```

Figure 6.4: UI depicting all the functionalities

7. Conclusion and Future Scope

The proposed model has the capabilities of Text Generation and Auto-Completion. These features helps in reducing the keystrokes while writing/drafting any text. The model resulted with 6% loss after training. Currently the model works well only for those categories of mails that are in the dataset. These categories include Leave, work-related, marketing, job applications, education, travel and personal templates. Having larger dataset would generate the templates even better. Another feature that could be worked on is the template generation for a mail, i.e. generating a reply template by storing the useful information from previous conversations.

REFERENCES

[1] Mia Xu Chen, Benjamin N Lee, Gagan Bansal, Yuan Cao, Shuyuan Zhang, Justin Lu, Jackie Tsay, Yinan Wang, Andrew M. Dai, Zhifeng Chen, Timothy Sohn, and Yonghui Wu. 2019. “Gmail Smart Compose: Real-Time Assisted Writing”. In The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD ’19), August 4–8, 2019, Anchorage, AK, USA.

[2] Yue Weng, Huaixiu Zheng, Franziska Bell, and Gokhan Tur. 2019. “OCC: A Smart Reply System for Efficient In-App Communications”. In Proceedings of The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Anchorage, AK, USA, August 4–8, 2019 (KDD ’19).

[3] Rajeev Gupta, Ranganath Kondapally, Chakrapani Ravi Kiran. “Impersonation: Modeling Persona in Smart Responses to Email”. IJCAI 2018 conference. Workshop on Humanizing AI.

[4] Shao T., Chen H., Chen W. “Query auto-completion based on word2vec semantic similarity”. J. Phys. Conf. Ser. 1004(1), 12–18 (2018).

[5] Aaron Jaech and Mari Ostendorf. 2018. “Personalized Language Model for Query Auto-Completion”. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics.

[6] Enron Email Dataset - https://www.cs.cmu.edu/ enron/