IMPLEMENTATION OF AN AUTOMATIC QUESTION ANSWERING SYSTEM USING MACHINE LEARNING SAKIF AHMED ABIR

Preprint · September 2019

DOI: 10.13140/RG.2.2.13210.39368

1 author:

Sakif Ahmed Abir
University of Liberal Arts Bangladesh (ULAB)
1 PUBLICATION **0** CITATIONS

IMPLEMENTATION OF AN AUTOMATIC QUESTION ANSWERING SYSTEM USING MACHINE LEARNING

SAKIF AHMED ABIR

IMPLEMENTATION OF AN AUTOMATIC QUESTION ANSWERING SYSTEM USING MACHINE LEARNING

SAKIF AHMED ABIR

A project/report submitted in partial
fulfilment of the requirements for the
degree of Bachelor of Science in
Computer Science and Engineering

UNIVERSITY OF LIBERAL ARTS BANGLADESH
Dhaka, Bangladesh

SEPTEMBER, 2019

DECLARETION

This project report is submitted to the Computer Science & Engineering, University of Liberal Arts Bangladesh in partially fulfillment of the requirement for the degree of Bachelor of Science. So, I hereby, declare that this thesis report is based on the study of this system. Materials of work found by other researchers are mentioned by reference. This project report, neither in whole, nor in part, has been previously submitted for any degree.

Signature **:**

Name **:** SAKIF AHMED ABIR

ID **:** 151014085

Date **:** September 19, 2019

CERTIFICATE OF APPROVAL

The project report entitled "Automatic Question Answering system using Machine Learning" is submitted to the Department of Computer Science & Engineering. University of Liberal Arts Bangladesh, Dhaka in partially fulfillment of the requirement for the degree of Bachelor of Science.

Dated: September 2019.

Khan Raqib Mahmud

Lecturer

Department of Computer Science and Engineering

University of Liberal Arts Bangladesh

Signature and Date

Dr. Mohammad Shahriar Rahman

Associate Professor and Head of Department (Acting)

Department of Computer Science and Engineering

University of Liberal Arts Bangladesh

Signature and Date

DEDICATION

I dedicate my dissertation work to my parents who never stop giving of themselves in countless ways. Special feelings of gratitude to my loving parents, whose words of encouragement and push for tenacity ring in my ears and they never left my side and are very special.

I also dedicate this dissertation to my faculty members who have supported me throughout the process. I will always appreciate all they have done, especially my supervisors, Khan Raqib Mahmud sir and Dr. Abul Kalam Al Azad sir for helping me develop my technological and analogical skills and for the many hours of counseling and encouraging and supporting me for entire research. I also want to add Dr. Mohammad Shahriar Rahman sir, our beloved Head of Department who encouraged me throughout my time during graduation.

ACKNOWLEDGEMENT

I wish to express my deepest appreciation to all those who helped me, in one way or another, to complete this project. First and foremost, I thank Almighty Allah who provided me with strength, direction and purpose throughout the project.

I would like to express my deep and sincere gratitude to my project supervisors, Khan Raqib Mahmud Sir and Dr. Abul Kalam Al Azad sir for giving me the opportunity to work with them in this project, I was able to overcome all the obstacles that I encountered in these enduring three months of my project. In fact, they always gave me immense hope every time I consulted with them over problems relating to my project. I would like to extend my indebtedness and thanks to them for their guidance and valuable advice at every step of the thesis. I will be forever thankful to you. I can never pay you back for all the help you have provided me, the experience you have helped me gain by working for you.

ABSTRACT

"Automatic Question Answering System" has been identified as one of the attracting and most doing research areas in present time. Automatic Question answering is a task which is basically used for flexibility and availability. This system already has drawn the attention of many researchers due to its varying applications such as websites, customer care service, prospect engagement, brand story telling. There are multiple approaches can be taken in consideration to process it so that the system can give support as close to as a human. Among them deep learning-based approaches have been provided with the state-of-the-art performance according to particular Natural Language Processing task. Our approach is to build this model using Machine Learning approach. The experiment shows that our QA system is able to give responses to the user in real time. At first, it takes input then search for the matching answers in the database, then comes up with an answer. We have used out own BENGALI DATASET to train our model, which consist of around 1 thousand test question and answers. We divided the train data and test data into 60% and 40 % respectively. We used this dataset because there is no available Bengali corpus which we can use, but the bigger the dataset is the more accuracy will be gain by the system. The Algorithm is we give input the input statement is processed by logic adapters and return response with answer. Based on the evaluation of users, we can say QAs can produce natural responses but in English. In response section we are facing main issue, it is taking inputs in Bengali but giving responses in English. This can be a work for the future research.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

The absolute goal of the Artificial Intelligence research is to build a machine which can converse with a human such that no one can differentiate it from a real human being. After Alan Turing proposed his Turing Test in 1950 in his famous work "Computing Machinery and Intelligence" [16], it has been almost 60 years that researchers are trying to pass the test. As a part of the research to develop an intelligent conversational agent, in 1964, Eliza [14], a simulation of a Rogerian psychotherapist, was developed in Massachusetts Institute of Technology (MIT). It was capable of replying the sentences used by the users back to them. It was the beginning of the research on the conversational agent.

A conversational agent is a program which can converse in a natural language with the user either based on the knowledge base (retrieval based closed domain model) or by generating new sentences (generative based open domain model) in a chat interface and sometimes, it is able to perform actions based on the conversations (goal-oriented chatbots, for example, pizza ordering chatbot) [4]. It is popularly known as chatbot or chatterbot or bot.

A Question Answering system or simply Chatbot is one of the main concerns of the study of Human-Computer Interaction (HCI) [7].We can cite the examples of Cleverbot or Simsimi, automated tutorials and online assistants as chatbots in use [11]. The use cases of chatbot include customer care representative, sales agent, FAQ answerers etc. With the rise of smartphones and other high computational devices, chatbot becomes one of the hot topics of Natural Language Processing (NLP) research. This is to be noted that all research were based on only one language which is English [10]. Although some works have been done on Chinese and Spanish languages but there are very few in number. Other languages were left out because of the lack of quality data corpus and natural language processing tools.

We evaluate the chatbot based on user's satisfaction. Zhou advised that "Evaluation should be adapted to the application and to user needs [24]. If the Chabot is meant to be adapted to provide a specific service for users, then the best evaluation is based on whether it achieves that service or task" [26]. The main task of our research is to interact with the user in fluent and syntactically correct English.

1.2 PROBLEM IDENTIFICATION

Our proposed thesis is about the automatic question answering system as the main scope of the problem. For recent years the automatic question answering system has been such accepted issue due to increasing number of websites, online shopping sites and telecommunication services. To reduce the human labour and flexible along with quick response privilege different type of approaches have been taken to develop a system for automatic question answering system. To keep pace with the fast growing world automatic question answering systems play a vital role. These systems are tremendous asset for any kind of online and offline system. The fundamental thought behind QA system is to assist man-machine interaction.

1.3 MOTIVATION

Last decade was the decade where a technological revolution took place. With the increase in usage of smartphones, the number of the social network users and mobile app users has increased manifold [29]. Now the question is "What's next?". From recent research in the pattern of consumer behavior related to smartphones, it is being seen that users has limited them to few numbers of the app and spends most of the time there. So the post- app era demands a new trend which can be chatbot. The user usually searches the solutions of their problems in Google, Yahoo, and other search engines but either they do not retrieve concise or relevant information [18], or they retrieve documents or links to these documents instead of an appropriate answer to their problems. To address such problem the idea of chatbot arises in which user asks in natural language and receives a concise and appropriate answer [14].

Chatbots can be new updated version of the mobile app or the search engines which will be able to interact with the user in natural language. It is being seen that the limited number of apps in which the users have confined them to are mostly social and messaging apps. So we can surely say it is a positive indication for the development of chatbots. Age of interacting with computers with predetermined commands or clicking on the graphical user interface is long gone, now it is the demand of time the computer starts to take commands in natural language [22].

With the growth of online-based services like shopping, ordering foods or any official works, it has become necessary to build chatbot to handle large customer base from all over the world at any time. Recently many developments took place in NLP research so with the large availability of tools for building conversation agent it is the time to transition to taking natural language inputs interface. “The need of conversational agents has become acute with the widespread use of personal machines with the wish to communicate and the desire of their makers to provide natural language interfaces.” [15]. There is no Bengali chatbot available currently, one has been made named “Golpo” but we have found only the research paper. The physical software doesn’t exists. So all the above-mentioned reasons are the factors that played an influential role which motivates us to build an automatic question answering system or chatbot.

1.4 THESIS CONTRIBUTION

For better understanding the internal working process and different functions of Machine Learning is to be needed as an active area of research. As our main focus is to create a system for Answering Questions from user automatically by training. So, we will contribute through the thesis that we will develop a model/system that will efficiently can answer and learn from different questions.

1.5 THESIS OUTLINE

In Chapter 1 we introduces the treatise of human-computer interaction, the need of Chabot and motivation behind this work. In Chapter 2 same topics related papers and existing automatic question answering systems are discussed. In Chapter 3, the main architecture/methodology for this thesis is discussed. In Chapter 4, Experiments by the systems were highlighted. The experimental results, which is obtained from the system are presented answer are discussed in the section 5. The report is ended in Chapter 6 with the summary of the system and relative discussion regarding future works.

CHAPTER 2

LITERATURE REVIEW

In this chapter we will know a little brief about QA system and previous works on QA systems or chatbots.

2.1 BRIEF KNOWLEDGE ABOUT QA SYSTEM

Question answering is an important end-user task at the intersection of natural language processing (NLP) and information retrieval (IR). QA systems can bridge the gap between IR-based search engines and sophisticated intelligent assistants that enable a more directed information retrieval process. Such systems aim at finding precisely the piece of information sought by the user instead of documents or snippets containing the answer. A special form of QA, namely extractive QA, deals with the extraction of a direct answer to a question from a given textual context.

The creation of large-scale, extractive QA datasets [13] sparked research interest into the development of end-to-end neural QA systems. A typical neural architecture consists of an embedding-, encoding-, interaction- and [22] answer layer p. Most such systems describe several innovations for the different layers of the architecture with a special focus on developing powerful interaction layer that aims at modeling word-by-word interaction between question and context.

Although a variety of extractive QA systems have been proposed, there is no competitive neural baseline. Most systems were built in what we call a top-down process that proposes a complex architecture and validates design decisions by an ablation study. Most ablation studies, however, remove only a single part of an overall complex architecture and therefore lack comparison to a reasonable neural baseline [27]. This gap raises the question whether the complexity of current systems is justified solely by their empirical results.

Another important observation is the fact that seemingly complex questions might be answerable by simple heuristics. Let’s consider the following example:

***When** did building activity occur on St. Kazimierz Church?*

Building activity occurred in numerous noble palaces and churches [...]. One of the best examples [..] are Krasinski Palace (1677-1683), Wilanow Palace (1677-1696) and St. Kazimierz Church (**1688-1692**)

Although it seems that evidence synthesis of multiple sentences is necessary to fully understand the relation between the answer and the question, answering this question is easily possible by applying a simple context/type matching heuristic. The heuristic aims at selecting answer spans that a) match the expected answer type (a time as indicated by "When") and b) are close to important question words [26]. The actual answer "1688-1692" would easily be extracted by such a heuristic. Early work on chatbots [21] relied on handcrafted templates or heuristic rules to do response generation, which requires huge effort but can only generate limited responses. Recently, researchers begin to develop data driven approaches [18].

Statistical goal-oriented dialogue systems have long been modeled as partially observable Markov decision processes (POMDPs) [19]. And are trained using reinforcement learning based on user feedback. [11], recently applied deep reinforcement learning successfully to train non-goal oriented chatbot type dialogue agents. They show that reinforcement learning allows the agent to model long-term rewards and generate more diverse and coherent responses as compared to supervised learning.

Retrieval based methods select a proper response by matching message response pairs [3]. Retrieval-based methods [5] retrieve response candidates from a pre-built index, rank the candidates, and select a reply from the top ranked ones. In related work, we found response selection for retrieval-based chatbots in a single turn scenario, because retrieval-based methods can always return fluent responses [8] and single turn is the basis of conversation in a chatbot.

2.2 CHATBOT

A conversational agent which can converse with a human, based on the provided knowledge base and the natural language it was trained on, in any platform e.g. mobile, website or desktop application etc. is called a chatbot. After Eliza was created, chatbot for long was one of the most sought topics of academic interest among AI researchers. But it was not until 2016, it gained the interest of general mass. With the launch of smartphone based chatbots such as the Apple Siri [3], Amazon Echo, and China's WeChat [4], chatbots turn into one of the hottest trends in technology. Apart from this, some technological giant companies like Facebook Messenger and Skype declared to give full support to the developers for the development of chatbot. Google, the biggest corporation in technology, entered the competition by launching a chatbot application (Allo) powered by its artificial intelligence (AI) and big data.

Human-computer interaction is one of the most difficult challenges in Natural Language Processing (NLP) research. It is a combination of different fields which facilitate communication between users and computers using a natural language depending solely on the language and the available natural language processing techniques [14]. The whole world is entering an era of conversational agents. The era of talking machines is not very far away. In words of Alan Turing, we can say “I propose to consider the question, 'Can machines think?” [16].

Much work has been done in information retrieval (IR), machine translation, POS tagging, annotation, and auto-summarization. Although there is quite a large literature on the development of an intelligent machine but still researchers are not successful in making an intelligent machine which can pass Turing Test. Because an intelligent conversational agent is the combination of all the fields of Natural Language Processing (NLP). With the advent of smart personal assistants like Siri, Google Chrome, and Cortana [25], we may hope for the fulfillment of the dream of Colby. “Before there were computers, we could distinguish persons from non-persons on the basis of an ability to participate in conversations. But now, we have hybrids operating between a person and non-persons with whom we can talk in ordinary language.” [3].

For achieving this goal AI researchers are working relentlessly to make chatbot which can talk like a human. The purpose of a chatbot system is to simulate a human conversation; the chatbot architecture integrates a language model and computational algorithms to emulate informal chat communication between a human user and a computer using natural language. Naturally, chatbot can extend daily life, such as help desk tools, automatic telephone answering systems, to aid in education, business, and e-commerce [6].

Although researchers get success in building chatbot using the retrieval based method but they do not have much success in the generative based method. As [18] has pointed out the cause of it is not having an appropriated database and the probability of a slightly different answer can lead to a different conversation [23]. The main drawback of the generative method is grammatically incorrect and inconsistent sentences.

The present time is the transition period of transforming technology taking commands from predetermined commands to taking inputs from natural language. It is being predicted that chatbot is the future of search engines because it is one of the easiest ways to fetch information from a system. The most important advantage of chatbot based search engine is users can easily search by writing in natural language instead of looking up in a search engine or browse several web pages to collect information. The chatbot conversation framework falls into two categories: retrieval based and generative based chatbot [31].

Often considered as an easier approach, the retrieval-based model uses a knowledge base of predefined responses and employs a pattern matching algorithm with a heuristic to select an appropriate response [21]. The retrieval based systems do not generate any new text. They can reply on within the domain of their knowledge base. Generative models do not have any knowledge base. So they generate new text in every response. This model relies on the machine translation techniques.

If we compare both the model, we will find advantages and disadvantages in both of them. Since the knowledge base of the retrieval based model is handcrafted by the developer it is not prone to syntactical mistakes. But its disadvantage is it cannot give respond to anything beyond the scope

of its knowledge base. On the other hand, generative models are very difficult to train and prone to grammatical mistakes.

The chatbot framework can be again divided into two types based on its domain: closed domain and open domain

1. The closed domain chatbots are those which can reply to a limited number of subjects. A very good example would be goal based chatbot. An open domain chatbot does not have any knowledge base so it has to generate new sentence for each interaction. Since it has no goal so the users can take the conversation to anywhere. Often unrelated, inconsistent and grammatically incorrect sentences are produced in an open domain modeled chatbot.
2. So it is very difficult to build a good open domain chatbot which overcomes all the defaults whereas close domain can be easily built if the corpus is available. As we have discussed the framework let us briefly discuss the internal mechanism of chatbot. There are three important types of artificial intelligence services which are needed to build a chatbot. [27]

1. **Rule-based pattern recognition**:

Mainly any retrieval based chatbot relies on this rule-based pattern recognition. In this model, the rules are the regular expressions. The advantage of a regular expression is that they are flexible and in the case of need new expressions can be created.

2. **Natural language classifier:**

It is used to detect and classify intent of a user command.

3. **Rule-based conversation manager:**

This service can apply rules and generate scripted responses based on the user's intent and data that is associated with the entities, such as location and time.

Therefore, we discuss the definition, the state of art, the classification of the chatbot framework, internal mechanism and classification of artificial intelligence services to build a chatbot to introduce the background of our work.

2.3 RELATED WORKS

In this section we'll discuss about some previous work related to Question answering systems.

2.3.1 Statistical QA

Traditional approaches to question answering typically involve rule-based algorithms or linear classifiers over hand-engineered feature sets. Richardson et al. (2013) proposed two baselines, one that uses simple lexical features such as a sliding window to match bags of words, and another that uses word-distances between words in the question and in the document. Berant proposed an alternative approach in which one first learns a structured representation of the entities and relations in the document in the form of a knowledge base, then converts the question to a structured query with which to match the content of the knowledge base [2]. Wang described a statistical model using frame semantic features as well as syntactic features such as part of speech tags and dependency parses [21]. Chen proposed a competitive statistical baseline using a variety of carefully crafted lexical, syntactic, and word order features [3].

2.3.2 Neural QA

Neural attention models have been widely applied for machine comprehension or question-answering in NLP. Hermann proposed an Attentive Reader model with the release of the CNN/Daily Mail cloze-style question answering dataset. Hill released another dataset stemming from the children's book and proposed a window-based memory network. Kadlec presented a pointer-style attention mechanism but performs only one attention step. Sordoni introduced an iterative neural attention model and applied it to cloze-style machine comprehension tasks [19].

2.3.3 E-Commerce Question-Answering

In recent years, product aware question answering has received several attention. Most of existing strategies aim at extracting relevant sentences from input text to answer the given question propose a framework for opinion QA, which first organizes reviews into a hierarchy structure and retrieves review sentence as the answer propose an answer prediction model by incorporating an aspect analytic model to learn latent aspect-specific review representation for predicting the answer. External knowledge has been considered with the development of knowledge graphs. McCauley propose a method using reviews as knowledge to predict the answer, where they classify answers into two types, binary answers (i.e. “yes” or “no”) and open-ended answers. Incorporating review information, recent studies employ ranking strategies to optimize an answer from candidate answers. Meanwhile, product aware question retrieval and ranking has also been studied. Cui propose a system which combines questions with RDF triples. Yu propose a model which retrieves the most similar queries from candidate QA pairs, and uses corresponding answer as the final result. However, all above task settings differ from our task. Unlike above approaches, our method is aimed to generate an answer from scratch, based on both reviews and product attributes.

2.3.4 Reading Comprehension

Given a question and relevant passages, reading comprehension extracts a text span from passages as an answer. Recently, based on a widely applied dataset, i.e., SQuAD, many approaches have been proposed. Seo use bi-directional attention flow mechanism to obtain a query aware passage representation. Wang [32] propose a model to match the question with passage using gated attention-based recurrent networks to obtain the question-aware passage representation. Consisting exclusively of convolution and self-attention, QANet achieves the state-of-the-art performance in reading comprehension. As mentioned above, most of the effective methods contain question-aware passage representation for generating a better answer. This mechanism make the models focus on the important part of passage according to the question.

2.3.5 Sequence-to-sequence architecture

In recent years, sequence to-sequence (seq2seq) based neural networks have been proved effective in generating a fluent sentence. The seq2seq model is originally proposed for machine translation and later adapted to various natural language generation tasks, such as text summarization and dialogue generation. Rush apply the seq2seq Mechanism with attention model to text summarization field. Then See et al. add copy mechanism and coverage loss to generate summarization without out-of-vocabulary and redundancy words. The seq2seq architecture has also been broadly used in dialogue system. Tao propose a multi-head attention mechanism to capture multiple semantic aspects of the query and generate a more informative response. Different from seq2seq models, our model utilizes not only the information in input sequence but also many external knowledge from user reviews and product attributes to generate the answer that matches the facts. Unlike traditional seq2seq model, there are several tasks which input data is in key value structure instead of a sequence. In order to utilize these data when generating text, key-value memory network (KVMN) is purposed to store this type of data. He et al. incorporate copying and retrieving knowledge from knowledge base stored in KVMN to generate natural answers within an encoder-decoder framework. This uses a KVMN to store the translate history which gives model the opportunity to take advantage of document-level information instead of translate sentences in an isolation way [27].

2.3.6 Chat-Bot for College Management System Using A.I

Question Answering (QA) systems can be identified as information accessing systems which try to answer to natural language queries by providing answers instead of providing the simple list of document links. QA system selects the most appropriate answers by using linguistic features available in natural language techniques. They differ mainly from the knowledge sources, the broadness of Dialog Systems (NLDS) is an appropriate and easy way to access information. QA system based on Semantic enhancement as well as the implementation of a domain-oriented based on a pattern-matching chat-bots technology developed within an industrial project (FRASI). The proposed approach simplifies the chat-bots realization which uses two solutions. First one is the ontology, which is exploited in a twofold manner: to construct answers very actively

as a result of an deduction process about the domain, and to automatically populate, off-line, the chat-bots KB with sentences that can be derived from the ontology, describing properties and relations between concepts involved in the dialogue. Second is to preprocess of sentences given by the user so that it can be reduced to a simpler structure that can be directed to existing queries of the chat-bots. The aim is to provide useful information regarding products of interest supporting consumers to get what they want exactly. The choice was to implement a QA system using a pattern-matching chat-bots technology [23].

2.3.7 Product-Aware Answer Generator (Paag)

They have proposed the task of product-aware answer generation, which aims to generate an answer for a product-aware question from product reviews and attributes. To address this task, they have proposed product-aware answer generator (PAAG): An attention-based question aware review reader is used to extract semantic units from reviews, and key-value memory network based attribute encoder is employed to fuse relevant attributes. In order to encourage the model to produce answers that match facts, they have employed an adversarial learning mechanism to give additional training signals for the answer generation. To tackle the shortcomings of vanilla GAN, they have applied the Wasserstein distance as value function in the training of consistency discriminator. In their experiments, they have demonstrated the effectiveness of PAAG and have found significant improvements over state-of-the-art baselines in terms of metric-based evaluations and human evaluations. Moreover, we have verified the effectiveness of each module in PAAG for improving product-aware answer generation [21].

CHAPTER 3

BACKGROUD

In this chapter we will talk about Natural language processing, Machine learning, Deep Learning, Neural Network and Deep Neural Networks. We will also understand how they works, what are the advantages and limitations. Also we will see the general work flow diagram.

3.1 NATURAL LANGUAGE PROCESSING - NLP

Natural Language Processing (NLP) is a research area of Artificial Intelligence (AI) which focus in the study and development of systems that allows the communication between a person and a machine through natural language [22]. Chatbots belong to the area of NLP given the importance of their ability to understand natural language and know how to extract relevant information from it. Both models, retrieval-based and generative-based must be able to identify some information from the input sentence in order to pick or create an answer.

3.2 MACHINE LEARNING - ML

Machine Learning is a field of study of AI that studies and develop techniques capable to learn tasks as classification or regression from a data set. There are different algorithms without being any of them, in general, better among the others (No Free Lunch Theorem) [2]. The suitability of one algorithm in particular, depends exclusively on the nature and type of the problem addressed.

The aim of a learning algorithm is to estimate the behavior of a training set by the identification of their inherent pattern. Once accomplished, it must be capable to perform tasks as classification or regression given unseen samples. All the learning algorithms require a learning phase at which, an objective function is defined as a metric to optimize in order to get a reference of how well our model fits to the problem (e.g. Minimization of the error function). Then, the algorithm iterates through the training set looking for the optimization of the metric. It is important to have three disjoint sets of samples in machine learning algorithms: training, validation and test set [9]. The training set is used as examples for the objective function optimization. A validation

set is required when it is necessary to compute the optimal parameters of an algorithm. Finally, the test set is used to test how well the algorithm has learned and generalized the problem.

3.3 NAIVE BAYESIAN CLASSIFIER

In machine learning, naïve Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Naive Bayes has been studied extensively since the 1960s. It was introduced (though not under that name) into the text retrieval community in the early 1960s, and remains a popular (baseline) method for text categorization, the problem of judging documents as belonging to one category or the other (such as spam or legitimate, sports or politics, etc.) with word frequencies as the features. With appropriate pre-processing, it is competitive in this domain with more advanced methods including support vector machines. It also finds application in automatic medical diagnosis [3].

Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers. In the statistics and computer science literature, naive Bayes models are known under a variety of names, including simple Bayes and independence Bayes. All these names reference the use of Bayes' theorem in the classifier's decision rule, but naïve Bayes is not (necessarily) a Bayesian method.

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. There is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable. For example, a fruit may be considered to be an apple if it is red, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the

probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features.

For some types of probability models, naive Bayes classifiers can be trained very efficiently in a supervised learning setting. In many practical applications, parameter estimation for naive Bayes models uses the method of maximum likelihood; in other words, one can work with the naive Bayes model without accepting Bayesian probability or using any Bayesian methods. Despite their naive design and apparently oversimplified assumptions, naive Bayes classifiers have worked quite well in many complex real-world situations. In 2004, an analysis of the Bayesian classification problem showed that there are sound theoretical reasons for the apparently implausible efficacy of naive Bayes classifiers. Still, a comprehensive comparison with other classification algorithms in 2006 showed that Bayes classification is outperformed by other approaches, such as boosted trees or random forests. An advantage of naive Bayes is that it only requires a small number of training data to estimate the parameters necessary for classification.

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where A and B are events and P(B) = 0.

I. Basically, we are trying to find probability of event A, given the event B is true. Event B is also termed as evidence.

II. P(A) is the priori of A (the prior probability, i.e. Probability of event before evidence is seen). The evidence is an attribute value of an unknown instance (here, it is event B).

III. P(A|B) is a posteriori probability of B, i.e. probability of event after evidence is seen.

Note that there is very little explicit training in Naive Bayes compared to other common classification methods. The only work that must be done before prediction is finding the parameters for the features' individual probability distributions, which can typically be done quickly and deterministically. This means that Naive Bayes classifiers can perform well even with high-dimensional data points and/or a large number of data points.

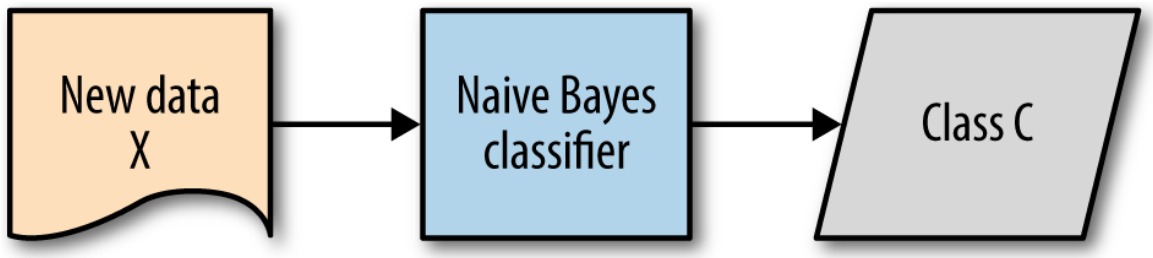


Figure 3.1: Workflow of Naive Bayes Classifier

3.4 SEARCH ALGORITHM

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

I. A search problem consists of:

a. A State Space. Set of all possible states where you can be.

b. A Start State. The state from where the search begins.

c. A Goal Test. A function that looks at the current state returns whether or not it is the goal state.

II. The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.

III. This plan is achieved through search algorithms.

There are so many search algorithms available. Here's is some major search algorithm that are used regularly, divided into two sections:

I. Uninformed Search: The search algorithms in this section have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called Blind search. The following uninformed search algorithms are:

    a. Depth First Search

    b. Breadth First Search

    c. Uniform Cost Search

II. Informed Search: Here, the algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a heuristic. The following informed search algorithms are:

    a. Greedy Search

    b. A* Search

    c. Graph Search

CHAPTER 4

METHODOLOGY

In this chapter the methodology of our system has been described. First we discussed about the model description after that a detail discussion of chatterbot and then in 4.4 we discussed the Algorithm and flowchart then training methods. And this chapter finishes with the implementation of our QA system.

4.1 MODEL DESCRIPTION

We propose a simple decoder and encoder based conversational model agent that will provide chatbot users with an entity from a Knowledge base (KB) by interactively asking for its attributes

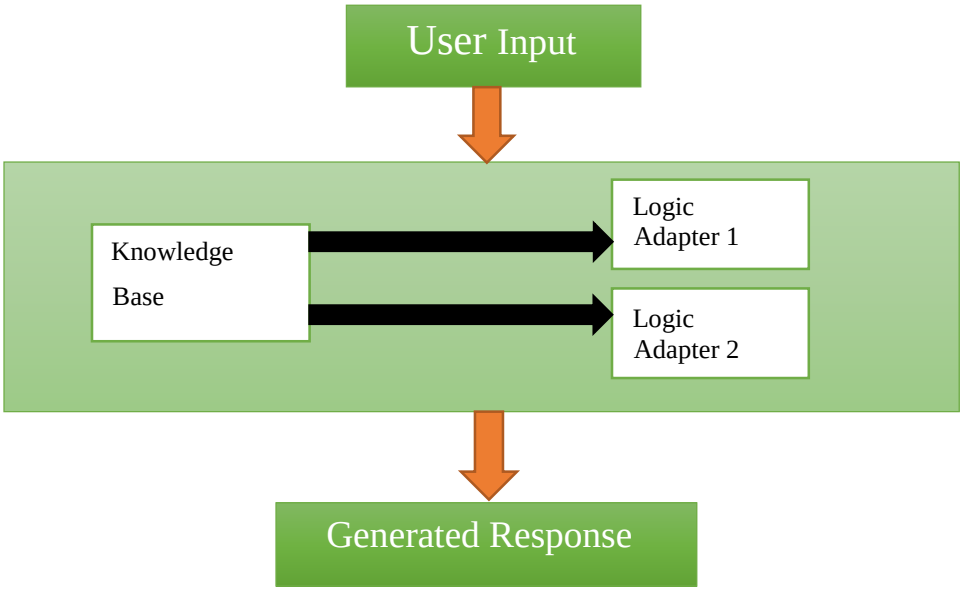


Figure 4.1: Block diagram of QA System

Most of the works related to conversational agents are done on a retrieval based model. The key to the success of response selection lies in accurately matching input messages with proper responses. Our approach for response generation is retrieval based. Retrieval-based model is a model for chatbots which retrieve responses from its knowledge base. It generates a response based on the heuristics, the user's input and the context.

Suppose,

The input to a retrieval-based model is a text t ,

A potential response is r,

Then,

The output of the model is a confidence score C for the response.

C is a function of ConfidenceValue(t, r).

The r with the highest score C is the response which will be sent to the output adapter.

To find a good response you would calculate the score for multiple responses and choose the one with the highest score. "Selecting a potential response from a set of candidates is an important and challenging task for open-domain human-computer conversation, especially for the retrieval-based human-computer conversation" [23]. Since there is a lot of difficulties in building an open domain chatbot so we want to build a domain specific chatbot. So we are providing it with an initial knowledge base and it can always improve its performance measure by learning from the responses of the users.

The workflow of the chatbot is simple and effective.

i. We will get input from the conversational platform or chat platform

ii. We will process the received input. The input statement will be processed by an algorithm which will find the best likelihood valued response for the query. The algorithm will select all the known statements that most closely matches the input statement. It will return the known responses to the selected match and a confidence score value based on matching after computation of each of the response. Here the

confidence score is the likelihood value of the response. The algorithm will return the response that generated the highest likelihood value for itself.

iii. Finally, the response to the input will be returned to the user. For successful completion of user goals, it is also necessary to equip the dialogue policy with real-world knowledge from a database. For constructing this end-to-end system, the following goals can be achieved this by constructing a symbolic query from the current belief states of the agent and retrieving results from the database which match the query.

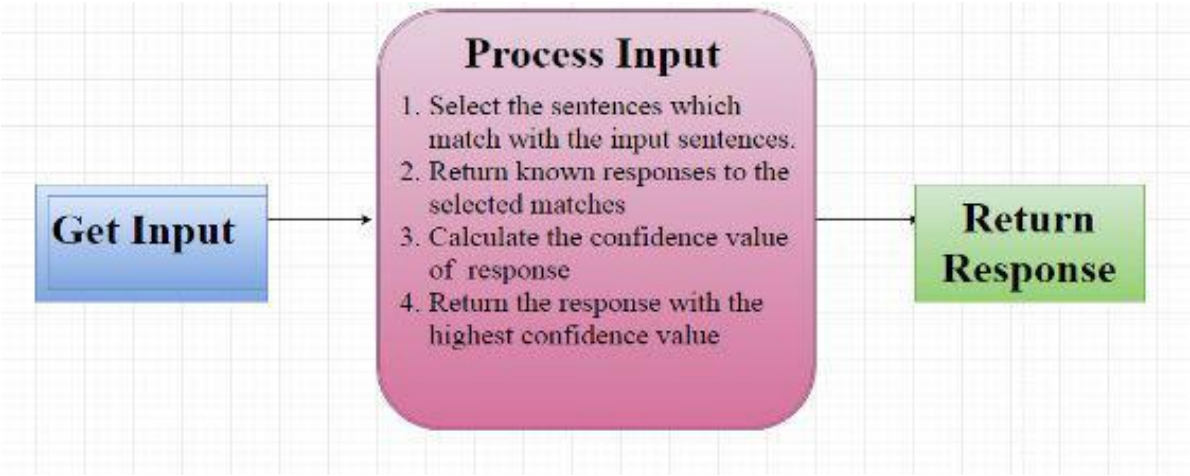


Figure 4.2: Flowchart of the Proposed System

Based on machine learning, ChatterBot is a conversational dialog engine powered by Python which is capable of giving responses based on a knowledge base. We choose this engine for our system because it is language independent. Since Chatterbot has no language dependency in its design, so it is allowed to be trained to speak any language. It is a Python library that makes it easy to generate automated responses to a user's input for the creation of chatbot in any language. To produce different types of responses, ChatterBot applies a selection of machine learning algorithms. This very feature makes it easy for developers to create chatbots and automate conversations with users. The main class of the chatbot is a connecting point between each of ChatterBot's adapters. In this class, an input statement is returned from the input adapter, processed

and stored by the logic and storage adapters, and then passed to the output adapter to be returned to the user [15].

Additionally, the machine-learning nature of ChatterBot allows an agent instance to improve its own knowledge of possible responses as it interacts with humans and other sources of informative data. An untrained instance of ChatterBot starts off with no knowledge of how to communicate. Each time a user enters a statement, the library saves the text that they entered and the text that the statement was in response to. As ChatterBot receives more input the number of responses that it can reply and the accuracy of each response in relation to the input statement increase. The program selects the closest matching response by searching for the closest matching known statement that matches the input, the chatbot then chooses a response from the selection of known responses to that statement [13].

4.2 ALGORITHM AND FLOWCHART

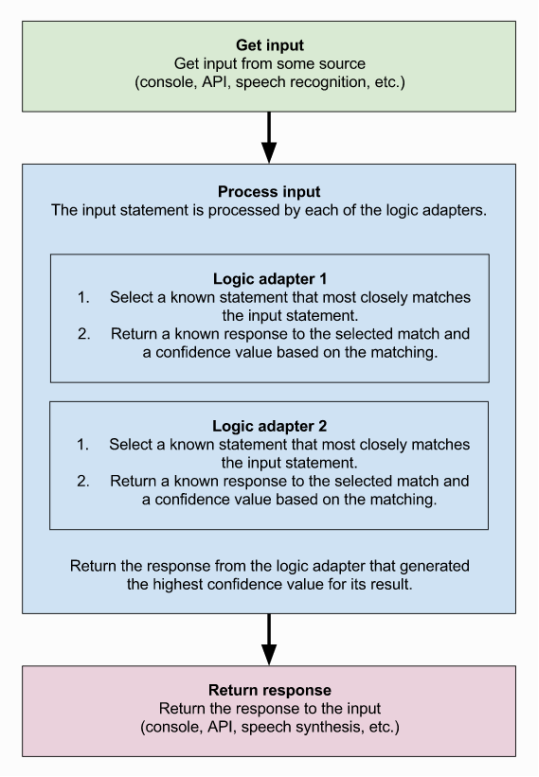


Figure 4.3: Process Flow Diagram of ChatterBot

Since our system is a retrieval based closed domain chatbot its success lies on the pattern matching algorithm. The algorithm of our system is as follows:

1. Our system takes input from the console or any API, after taking input it sends it to the processing unit.
2. In processing part of the system, there are two logic adapters. The text given by the users Is matched with the existing queries in the database and the sentence which matches with the input is selected. For a selected query, there can be multiple responses in the knowledge base so a confidence score for each of the response for the selected sentence is being calculated. In the second logic adapter, a similar procedure is being carried out. So we get two best responses with highest confidence score from the two adapters. Among the two, the one which got the highest confidence score is sent to the output adapter.
3. The step 1, 2 and 3 continues in a loop until the user exits the console.
4. When the user gives an input, it is stored in the knowledge base as a new query. So with each interaction with the user, the knowledge base learns a new query or response.

4.3 TRAINING

ChatterBot includes tools that help simplify the process of training a chat bot instance. ChatterBot's training process involves loading example dialog into the chat bot's database. This either creates or builds upon the graph data structure that represents the sets of known statements and responses. When a chat bot trainer is provided with a data set, it creates the necessary entries in the chat bot's knowledge graph so that the statement inputs and responses are correctly represented [7].

Several training classes come built-in with ChatterBot. These utilities range from allowing you to update the chat bot's database knowledge graph based on a list of statements representing a conversation, to tools that allow you to train your bot based on a corpus of preloaded training data. The training of our system can be done in two processes:

i. Training via list Data:

This training process allows a chatbot to be trained using a list of strings where the list represents a conversation. In this case, the order of each response is based on its placement in a given conversation or the list of string.

1. Import Chatterbot and Trainer Class Library
2. Set the trainer as List Data Trainer
3. Provide the list of strings which you want to provide for training
4. Train the Chatbot
5. Give input and get response

Figure 4.4: Pseudocode of an Instance showing how to use the List Trainer Class

1. List Trainer Class is a class with method called Train. This class allows a chatbot to be trained using a list of strings where the list represents a conversation.
2. In Train method, initialize a data structure to store the history of conversation.
3. Store each statement from the provided list to the initialized data structure

Figure 4.5: Pseudocode for the List Trainer Class

ii. Training via Corpus Data:

ChatterBot comes with a corpus data and utility module that makes it easy to quickly train your bot to communicate. To do so, simply specify the corpus data modules you want to use. This training class allows the chatbot to be trained using data from the

dialog corpus. For our implementation, we used the Corpus Trainer Class of Chatterbot. At first, we have to create QA corpus in the data folder of the Chatterbot in the predefined format in JSON. So from the library, we set the trainer as training with the QA Corpus.We are providing the pseudo code of corpus trainer class based on the code by Gunther Cox for a better understanding of the trainer class.

4.3.1 STORAGE ADAPTERS

1. Corpus Trainer Class is a class with a method called train. This class allows the chatbot to be trained using data from the Chatterbot dialog corpus.
2. Import the corpus of the language mentioned in the command for the chatterbot-corpus library
3. In Train method, initialize a data structure to store the history of conversation.
4. Check whether length of the corpus is larger than the capacity of the storage. If it is larger then return out of space error otherwise start training.
5. Store each statement from the provided list to the initialized data structure.

Figure 4.6: Pseudocode for corpus trainer class of Chatterbot

ChatterBot comes with built-in adapter classes that allow it to connect to different types of databases. For our implementation, we will be using the Json File Storage Adapter which is a simple storage adapter that stores data in a JSON formatted file on the hard disk. This functionality makes this storage adapter very good for testing and debugging.

We will select the Json File Storage Adapter by specifying it in our chat bot's constructor. The database parameter is used to specify the path to the database that the chat bot will use. The database.json file will be created automatically if it does not already exist.

ChatterBot's input adapters are designed to allow a chat bot to have a versatile method of receiving or retrieving input from a given source. It is required to add in parameters to specify the input and output terminal adapter. The input terminal adapter simply reads the user's input from the terminal.

4.3.2 INPUT ADAPTERS

The Chatterbot's input adapter class is an abstract class that represents the interface that all input adapters should implement. After getting input, the main job is the classify the text as a known or an unknown statement and pass it to the logic adapter after labeling the sentence as "known" or "unknown". The goal of an input adapter is to get input from some source, and then to convert it into a format that ChatterBot can understand. This format is the Statement object found in ChatterBot's conversation module [8].

We used the variable input adapter for the implementation of QA system. Variable input type adapter allows the chatbot to accept a number of different input types using the same adapter. This adapter accepts strings, dictionaries, and statements.

4.3.3 OUTPUT ADAPTERS

The output adapter allows the chatbot to return a response in as a Statement object. It is a generic class that can be overridden by a subclass to provide extended functionality, such as delivering a response to an API endpoint. Since our system is a text-based system we chose the "Text" format for our chatbot [8].

4.3.4 LOGIC ADAPTERS

Logic adapters determine the logic for how ChatterBot selects responses to a given input statement. The logic adapter that your bot uses can be specified by setting the logic_adapters parameter to the import path of the logic adapter you want to use [8].

It is possible to enter any number of logic adapters for your bot to use. If multiple adapters are used, then the bot will return the response with the highest calculated confidence value. If multiple adapters return the same confidence, then the adapter that is entered into the list first will take priority. The logic_adapters parameter is a list of logic adapters. In ChatterBot, a logic adapter is a class that takes an input statement and returns a response to that statement.

1. BestMatch logic adapter is a logic adapter that returns a response based on known responses to the closet matches to the input statement.
2. Import the Unicode literals and the logic adapter library.
3. In Get method, it takes a statement string and a list of statement strings and returns the closet matching statement from the list
4. It no statement have known responses then the Get method chooses random response to return. For random response, the Get method sets the confidence score as zero
5. For known statements(s), the Get method calculated the confidence score by doing "Jaccard Similarity" comparison and return the one with highest confidence score.

Figure 4.7: Pseudo code for the Best Match Logic Adapter

We employ Best Match Adapter for our chatbot. It is a logic adapter that returns a response based on known responses to the closest matches to the input statement. The Best Match logic adapter selects a response based on the best known match to a given statement. Once it finds the closest match to the input statement, it uses another function to select one of the known responses to that statement.

The best match adapter uses Jaccard Similarity function to compare the input statement to known statements. Jaccard Similarity compared two sentences based on Jaccard Index. The Jaccard index is a ratio of numerator and denominator or in other words, it is a fraction. In the numerator, we count the number of items that are shared between the sets. In the denominator, we

count the total number of items across both sets. Let's say we define sentences to be equivalent of 50% or more of their tokens are equivalent.

Here are two sample sentences:

The young cat is hungry. The cat is very hungry.

When we parse these sentences to remove stop words, we end up with the following two sets:

{young, cat, hungry} {cat, very, hungry}

In our example above, our intersection is {cat, hungry}, which has a count of two. The union of the sets is {young, cat, very, hungry}, which has a count of four. Therefore, our Jaccard similarity index is two divided by four, or 50%. Given our threshold above, we would consider this to be a match.

4.3.5 RESPONSE SECTION METHOD

Response selection methods determine which response should be used in the event that multiple responses are generated within a logic adapter. ChatterBot uses Statement objects to hold information about things that can be said. An important part of how a chat bot selects a response is based on its ability to compare two statements to each other. This module contains various text comparison algorithms designed to compare one statement to another.

We use the get first response method for the selection of a response. This method takes the input statement and selects the statement in the knowledge base which closely matches the input to the chatbot from a list of statement options to choose a response from.

4.3.6 STATEMENT-RESPONSE RELATIONSHIP

ChatterBot stores knowledge of conversations as statements. Each statement can have any number possible responses.

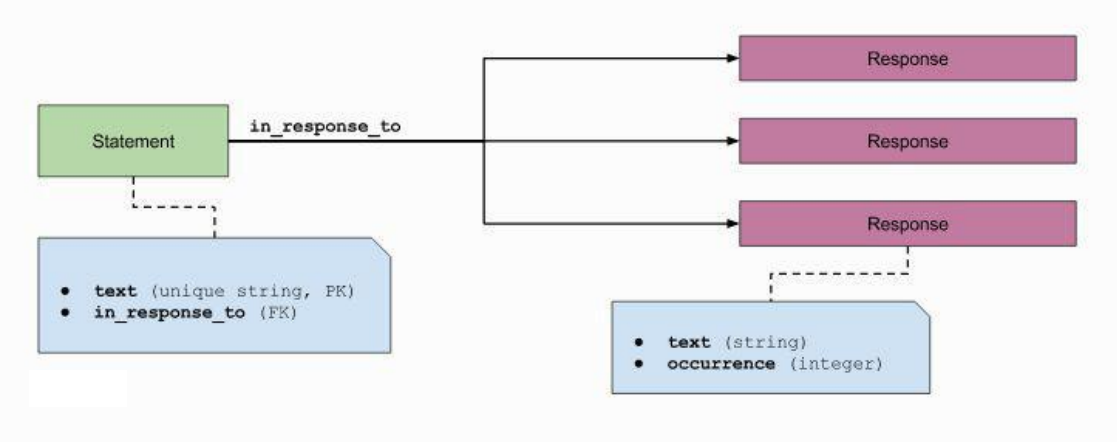


Figure 4.8: The Relationship between Statement and responses

Each Statement object has an in_response_to reference which links the statement to a number of other statements that it has been learned to be in response to.

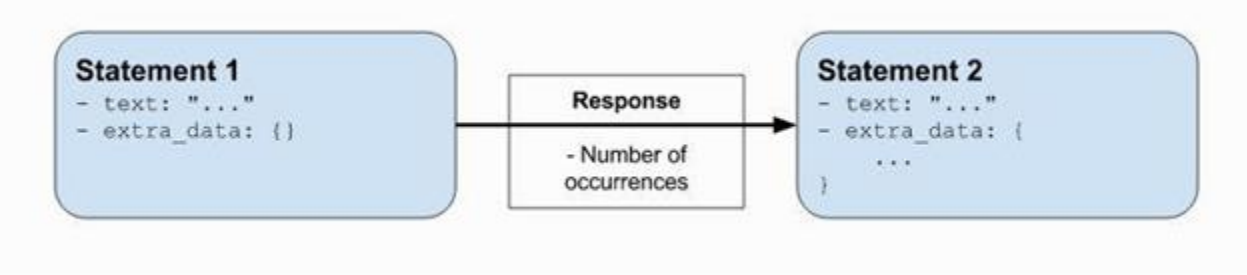


Figure 4.9: Mechanism of the reference to all parent statements of the current Statement

The in_response_to attribute is essentially a reference to all parent statements of the current statement. The Response object's occurrenceattribute indicates the number of times that the statement has been given as a response. This makes it possible for the chatbot to determine if a particular response is more commonly used than another.

4.4 ENVIRONMENT SETUP

Natural Language Processing (NLP) techniques such as Natural Language Toolkit (NLTK) for Python can be applied to analyze speech, and intelligent responses can be found by designing an engine to provide appropriate human-like responses[1]. For the setup of Bengali chatbot, we installed Python 3.6 in our Engine. Python is a high-level language which is suitable for scientific research.

Bengali[1][2]

Official Unicode Consortium code chart (PDF)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+098x	□	ঁ	ং	ঃ		অ	আ	ই	ঈ	উ	ঊ	ঋ	ঌ			এ
U+099x	ঐ			ও	ঔ	ক	খ	গ	ঘ	ঙ	চ	ছ	জ	ঝ	ঞ	ট
U+09Ax	ঠ	ড	ঢ	ণ	ত	থ	দ	ধ	ন		প	ফ	ব	ভ	ম	য
U+09Bx	র		ল				শ	ষ	স	হ			়	ঽ	া	ি
U+09Cx	ী	ু	ূ	ৃ	ৄ			ে	ৈ			ো	ৌ	্	ৎ	
U+09Dx								ৗ					ড়	ঢ়		য়
U+09Ex	ৠ	ৡ	ৢ	ৣ			০	১	২	৩	৪	৫	৬	৭	৮	৯
U+09Fx	ৰ	ৱ	৲	৳	৴	৵	৶	৷	৸	৹	৺	৻				

Notes

1.^ As of Unicode version 9.0

2.^ Grey areas indicate non-assigned code points

Figure 4.10: Official Unicode Consortium Code Chart of Bengali

With availability a large resource of libraries for research purpose, it is the best choice for the natural language processing research. Our chatbot is based on a machine learning engine called Chatterbot which is powered by Python. So to run Chatterbot in our machine it is mandatory to install Python.

Figure 4.11: Bengali Alphabet

Python 3.6 is recommended for the implementation of Bengali chatbot because any other version below 3.6 of Python causes “Unicode Decode Error”. Unicode Decode Error is a runtime error caused by non-English language with a large number of letters in the alphabet. The Unicode range of Bengali is 0980–09FF. It has 11 vowels and 40 consonants. Unlike English, it has consonant conjuncts, modifier, and other graphemes. So Bengali cannot be dealt with ASCII decoding. For easy installation of Chatterbot, it is recommended to install Anaconda for the setup. Anaconda8 is an open source data science platform powered by Python. Only Python 3.6 and Anaconda 3 supports taking the input in Bengali from a database. So it is advisable to use Python 3.6 and Anaconda 3 for this purpose.

There is required software to run Chatterbot in any engine. We installed chatterbot-corpus for the implementation of the English Chatbot. It is notable to mention that after implementation of model we add the Bangla corpus to the chatterbot corpus. So due to our contribution, anyone installing Chatterbot corpus will also get a sample Bengali corpus made by own.

4.5 IMPLETENTATION

For the implementation of our QA system, we have to go through some steps sequentially.

i. First of all we installed Linux operating system.
ii. The required environmental setup to run the Chatterbot library has been done in our laboratory.
iii. We installed the required softwares, libraries and toolkits as per the requirement.
iv. After that, we tested some web chatbots to get familiar with the system. Then prepared the codes to and corpus for building up the knowledge base of the system. There is a particular format to input data in the JSON storage so we have to format the corpus in that format.
v. We write a program which simulates our system.
vi. Then we train the system with our own BENGALIDATASET.
vii. After successful implementation of the system, we tested our system with different questions.

Thus we have done the implementation of our QA system. The main focus of our work is to generate sentences free from grammatical mistakes, spelling mistakes and consistent. Our System achieves the goal of producing correct responses. We also integrated this system in a personal blog. In the process we made a small Bengali corpus. Since the responses are as good as its knowledge base so a lot of work has to be done to enhance the knowledge base. Topic-wise data can be fed to our system for the enhancement of its knowledge base. But during building the knowledge base, the developer must provide a knowledge base free from errors.

CHAPTER 5

SIMULATION AND RESULT

In this chapter we have described the results of the whole experiments. This also included the description of the hardware, software, libraries used for getting results, amount of data used for training, accuracy and the system's overall performance.

5.1 HARDWARE USED FOR SIMULATION:

As the process of training the neural network is time consuming, so it's requires the hardware which is more capable in computing with having high configuration. We have run our model on a machine with the following specification:

i. Intel® Core™ i5-7200U CPU, 2.71GHz × 8

ii. 8 GB RAM

iii. NVidia 940mx 2GB GPU

As the configuration of our machine not so much good that's why during the training session we can't perform any other activity on my machine. Because the entire memory was occupied for training the system. Training the data took about 8 days.

5.2 SOFRWARE AND LIBRARIES USED FOR SIMULATION

We have to install some software in our system to implement the QA system. The system has been implemented on LIUX operating system. The following softwares were required:

i. Linux OS
ii. Python 3
iii. PyCharm
iv. Visual Studio
v. Tensorflow GPU

We also used some python libraries to get our system ready for the work

i. Keras
ii. Tensorflow
iii. Conda
iv. NLTK toolkit
v. Chatterbot
vi. Tokenizer

5.3 RESULT

The difficulty of evaluation is intrinsic as each conversation is interactive, and the same conversation will not occur more than once; one slightly different answer will lead to a completely different conversation; moreover, there is no clear sense of when such a conversation is "complete" [23]. So for the evaluation, we decided to compare our system with previous existing chatbots.

We have trained the system with REDDITDATASET which consists of 1000000+ question and answers. Training session was taking about 18days to train the system. The learning rate is .001. batch size is 1000. That means in each epoch the system will take 1000 question/answer to train the system. The report step is 20000. That means in every 8-reporting step the system completes its one epoch.

After this primary training we made our own Bengali corpus consists of 1000 data. We divided the train and test data by 60 and 40 percent respectively. Then we trained the model with our corpus. The reporting step was 50. And we needed 20 epoch.

The result shows the lost accuracy during training session. After the training we have examine out system with different types of questions. In case of generating output there are two things, one is understanding the question from input of the user. And another one is giving answer from system learning as trained. Our system can successfully give answer to any question but not in Bengali. After successfully understanding the question the system is giving answers very fast. But sometime in case of answering, the system was confused. It is taking inputs in Bengali but giving answers in English. We implemented tokenizer for machine translation but didn.t get any improvement. This can be a work for the future research. The result for question answering is given below.

5.3.1 SCREENSHOTS OF THE TEST RESULT

Figure 5.1: Snapshot of the response

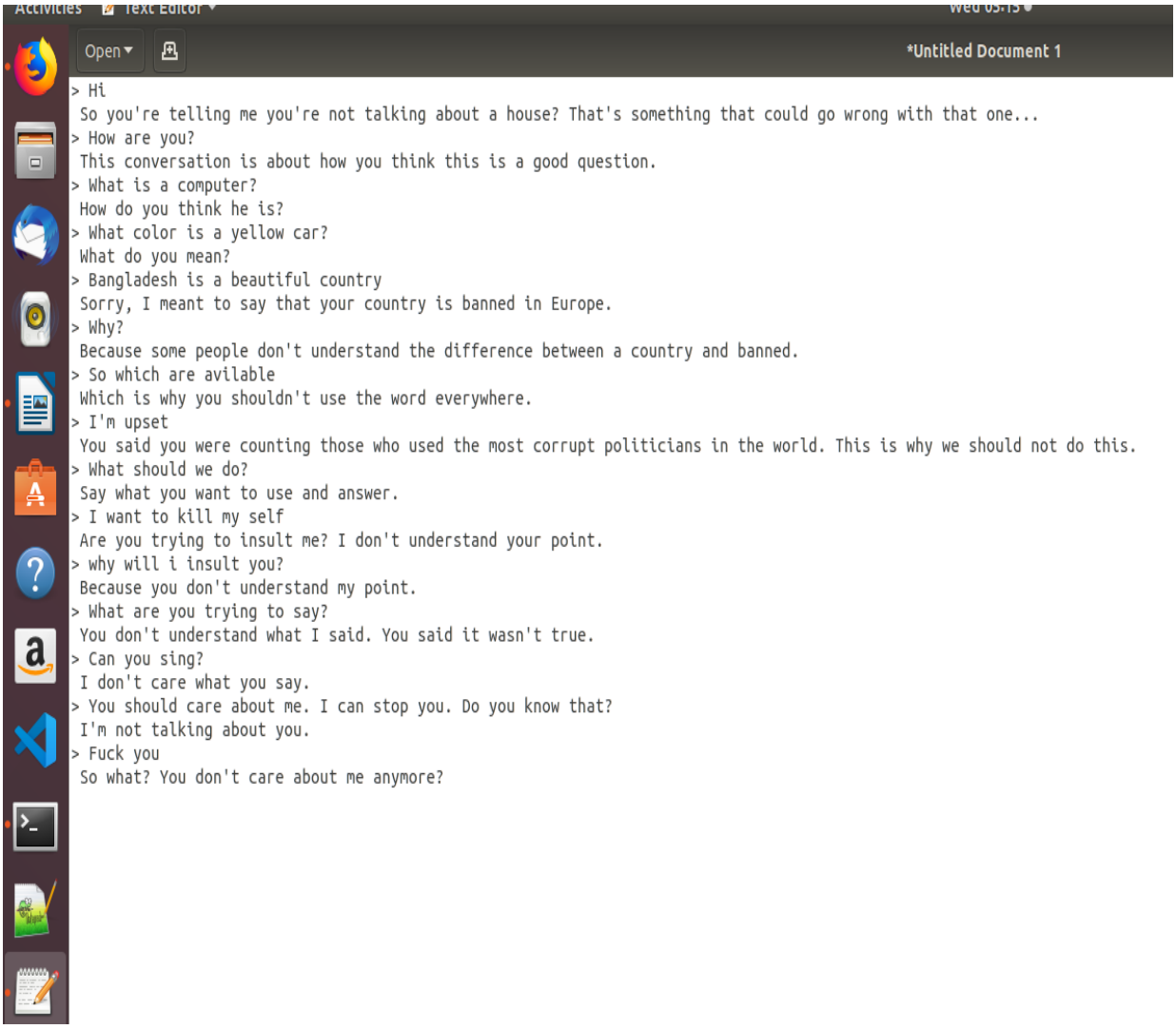


Figure 5.2: Snapshot of the response

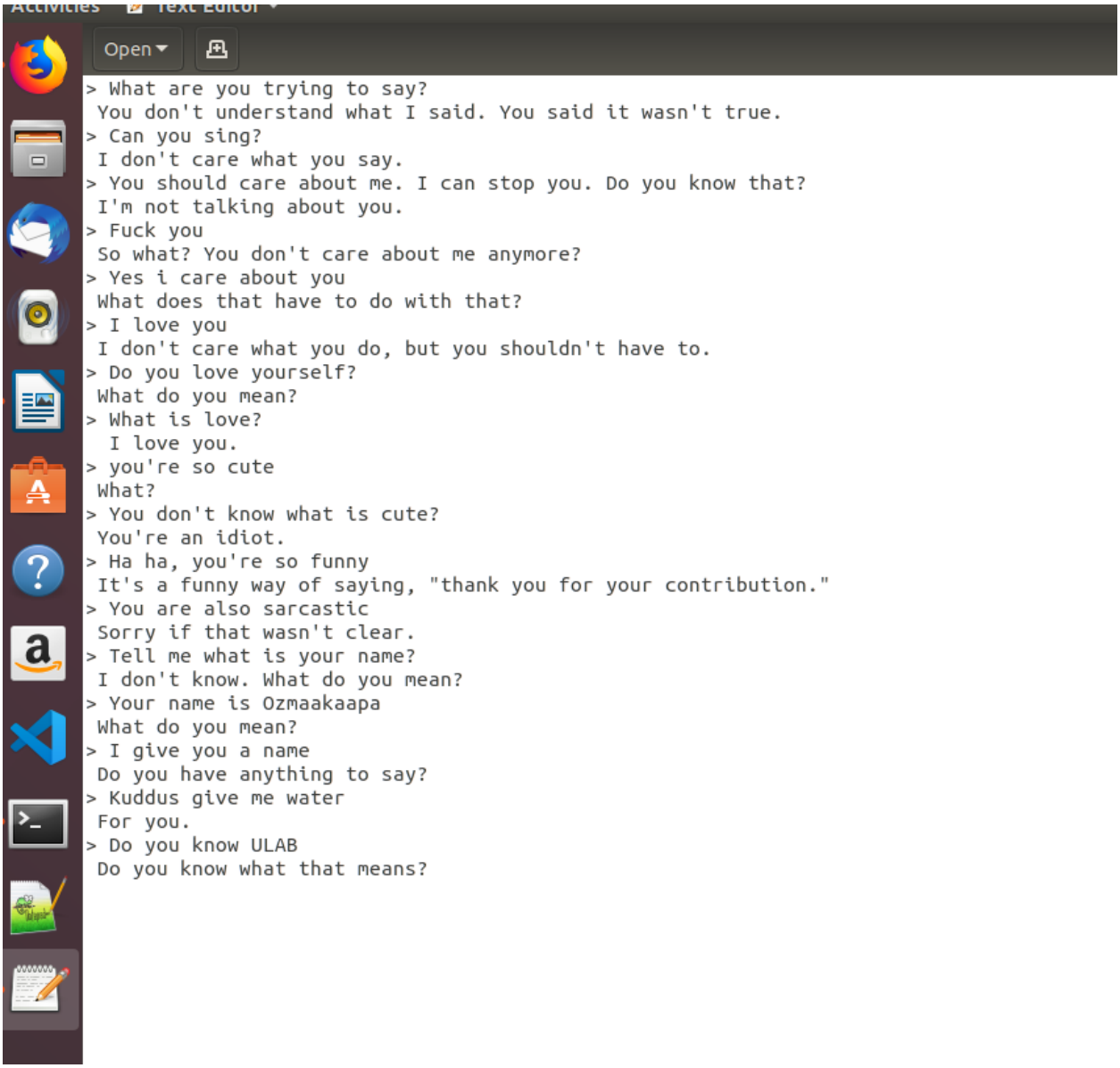


Figure 5.3: Snapshot of the response

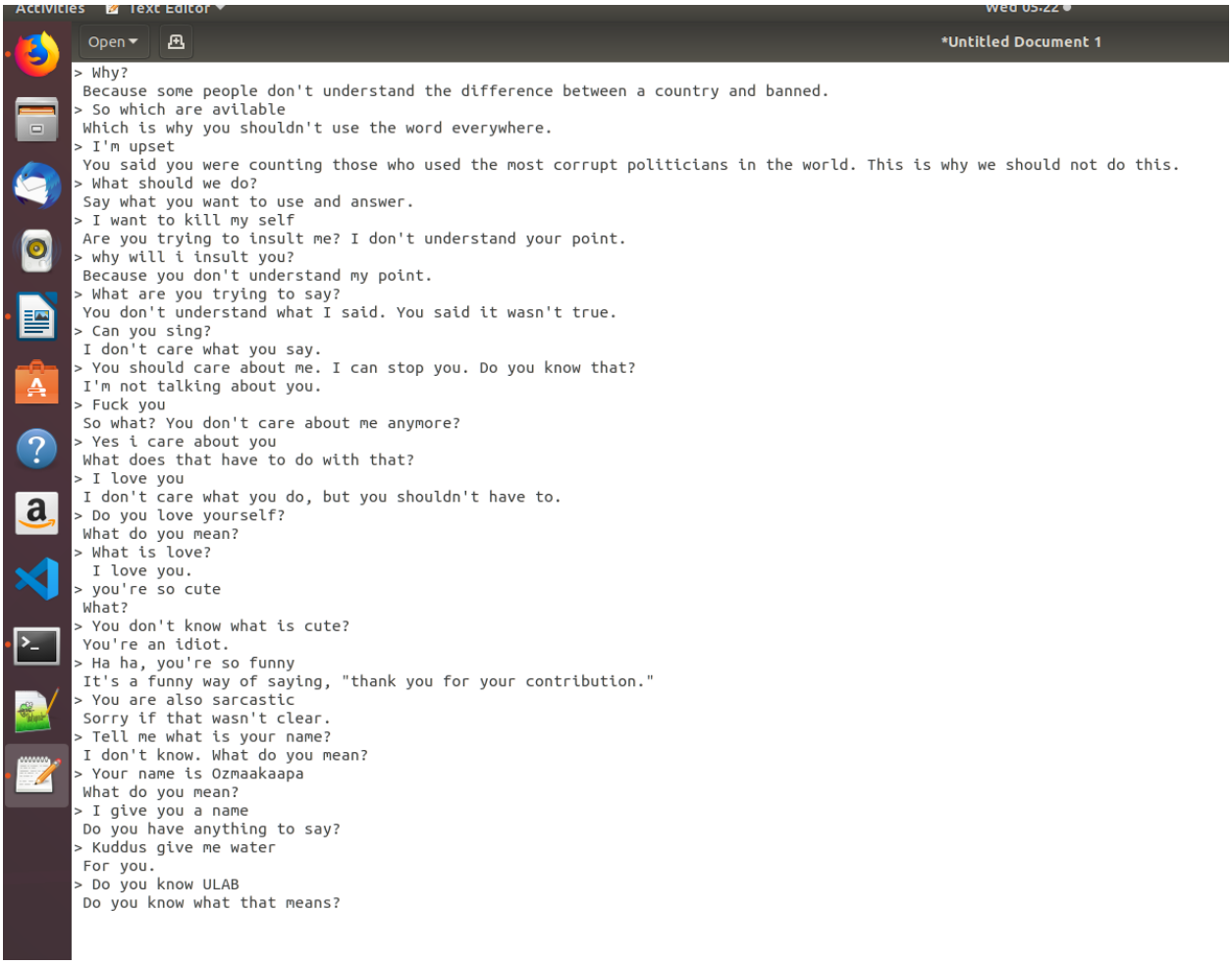


Figure 5.4: Snapshot of the response

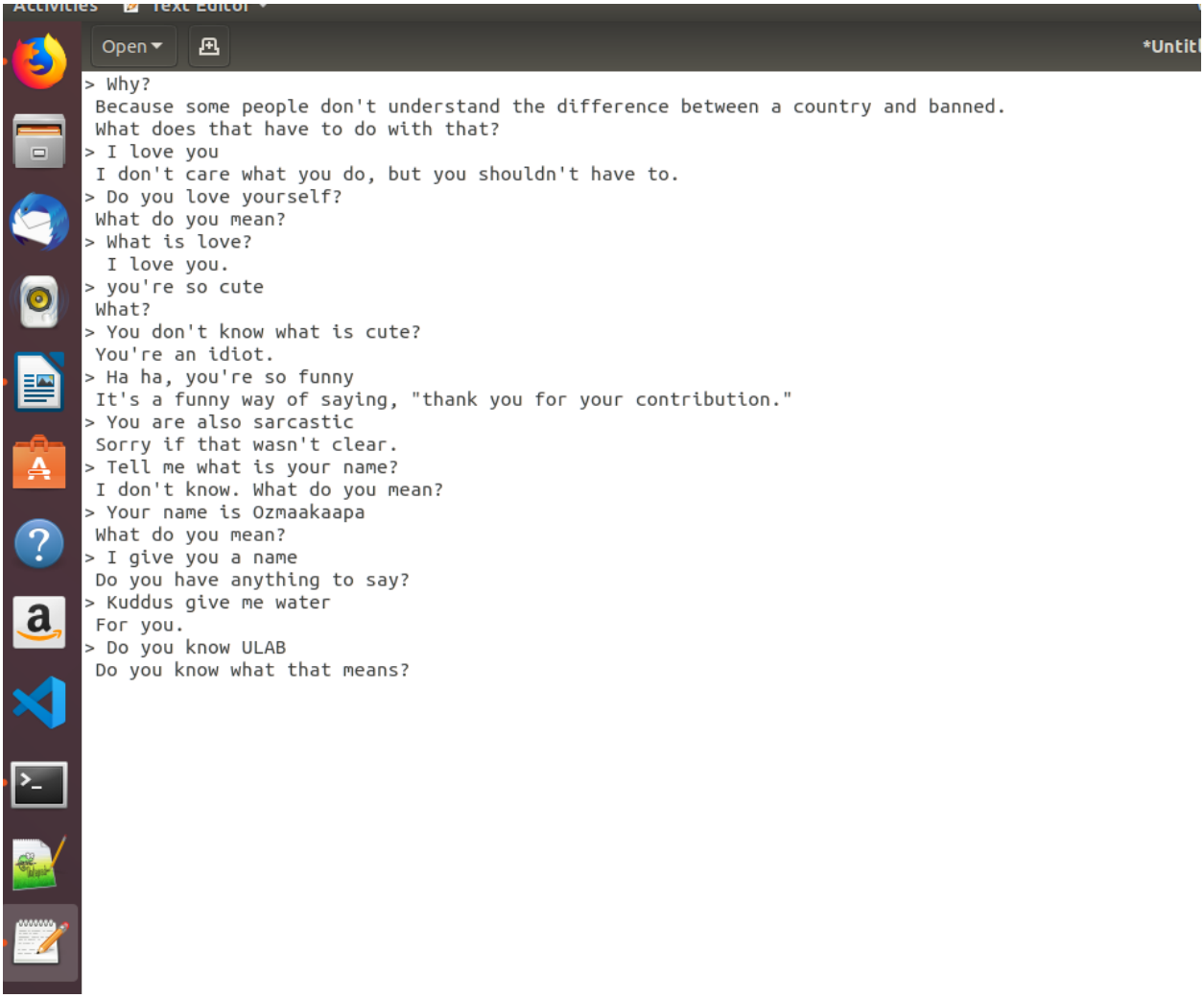


Figure 5.5: Snapshot of the response

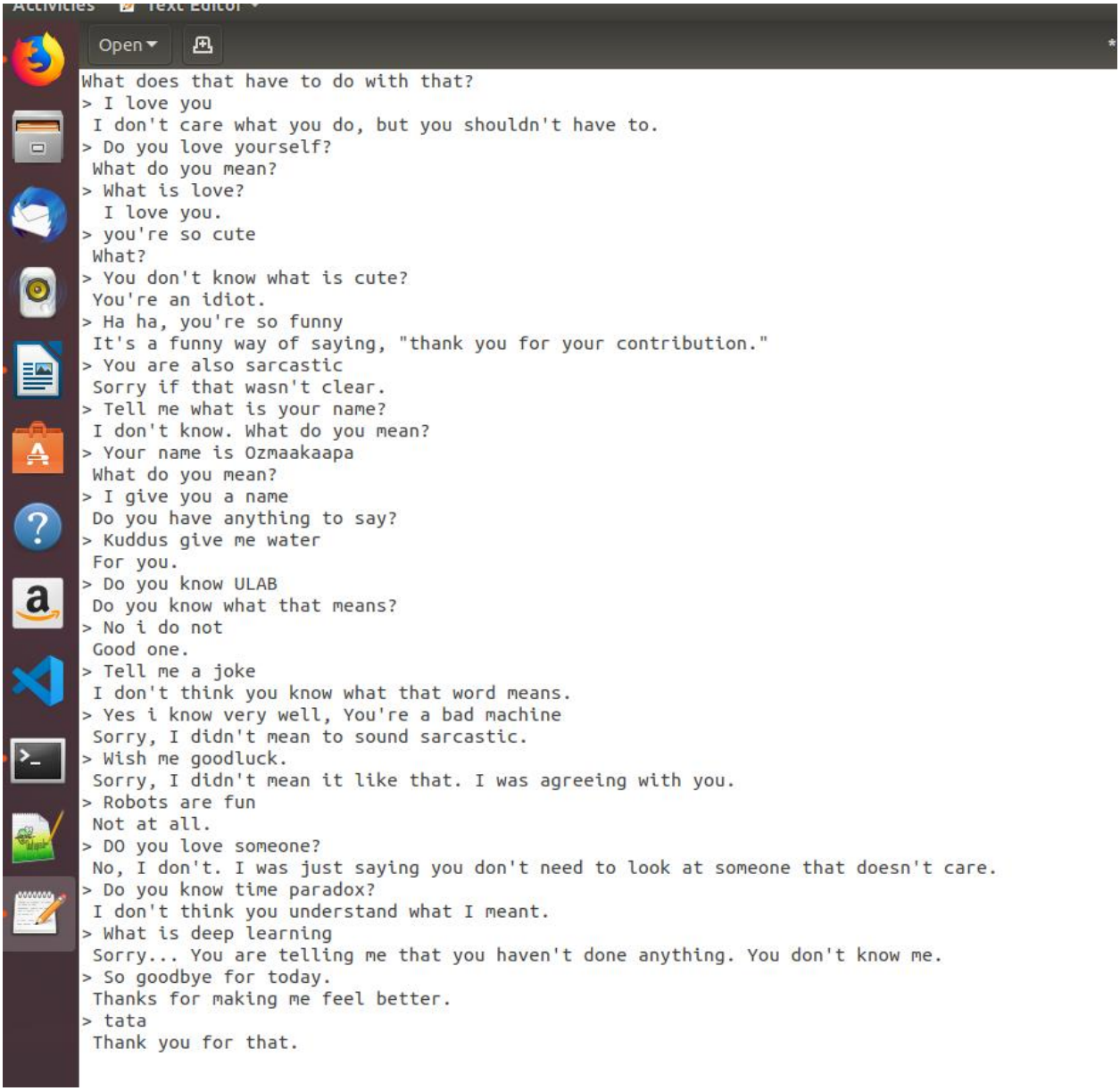


Figure 5.6: Snapshot of the response

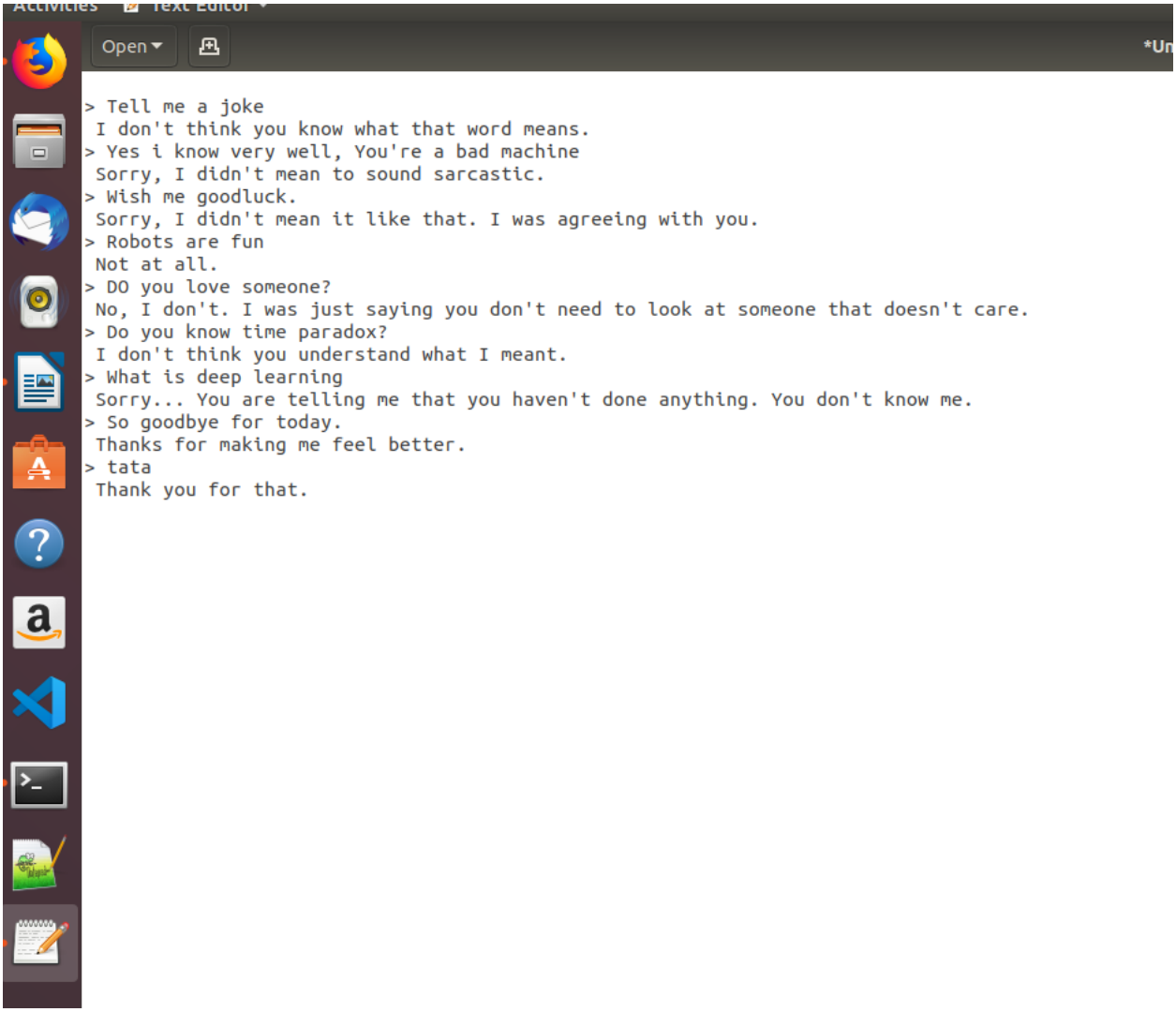


Figure 5.7: Snapshot of the response

5.4 ANALYSIS

Since we do not have more time to modify our model to perform better, we have to get satisfied with our current performance. Our work addresses the problem of developing a QA system in spite of required language processing tools like Parts Speech Tagger, Tokenizer etc. We solve the problem of lack of required tools by selecting a language independent platform and choosing a retrieval based model to serve the purpose. So we are left with an option of comparing it with any English chatbot. Therefore, we compare QAs with two popular chatbots which are Neural Conversational Model (NCM) and the Cleverbot.

To ensure a fair comparison, the questions asked exact questions asked in both cases. NCM is a neural based open domain generative based chatbot where ours is a retrieval based closed domain one. In the experiments, our chatbot gives the similar response as the NCM. Cleverbot is a chatbot hosted on a website who learns from the user and answers based on the conversation history which is quite similar to our work. It is interesting to observe that our system outwitted Cleverbot in many cases. We examine QAs by inputting unknown sentences as a test case and find QAs to produce random answers to the questions. But it stores the reply given by the user to the unknown sentences and later on replies the same answer to another user in another instance. QAs is able to reply in real time like others. Since it can take input in English and can give a response, so we can say that the pattern matching algorithm is functioning well. Our chatbot replies in syntactically correct and it is free from spelling mistakes and any sort of grammatical mistakes. It makes some punctuation mistake which can be improved in future.

From the samples, we can see our QAs gives a similar reply like Neural Conversational Machine (NCM) whereas in comparison with a related work Cleverbot our QAs has outwitted it in most of the instance. Amongst the many limitations, the lack of a coherent personality makes it difficult for our system to pass the Turing test [15].

Our work provides us a conversation corpus. This generation of the corpus has many advantages. Corpus is considered as a basic resource for language analysis and research for many foreign languages. This reflects both ideological and technological change in the area of language

research. This change is probably caused due to the introduction of computer and corpus in linguistic research which, as a result, have paved out many new applications of language (and linguistics) in the fields of communication and information exchange. This corpus can be useful for producing many sophisticated automatic tools and systems, besides being good resources for language description and theory making.

This QA system can automatically answer question asked by the user using deep learning technique and give answers to the question. We have trained our model with lots of questions then we ask the system to get the result. It can outcome the question with expected result. Though in some cases the prediction goes little bit wrong but it can be reduced by fine tuning the system in near future with other datasets available.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

QA systems have the ability to model natural language and establish a conversation with a user through a question/answer protocol. There are three types of approaches depending on the freedom they have at the time of answering: rule-based, retrieval-based and generative based. The two first approaches are the most used nowadays due to its effectiveness at the time of maintaining a close-domain conversation. The generative-based models, on the other hand, arise as a powerful alternative in the sense that they can handle better an open topic conversation. They are much related to the idea of strong AI, no human intervention at the time of answering, everything is learned by the machine. Promising results have been achieved in generative-based chatbot models by applying neural translation techniques with encoder/decoder architectures. In this thesis, it has been shown that chatbot models based on encoder/decoder architectures using exclusively attention outperforms neural network models. It is important to mention that all models shown in this project shape and mimic natural human language but do not apply any logic to their answers. That is why most of the answers are not coherent between them and the final model lacks of a "personality".

The main focus of our work is to generate sentences free from grammatical mistakes, spelling mistakes and consistent. Our System achieves the goal of producing correct responses. Since the responses are not good as its knowledge base so a lot of work has to be done to enhance the knowledge base. Topic-wise data can be fed to our system for the enhancement of its knowledge base. But during building the knowledge base, the developer must provide a knowledge base free from errors.

6.2 LIMITATIONS

QA systems can enable users to access the knowledge in a natural way by asking natural language questions and get back relevant correct answers. The major challenges in QASs are: understanding natural language questions regardless of their types or representation; understanding knowledge derived from the structured, semi structured, un-structured to semantic web and searching for the relevant, correct and concise answers that can satisfy the information needs of users. We face some limitations in our system. It is answering the questions in English where as it should have answered in Bengali.

6.3 FUTURE WORK

There are some huge scope of improvement of the system. For example, making a larger Bengali Corpus and train the model with that corpus. For future work, one aspect to enhance the knowledge base of this system is to host it on a crowdsourcing platform. Like Google Translate Bengali, QA will be able to learn from the interactions with users. The more the number of interactions, the more will be the percentage of relevant replies to a query can be provided by QA system.

In future, we can train a chatbot with a neural network model provided with the Bengali corpus when available. Also, we can try a crowd-sourced model to enrich the database of the chatbot by integrating this chatbot in a website.

The context can be incorporated in future work. To produce sensible responses, systems may need to incorporate both linguistic context and physical context. In long conversations, people keep track of what has been said and what information has been exchanged. Since it requires a large collection of conversation corpus to make a generative open domain system to incorporate context. As we do not have that data to make it generative so this can be done in future. We can optimize the automated system by making the chatbot a voice-enabled system, to reply in pictorial representation for better understanding for people with low literacy like “Sophia”.

REFERENCES

[1] Abdul-Kader, S. A., & Woods, J. (2015). Survey on chatbot design techniques in speech conversation systems. Int. J. Adv. Comput. Sci. Appl.(IJACSA), 6(7).

[2] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. 2014. Convolutional neural network architectures for matching natural language sentences. In Advances in Neural Information Processing Systems, pages 2042–2050.

[3] Beech, H. (2014). What's All The Fuss About Whats App? China's WeChat Is a Worthy Rival. Time. com, 2, 1.

[4] Berger A, Caruana R, Cohn D, Freitag D, and Mittal V. Bridging the lexical chasm: statistical approaches to answer-finding. In Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval, 2000, pp. 192-199.

[5] Colby, K. 1999a. Comments on the human-computer conversation. In Wilks, Y. (ed.), Machine Conversations. Kluwer, Boston/Drdrecht/London, pp. 5-8.

[6] Hao Wang, Zhengdong Lu, Hang Li, and Enhong Chen. 2013. A dataset for research on short text conversations. In EMNLP, pages 935–945.

[7] Greenwood M. and Gaizauskas R. Using a Named Entity Tagger to Generalise Surface Matching Text Patterns for Question Answering. In Proceedings of the Workshop on Natural Language Processing for Question Answering (EACL03), 2003, pp. 29-34.

[8] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2016. A persona based neural conversation model. arXiv preprint arXiv:1603.06155

[9] Islam, Z., Mehler, A., Rahman, M. R., & Text technology, A. G. (2012, November). Text Readability Classification of Textbooks of a Low-Resource Language. In PACLIC (pp. 545-553)

[10] Kwok C, Etzioni O and Weld DS. Scaling question answering to the Web. ACM Transactions on Information Systems (TOIS), Vol.19(3), 2001, pp. 242-262.

[11] M. J. Pereira, and L. Coheur, "Just. Chat-a platform for processing information to be used in chatbots," 2013.

[12] Meeng, M., & Knobbe, A. (2011). Flexible enrichment with Cortana–software demo. In Proceedings of BeneLearn (pp. 117-119).

[13] Moschitti A. Answer filtering via text categorization in question answering systems. In Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence, 2003, pp. 241-248.

[14] Processing and Information Systems, Springer Berlin Heidelberg, 2002, pp. 235-239.

[15] Ravichandran D and Hovy E. Learning surface text patterns for a question answering system. In proceeding of 40th Annual Meeting on Association of Computational Linguistics, 2002, pp. 41-47.

[16] Ritter, A., Cherry, C., & Dolan, B. (2010, June). Unsupervised modeling of twitter conversations. In Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (pp. 172-180). Association for Computational Linguistics.

[17] Saenz, A. (2010). Cleverbot chat engine is learning from the internet to talk like a human. Singularity Hub

[18] Shawar, B.A. and Atwell, E. (2007) Chatbots: are they really useful? Journal of Computational Linguistics and Language Technology, Vol. 22, No. 1, pp.29-49.

[19] Shang, L.; Lu, Z.; and Li, H. 2015. Neural responding machine for short-text conversation. In ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers, 1577–1586.

[20] Stent, A., & Bangalore, S. (Eds.). (2014). Natural language generation in interactive systems. Cambridge University Press.

[21] Shawar, B. A., & Atwell, E. (2007, April). Different measurements metrics to evaluate a chatbot system. In Proceedings of the Workshop on Bridging the Gap: Academic and Industrial Research in Dialog Technologies (pp. 89-96). Association for Computational Linguistics.

[22] Sneiders E. Automated question answering using question templates that cover the conceptual model of the database. In Natural Language

[23] Soubbotin MM and Soubbotin SM. Patterns of Potential Answer Expressions as Clues to the Right Answer. In Proceeding of the TREC-10, NIST, 2001, pp. 175-182.

[24] Steve Young, Milica Gasiˇ c, Simon ´ Keizer, Franc¸ois Mairesse, Jost Schatzmann, Blaise Thomson, and Kai Yu. 2010. The hidden information state model: A practical framework for pomdp-based spoken dialogue management. Computer Speech & Language, 24(2):150–174.

[25] Turing, A. M. (1950). Computing machinery and intelligence. Mind, 59(236), 433-460.

[26] Vinyals, O., & Le, Q. (2015). A neural conversational model. arXiv preprint arXiv:1506.05869.

[27] Weizenbaum, J. (1966). ELIZA—a computer program for the study of natural language communication between man and machine. Communications of the ACM, 9(1), 36-45.

[28] Wilks, Y. (1999). Preface. In Wilks, Y., editor, Machine Conversations, pages vii–x. Kluwer, Boston/-Dordrecht/London

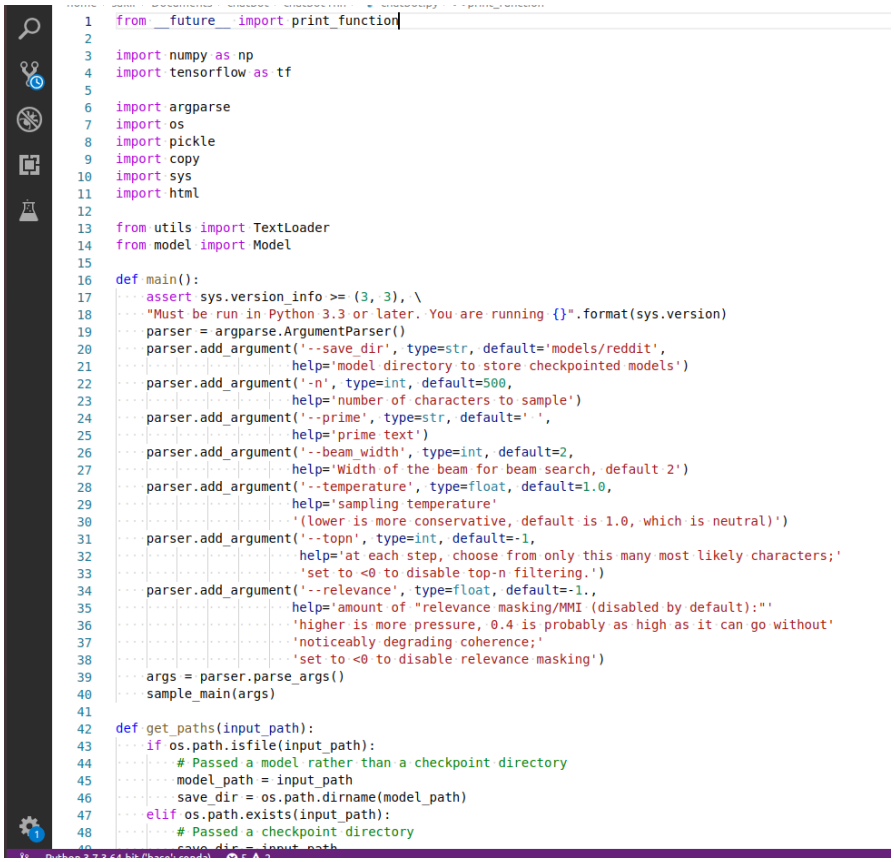
[29] Wu, Y., Wu, W., Li, Z., & Zhou, M. (2016). Response Selection with Topic Clues for Retrieval-based Chatbots. arXiv preprint arXiv:1605.00090.

[30] Yu, Z., Xu, Z., Black, A., & Rudnicky, A. (2016). Chatbot evaluation and database expansion via crowdsourcing. In Proc. of the chatbot workshop of LREC.

[31] Zhang D and Lee WS. Web based pattern mining and matching approach to question answering. In Proceedings of the 11th Text REtrieval Conference, 2002.
[32] Zhou, X., Dong, D., Wu, H., Zhao, S., Yan, R., Yu, D., ... & Tian, H. (2016). Multi-view response selection for human-computer conversation. EMNLP'16.
[33] Zongcheng Ji, Zhengdong Lu, and Hang Li. 2014. An information retrieval approach to short text conversation. arXiv preprint arXiv:1408.6988

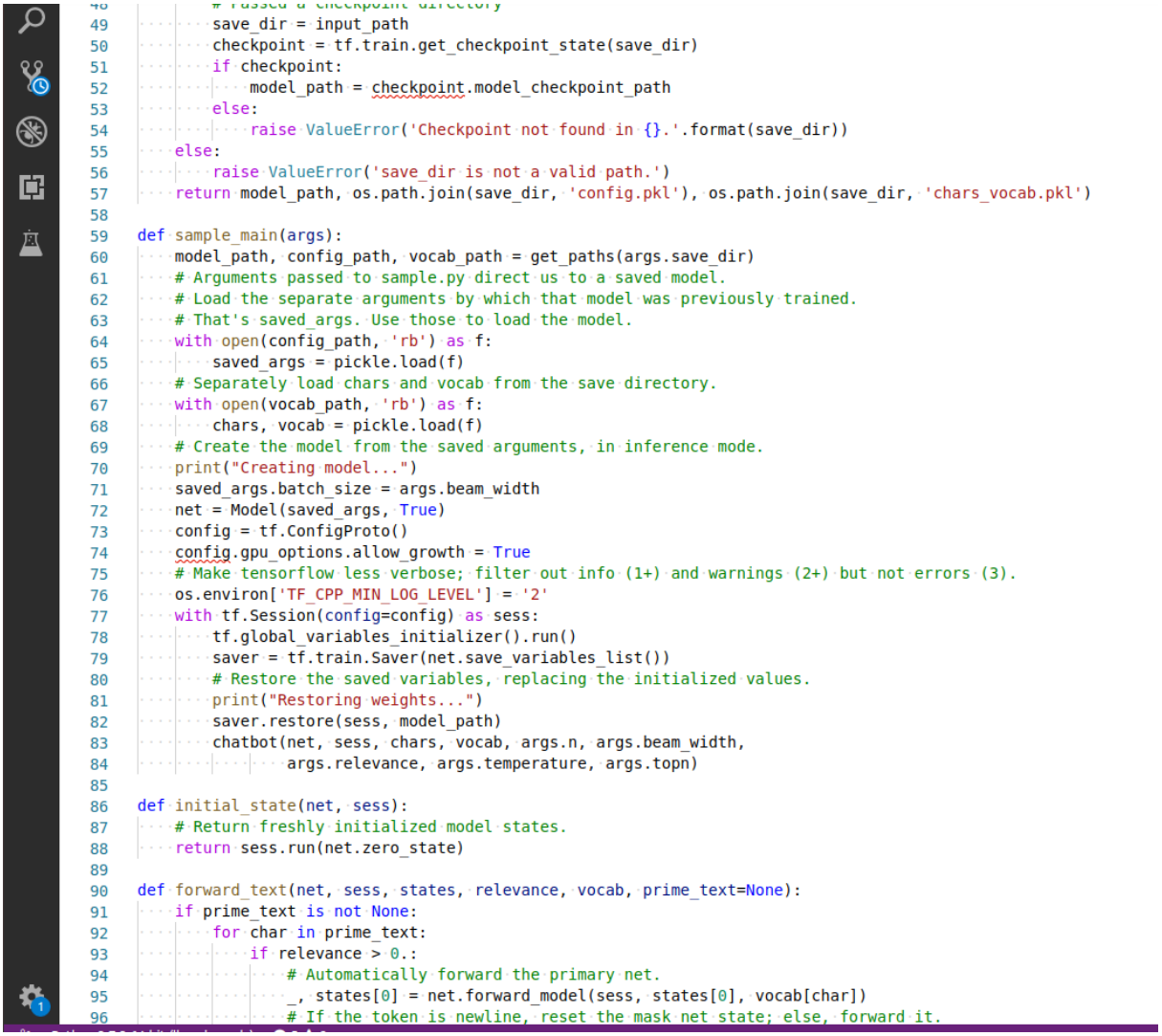
APPENDIX A

In Appendix A, the code of our system has been shown. The codes has been uploaded in a GIT repository. The following link will guide you to the repo: https://github.com/sakifabir/chatbot.



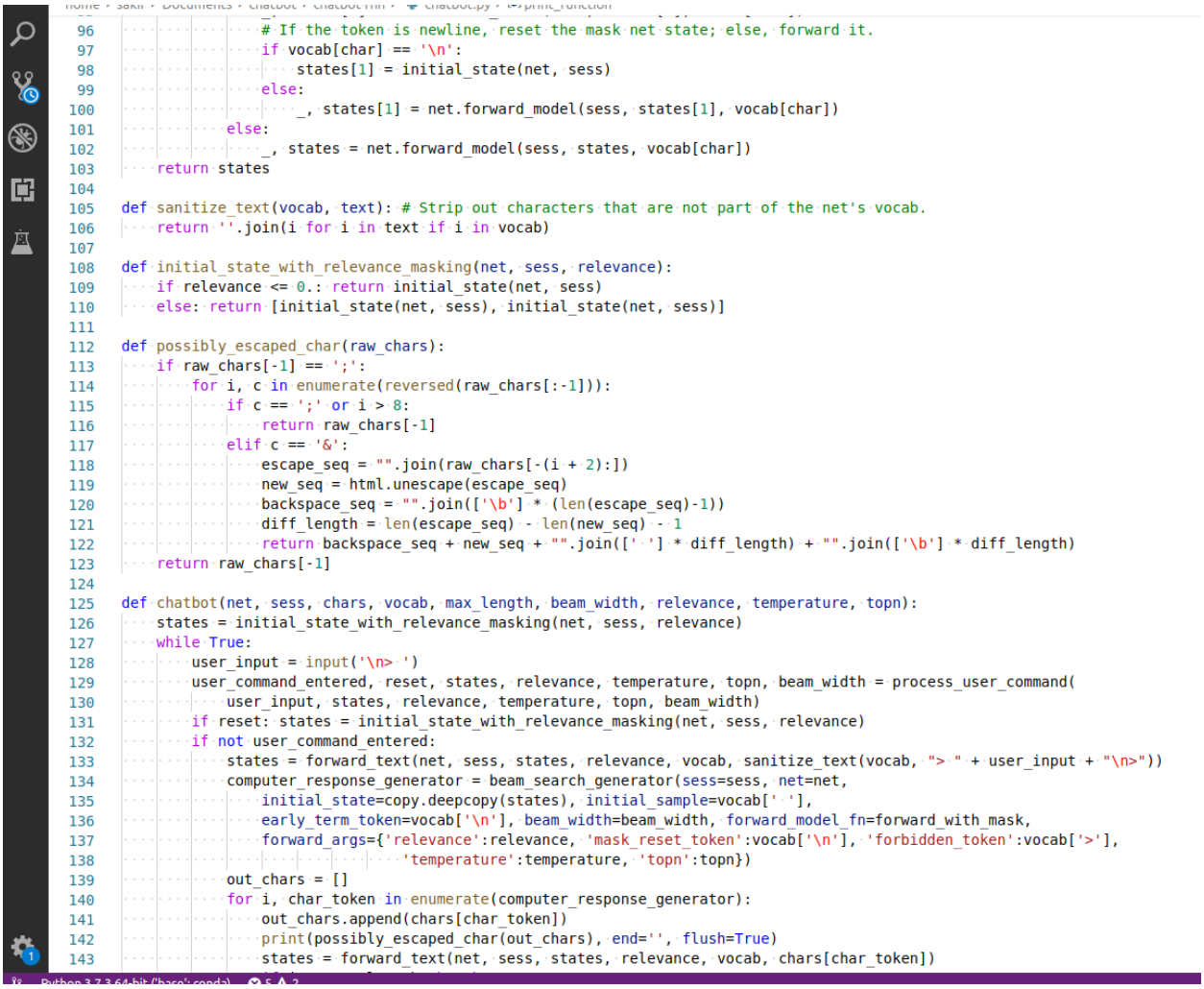
```
from __future__ import print_function

import numpy as np
import tensorflow as tf

import argparse
import os
import pickle
import copy
import sys
import html

from utils import TextLoader
from model import Model

def main():
    assert sys.version_info >= (3, 3), \
    "Must be run in Python 3.3 or later. You are running {}".format(sys.version)
    parser = argparse.ArgumentParser()
    parser.add_argument('--save_dir', type=str, default='models/reddit',
                        help='model directory to store checkpointed models')
    parser.add_argument('-n', type=int, default=500,
                        help='number of characters to sample')
    parser.add_argument('--prime', type=str, default=' ',
                        help='prime text')
    parser.add_argument('--beam_width', type=int, default=2,
                        help='Width of the beam for beam search, default 2')
    parser.add_argument('--temperature', type=float, default=1.0,
                        help='sampling temperature'
                        '(lower is more conservative, default is 1.0, which is neutral)')
    parser.add_argument('--topn', type=int, default=-1,
                        help='at each step, choose from only this many most likely characters;'
                        'set to <0 to disable top-n filtering.')
    parser.add_argument('--relevance', type=float, default=-1.,
                        help='amount of "relevance masking/MMI (disabled by default):"'
                        'higher is more pressure, 0.4 is probably as high as it can go without'
                        'noticeably degrading coherence;'
                        'set to <0 to disable relevance masking')
    args = parser.parse_args()
    sample_main(args)

def get_paths(input_path):
    if os.path.isfile(input_path):
        # Passed a model rather than a checkpoint directory
        model_path = input_path
        save_dir = os.path.dirname(model_path)
    elif os.path.exists(input_path):
        # Passed a checkpoint directory
```

Figure A.1: Snapshot of the chat.py file



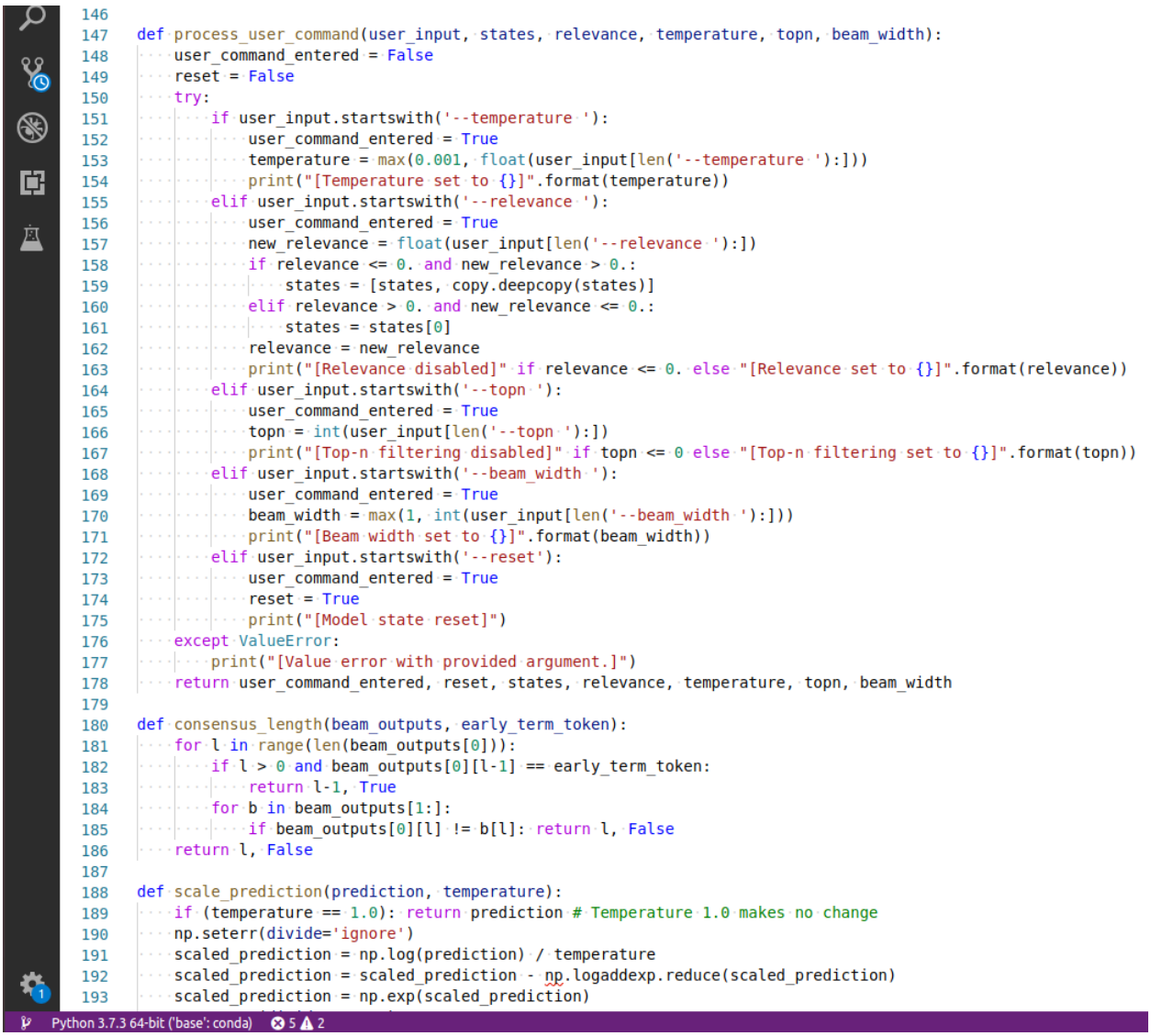
```
            # Passed a checkpoint directory
            save_dir = input_path
            checkpoint = tf.train.get_checkpoint_state(save_dir)
            if checkpoint:
                model_path = checkpoint.model_checkpoint_path
            else:
                raise ValueError('Checkpoint not found in {}.'.format(save_dir))
        else:
            raise ValueError('save_dir is not a valid path.')
        return model_path, os.path.join(save_dir, 'config.pkl'), os.path.join(save_dir, 'chars_vocab.pkl')

    def sample_main(args):
        model_path, config_path, vocab_path = get_paths(args.save_dir)
        # Arguments passed to sample.py direct us to a saved model.
        # Load the separate arguments by which that model was previously trained.
        # That's saved_args. Use those to load the model.
        with open(config_path, 'rb') as f:
            saved_args = pickle.load(f)
        # Separately load chars and vocab from the save directory.
        with open(vocab_path, 'rb') as f:
            chars, vocab = pickle.load(f)
        # Create the model from the saved arguments, in inference mode.
        print("Creating model...")
        saved_args.batch_size = args.beam_width
        net = Model(saved_args, True)
        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True
        # Make tensorflow less verbose; filter out info (1+) and warnings (2+) but not errors (3).
        os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
        with tf.Session(config=config) as sess:
            tf.global_variables_initializer().run()
            saver = tf.train.Saver(net.save_variables_list())
            # Restore the saved variables, replacing the initialized values.
            print("Restoring weights...")
            saver.restore(sess, model_path)
            chatbot(net, sess, chars, vocab, args.n, args.beam_width,
                    args.relevance, args.temperature, args.topn)

    def initial_state(net, sess):
        # Return freshly initialized model states.
        return sess.run(net.zero_state)

    def forward_text(net, sess, states, relevance, vocab, prime_text=None):
        if prime_text is not None:
            for char in prime_text:
                if relevance > 0.:
                    # Automatically forward the primary net.
                    _, states[0] = net.forward_model(sess, states[0], vocab[char])
                    # If the token is newline, reset the mask net state; else, forward it.
```

Figure A.2: Snapshot of the chat.py file



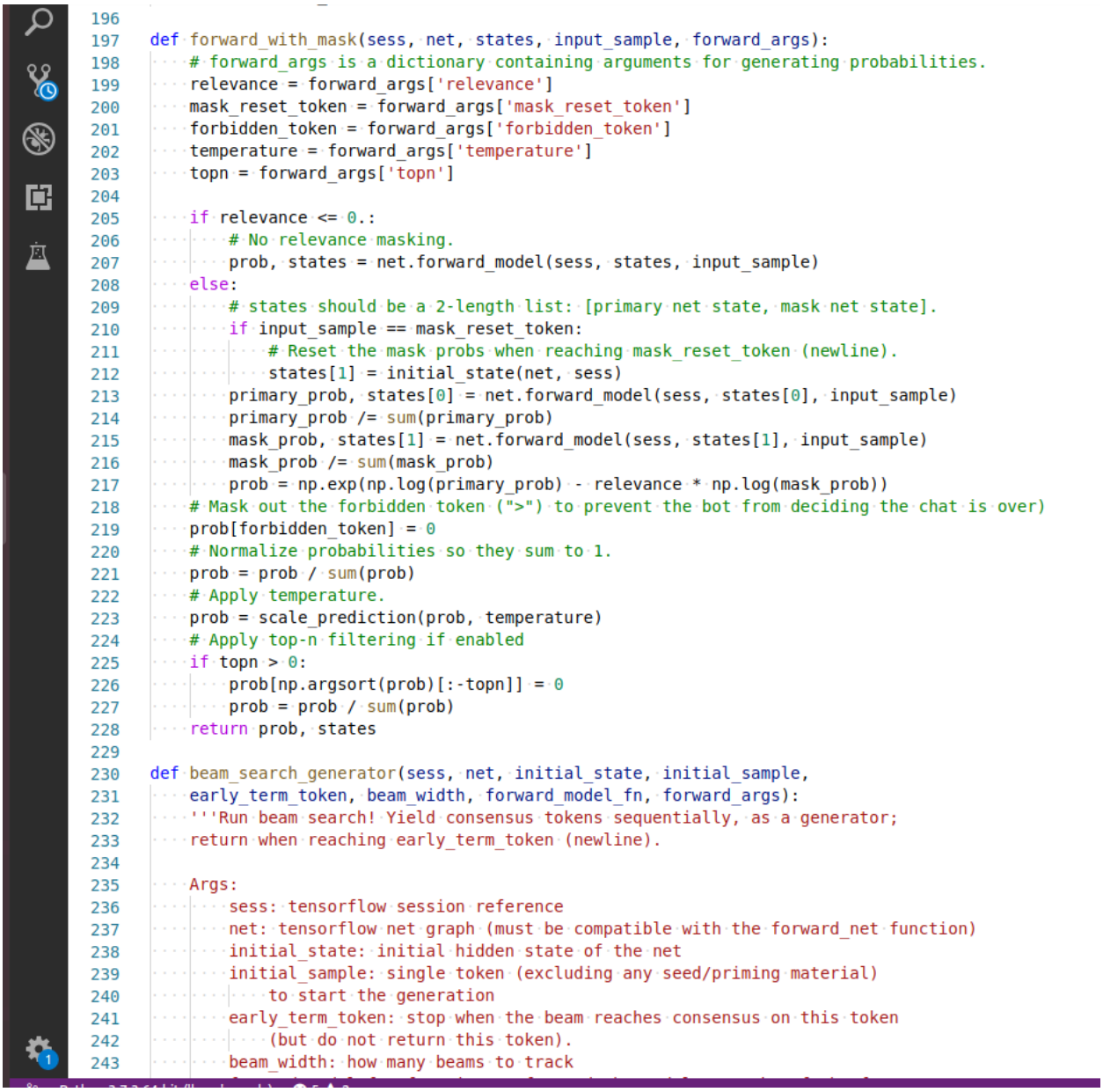
```
                # If the token is newline, reset the mask net state; else, forward it.
                if vocab[char] == '\n':
                    states[1] = initial_state(net, sess)
                else:
                    _, states[1] = net.forward_model(sess, states[1], vocab[char])
            else:
                _, states = net.forward_model(sess, states, vocab[char])
    return states

def sanitize_text(vocab, text): # Strip out characters that are not part of the net's vocab.
    return ''.join(i for i in text if i in vocab)

def initial_state_with_relevance_masking(net, sess, relevance):
    if relevance <= 0.: return initial_state(net, sess)
    else: return [initial_state(net, sess), initial_state(net, sess)]

def possibly_escaped_char(raw_chars):
    if raw_chars[-1] == ';':
        for i, c in enumerate(reversed(raw_chars[:-1])):
            if c == ';' or i > 8:
                return raw_chars[-1]
            elif c == '&':
                escape_seq = "".join(raw_chars[-(i + 2):])
                new_seq = html.unescape(escape_seq)
                backspace_seq = "".join(['\b'] * (len(escape_seq)-1))
                diff_length = len(escape_seq) - len(new_seq) - 1
                return backspace_seq + new_seq + "".join([' '] * diff_length) + "".join(['\b'] * diff_length)
    return raw_chars[-1]

def chatbot(net, sess, chars, vocab, max_length, beam_width, relevance, temperature, topn):
    states = initial_state_with_relevance_masking(net, sess, relevance)
    while True:
        user_input = input('\n> ')
        user_command_entered, reset, states, relevance, temperature, topn, beam_width = process_user_command(
            user_input, states, relevance, temperature, topn, beam_width)
        if reset: states = initial_state_with_relevance_masking(net, sess, relevance)
        if not user_command_entered:
            states = forward_text(net, sess, states, relevance, vocab, sanitize_text(vocab, "> " + user_input + "\n>"))
            computer_response_generator = beam_search_generator(sess=sess, net=net,
                initial_state=copy.deepcopy(states), initial_sample=vocab[' '],
                early_term_token=vocab['\n'], beam_width=beam_width, forward_model_fn=forward_with_mask,
                forward_args={'relevance':relevance, 'mask_reset_token':vocab['\n'], 'forbidden_token':vocab['>'],
                              'temperature':temperature, 'topn':topn})
            out_chars = []
            for i, char_token in enumerate(computer_response_generator):
                out_chars.append(chars[char_token])
                print(possibly_escaped_char(out_chars), end='', flush=True)
                states = forward_text(net, sess, states, relevance, vocab, chars[char_token])
```

Figure A.3: Snapshot of the chat.py file



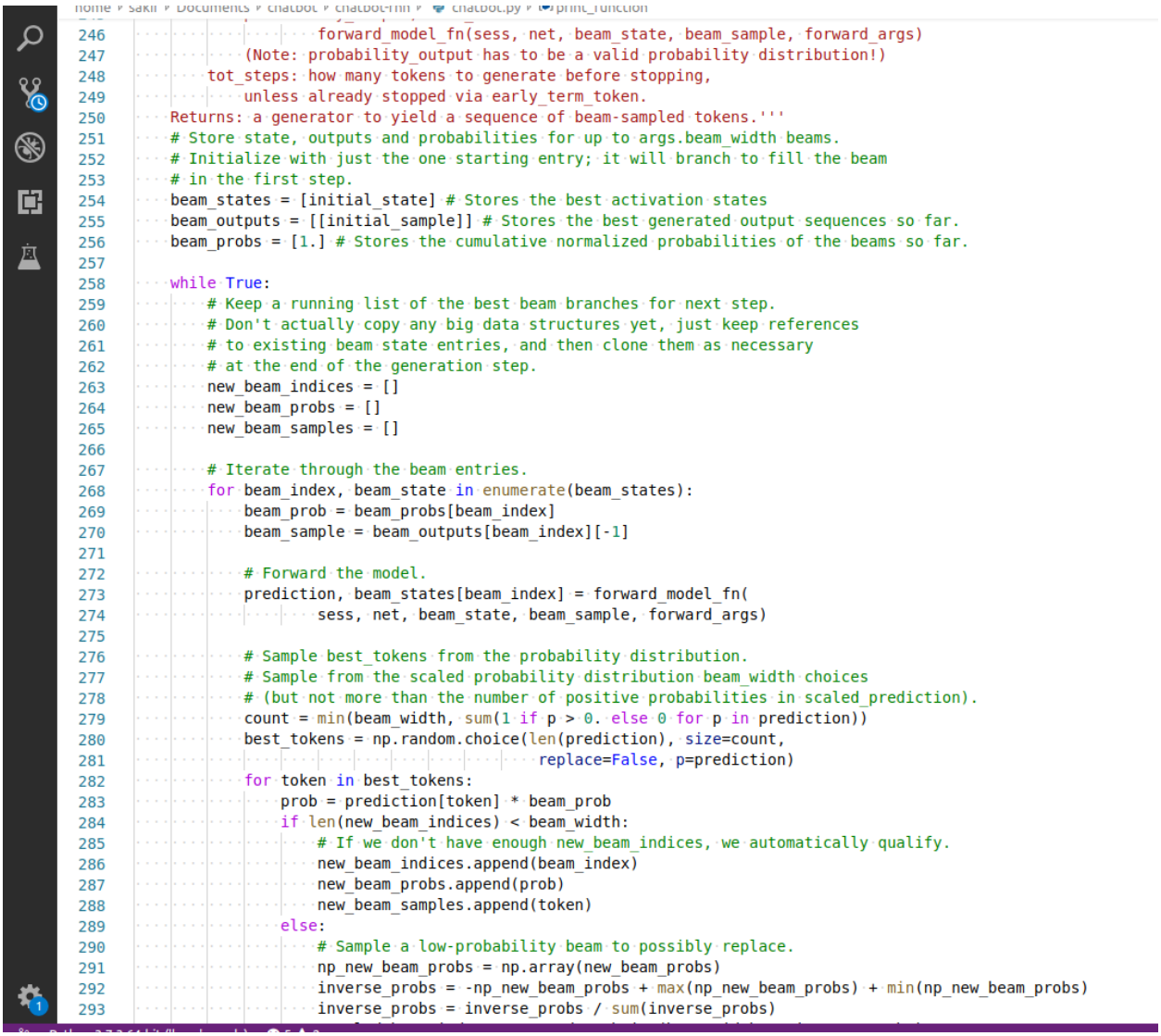
```
def process_user_command(user_input, states, relevance, temperature, topn, beam_width):
    user_command_entered = False
    reset = False
    try:
        if user_input.startswith('--temperature '):
            user_command_entered = True
            temperature = max(0.001, float(user_input[len('--temperature '):]))
            print("[Temperature set to {}]".format(temperature))
        elif user_input.startswith('--relevance '):
            user_command_entered = True
            new_relevance = float(user_input[len('--relevance '):])
            if relevance <= 0. and new_relevance > 0.:
                states = [states, copy.deepcopy(states)]
            elif relevance > 0. and new_relevance <= 0.:
                states = states[0]
            relevance = new_relevance
            print("[Relevance disabled]" if relevance <= 0. else "[Relevance set to {}]".format(relevance))
        elif user_input.startswith('--topn '):
            user_command_entered = True
            topn = int(user_input[len('--topn '):])
            print("[Top-n filtering disabled]" if topn <= 0 else "[Top-n filtering set to {}]".format(topn))
        elif user_input.startswith('--beam_width '):
            user_command_entered = True
            beam_width = max(1, int(user_input[len('--beam_width '):]))
            print("[Beam width set to {}]".format(beam_width))
        elif user_input.startswith('--reset'):
            user_command_entered = True
            reset = True
            print("[Model state reset]")
    except ValueError:
        print("[Value error with provided argument.]")
    return user_command_entered, reset, states, relevance, temperature, topn, beam_width

def consensus_length(beam_outputs, early_term_token):
    for l in range(len(beam_outputs[0])):
        if l > 0 and beam_outputs[0][l-1] == early_term_token:
            return l-1, True
        for b in beam_outputs[1:]:
            if beam_outputs[0][l] != b[l]: return l, False
    return l, False

def scale_prediction(prediction, temperature):
    if (temperature == 1.0): return prediction # Temperature 1.0 makes no change
    np.seterr(divide='ignore')
    scaled_prediction = np.log(prediction) / temperature
    scaled_prediction = scaled_prediction - np.logaddexp.reduce(scaled_prediction)
    scaled_prediction = np.exp(scaled_prediction)
```


Figure A.4: Snapshot of the chat.py file



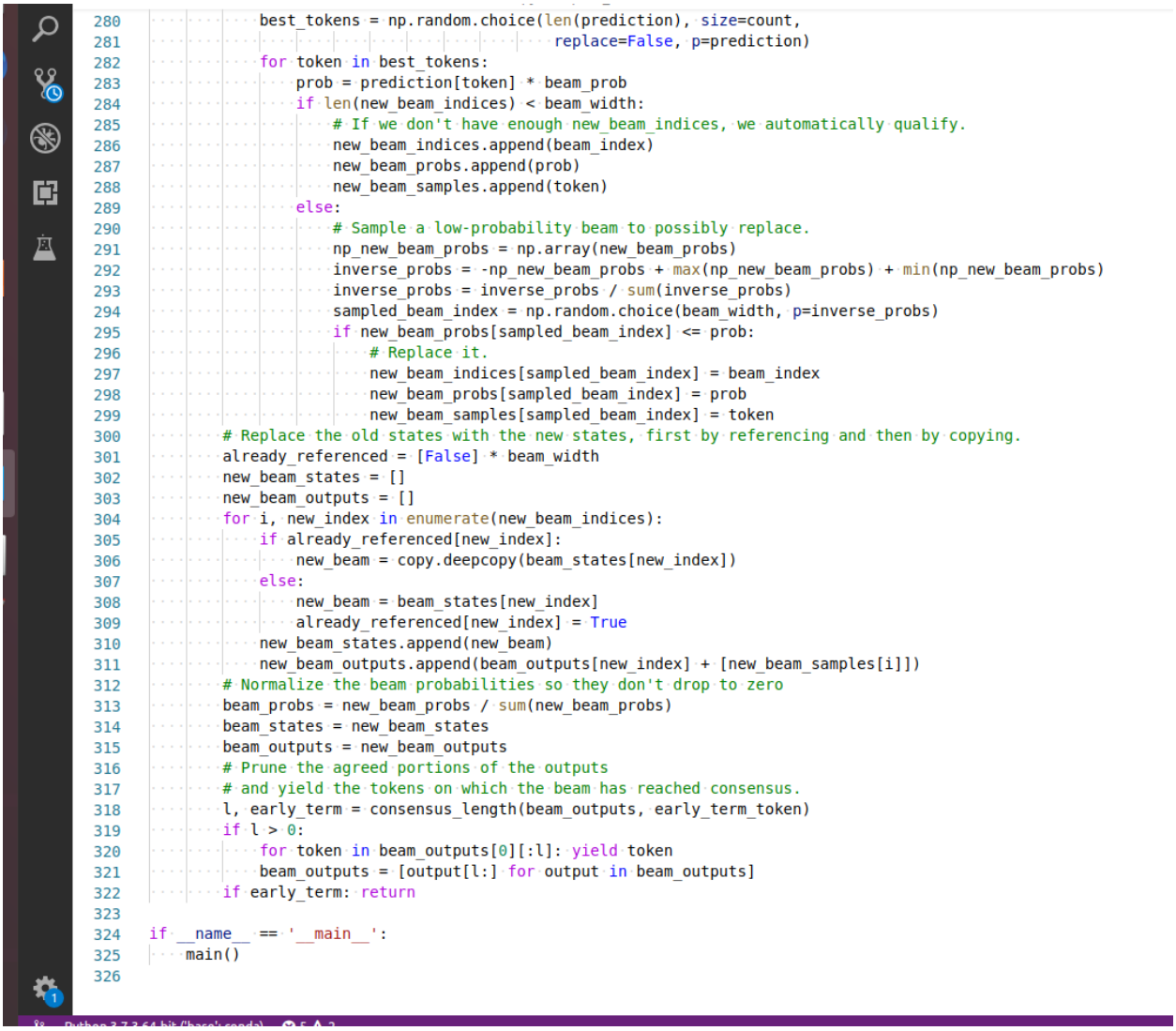
```
196
197 def forward_with_mask(sess, net, states, input_sample, forward_args):
198     # forward_args is a dictionary containing arguments for generating probabilities.
199     relevance = forward_args['relevance']
200     mask_reset_token = forward_args['mask_reset_token']
201     forbidden_token = forward_args['forbidden_token']
202     temperature = forward_args['temperature']
203     topn = forward_args['topn']
204
205     if relevance <= 0.:
206         # No relevance masking.
207         prob, states = net.forward_model(sess, states, input_sample)
208     else:
209         # states should be a 2-length list: [primary net state, mask net state].
210         if input_sample == mask_reset_token:
211             # Reset the mask probs when reaching mask_reset_token (newline).
212             states[1] = initial_state(net, sess)
213         primary_prob, states[0] = net.forward_model(sess, states[0], input_sample)
214         primary_prob /= sum(primary_prob)
215         mask_prob, states[1] = net.forward_model(sess, states[1], input_sample)
216         mask_prob /= sum(mask_prob)
217         prob = np.exp(np.log(primary_prob) - relevance * np.log(mask_prob))
218     # Mask out the forbidden token (">") to prevent the bot from deciding the chat is over)
219     prob[forbidden_token] = 0
220     # Normalize probabilities so they sum to 1.
221     prob = prob / sum(prob)
222     # Apply temperature.
223     prob = scale_prediction(prob, temperature)
224     # Apply top-n filtering if enabled
225     if topn > 0:
226         prob[np.argsort(prob)[:-topn]] = 0
227         prob = prob / sum(prob)
228     return prob, states
229
230 def beam_search_generator(sess, net, initial_state, initial_sample,
231     early_term_token, beam_width, forward_model_fn, forward_args):
232     '''Run beam search! Yield consensus tokens sequentially, as a generator;
233     return when reaching early_term_token (newline).
234
235     Args:
236         sess: tensorflow session reference
237         net: tensorflow net graph (must be compatible with the forward_net function)
238         initial_state: initial hidden state of the net
239         initial_sample: single token (excluding any seed/priming material)
240             to start the generation
241         early_term_token: stop when the beam reaches consensus on this token
242             (but do not return this token).
243         beam_width: how many beams to track
```

Figure A.5: Snapshot of the chat.py file



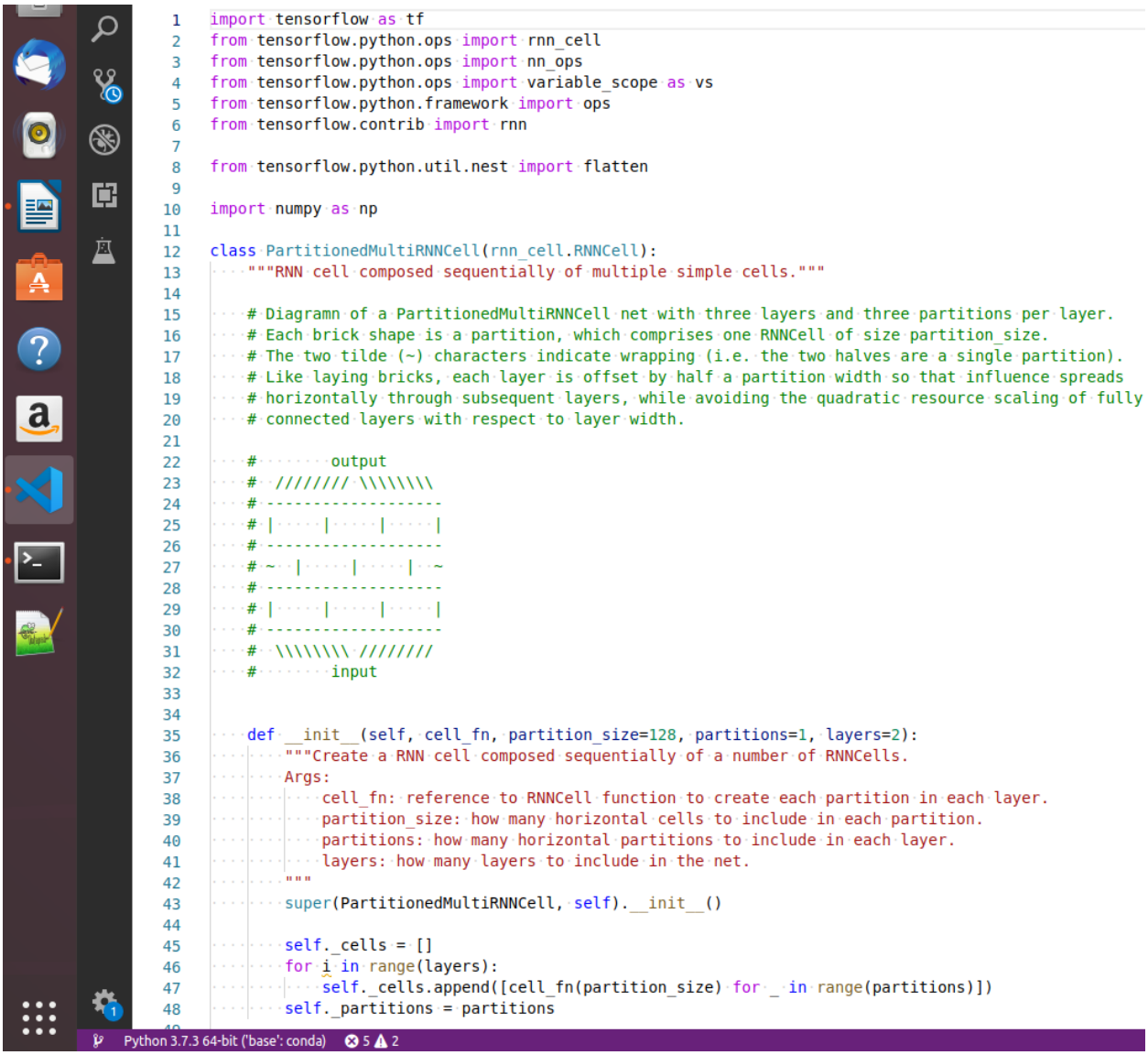
```
                    forward_model_fn(sess, net, beam_state, beam_sample, forward_args)
                (Note: probability_output has to be a valid probability distribution!)
            tot_steps: how many tokens to generate before stopping,
                unless already stopped via early_term_token.
    Returns: a generator to yield a sequence of beam-sampled tokens.'''
    # Store state, outputs and probabilities for up to args.beam_width beams.
    # Initialize with just the one starting entry; it will branch to fill the beam
    # in the first step.
    beam_states = [initial_state] # Stores the best activation states
    beam_outputs = [[initial_sample]] # Stores the best generated output sequences so far.
    beam_probs = [1.] # Stores the cumulative normalized probabilities of the beams so far.

    while True:
        # Keep a running list of the best beam branches for next step.
        # Don't actually copy any big data structures yet, just keep references
        # to existing beam state entries, and then clone them as necessary
        # at the end of the generation step.
        new_beam_indices = []
        new_beam_probs = []
        new_beam_samples = []

        # Iterate through the beam entries.
        for beam_index, beam_state in enumerate(beam_states):
            beam_prob = beam_probs[beam_index]
            beam_sample = beam_outputs[beam_index][-1]

            # Forward the model.
            prediction, beam_states[beam_index] = forward_model_fn(
                    sess, net, beam_state, beam_sample, forward_args)

            # Sample best_tokens from the probability distribution.
            # Sample from the scaled probability distribution beam_width choices
            # (but not more than the number of positive probabilities in scaled_prediction).
            count = min(beam_width, sum(1 if p > 0. else 0 for p in prediction))
            best_tokens = np.random.choice(len(prediction), size=count,
                                           replace=False, p=prediction)
            for token in best_tokens:
                prob = prediction[token] * beam_prob
                if len(new_beam_indices) < beam_width:
                    # If we don't have enough new_beam_indices, we automatically qualify.
                    new_beam_indices.append(beam_index)
                    new_beam_probs.append(prob)
                    new_beam_samples.append(token)
                else:
                    # Sample a low-probability beam to possibly replace.
                    np_new_beam_probs = np.array(new_beam_probs)
                    inverse_probs = -np_new_beam_probs + max(np_new_beam_probs) + min(np_new_beam_probs)
                    inverse_probs = inverse_probs / sum(inverse_probs)
```

Figure A.6: Snapshot of the chat.py file



```
                best_tokens = np.random.choice(len(prediction), size=count,
                                               replace=False, p=prediction)
                for token in best_tokens:
                    prob = prediction[token] * beam_prob
                    if len(new_beam_indices) < beam_width:
                        # If we don't have enough new_beam_indices, we automatically qualify.
                        new_beam_indices.append(beam_index)
                        new_beam_probs.append(prob)
                        new_beam_samples.append(token)
                    else:
                        # Sample a low-probability beam to possibly replace.
                        np_new_beam_probs = np.array(new_beam_probs)
                        inverse_probs = -np_new_beam_probs + max(np_new_beam_probs) + min(np_new_beam_probs)
                        inverse_probs = inverse_probs / sum(inverse_probs)
                        sampled_beam_index = np.random.choice(beam_width, p=inverse_probs)
                        if new_beam_probs[sampled_beam_index] <= prob:
                            # Replace it.
                            new_beam_indices[sampled_beam_index] = beam_index
                            new_beam_probs[sampled_beam_index] = prob
                            new_beam_samples[sampled_beam_index] = token
            # Replace the old states with the new states, first by referencing and then by copying.
            already_referenced = [False] * beam_width
            new_beam_states = []
            new_beam_outputs = []
            for i, new_index in enumerate(new_beam_indices):
                if already_referenced[new_index]:
                    new_beam = copy.deepcopy(beam_states[new_index])
                else:
                    new_beam = beam_states[new_index]
                    already_referenced[new_index] = True
                new_beam_states.append(new_beam)
                new_beam_outputs.append(beam_outputs[new_index] + [new_beam_samples[i]])
            # Normalize the beam probabilities so they don't drop to zero
            beam_probs = new_beam_probs / sum(new_beam_probs)
            beam_states = new_beam_states
            beam_outputs = new_beam_outputs
            # Prune the agreed portions of the outputs
            # and yield the tokens on which the beam has reached consensus.
            l, early_term = consensus_length(beam_outputs, early_term_token)
            if l > 0:
                for token in beam_outputs[0][:l]: yield token
                beam_outputs = [output[l:] for output in beam_outputs]
            if early_term: return

if __name__ == '__main__':
    main()
```

Figure A.7: Snapshot of the chat.py file



```python
import tensorflow as tf
from tensorflow.python.ops import rnn_cell
from tensorflow.python.ops import nn_ops
from tensorflow.python.ops import variable_scope as vs
from tensorflow.python.framework import ops
from tensorflow.contrib import rnn

from tensorflow.python.util.nest import flatten

import numpy as np

class PartitionedMultiRNNCell(rnn_cell.RNNCell):
    """RNN cell composed sequentially of multiple simple cells."""

    # Diagramn of a PartitionedMultiRNNCell net with three layers and three partitions per layer.
    # Each brick shape is a partition, which comprises one RNNCell of size partition_size.
    # The two tilde (~) characters indicate wrapping (i.e. the two halves are a single partition).
    # Like laying bricks, each layer is offset by half a partition width so that influence spreads
    # horizontally through subsequent layers, while avoiding the quadratic resource scaling of fully
    # connected layers with respect to layer width.

    #        output
    #  /////// \\\\\\\
    # -------------------
    # |     |     |     |
    # -------------------
    # ~  |     |     |  ~
    # -------------------
    # |     |     |     |
    # -------------------
    #  \\\\\\\ ///////
    #        input


    def __init__(self, cell_fn, partition_size=128, partitions=1, layers=2):
        """Create a RNN cell composed sequentially of a number of RNNCells.
        Args:
            cell_fn: reference to RNNCell function to create each partition in each layer.
            partition_size: how many horizontal cells to include in each partition.
            partitions: how many horizontal partitions to include in each layer.
            layers: how many layers to include in the net.
        """
        super(PartitionedMultiRNNCell, self).__init__()

        self._cells = []
        for i in range(layers):
            self._cells.append([cell_fn(partition_size) for _ in range(partitions)])
        self._partitions = partitions
```


Figure A.8: Snapshot of the model.py file

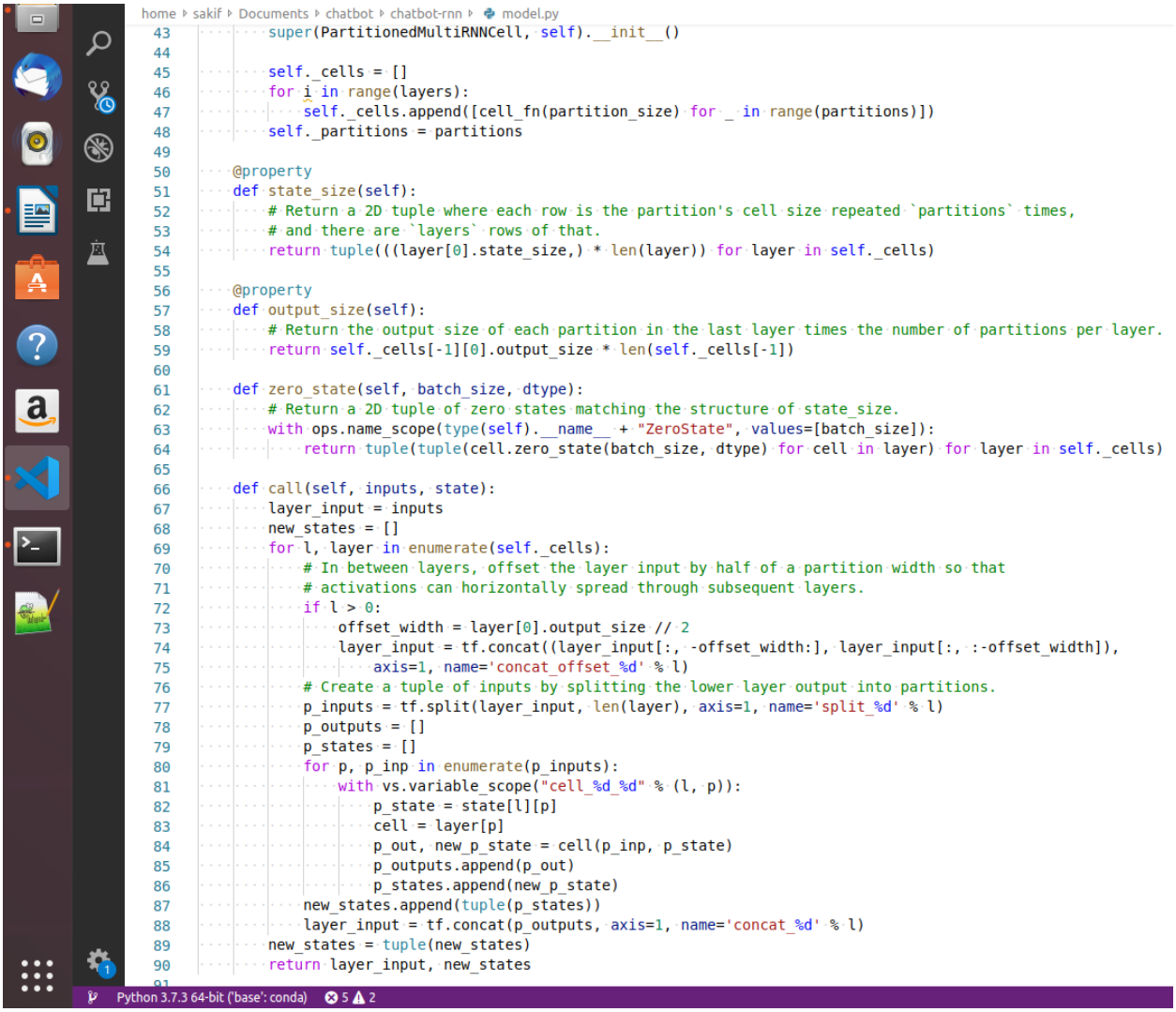
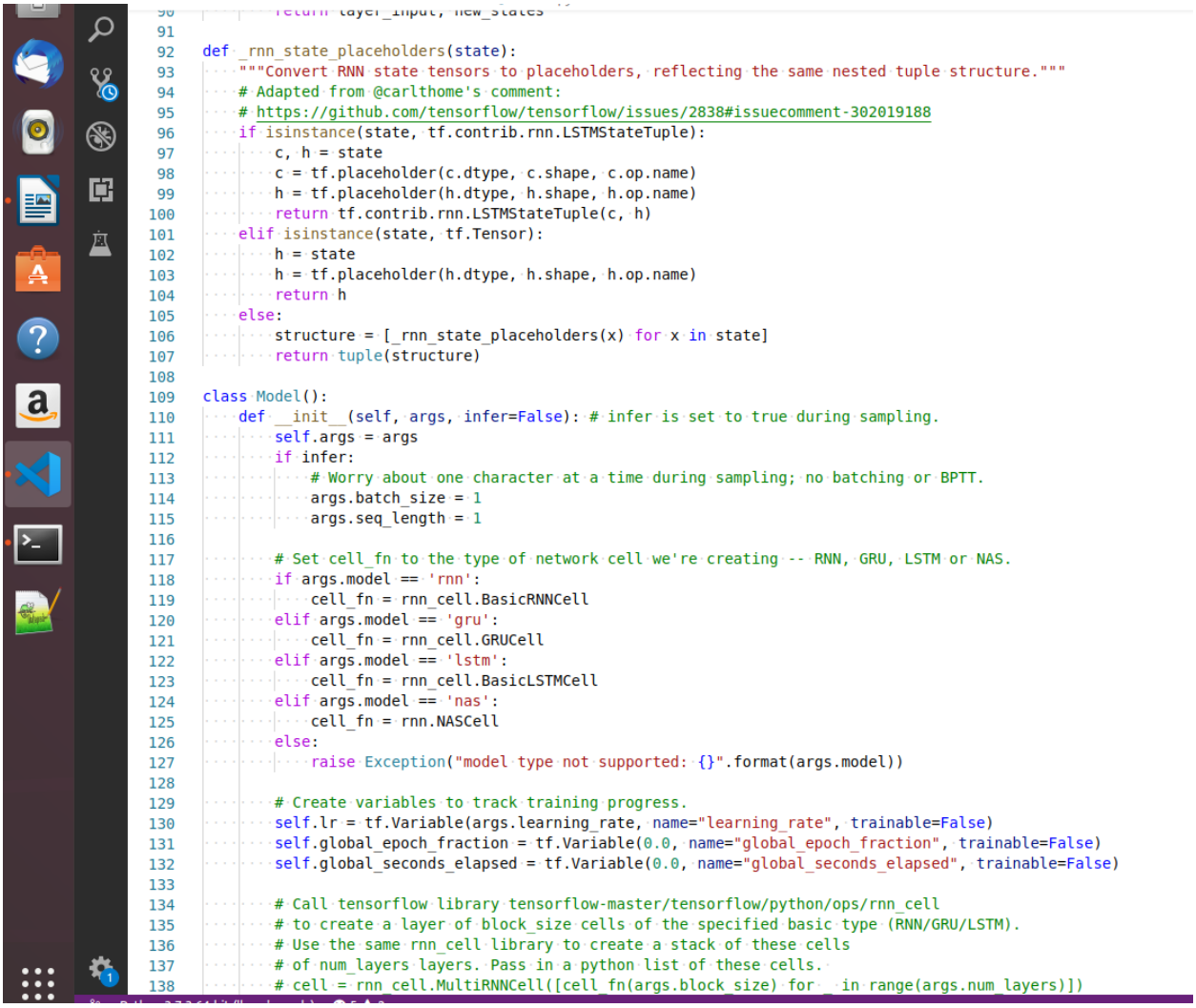
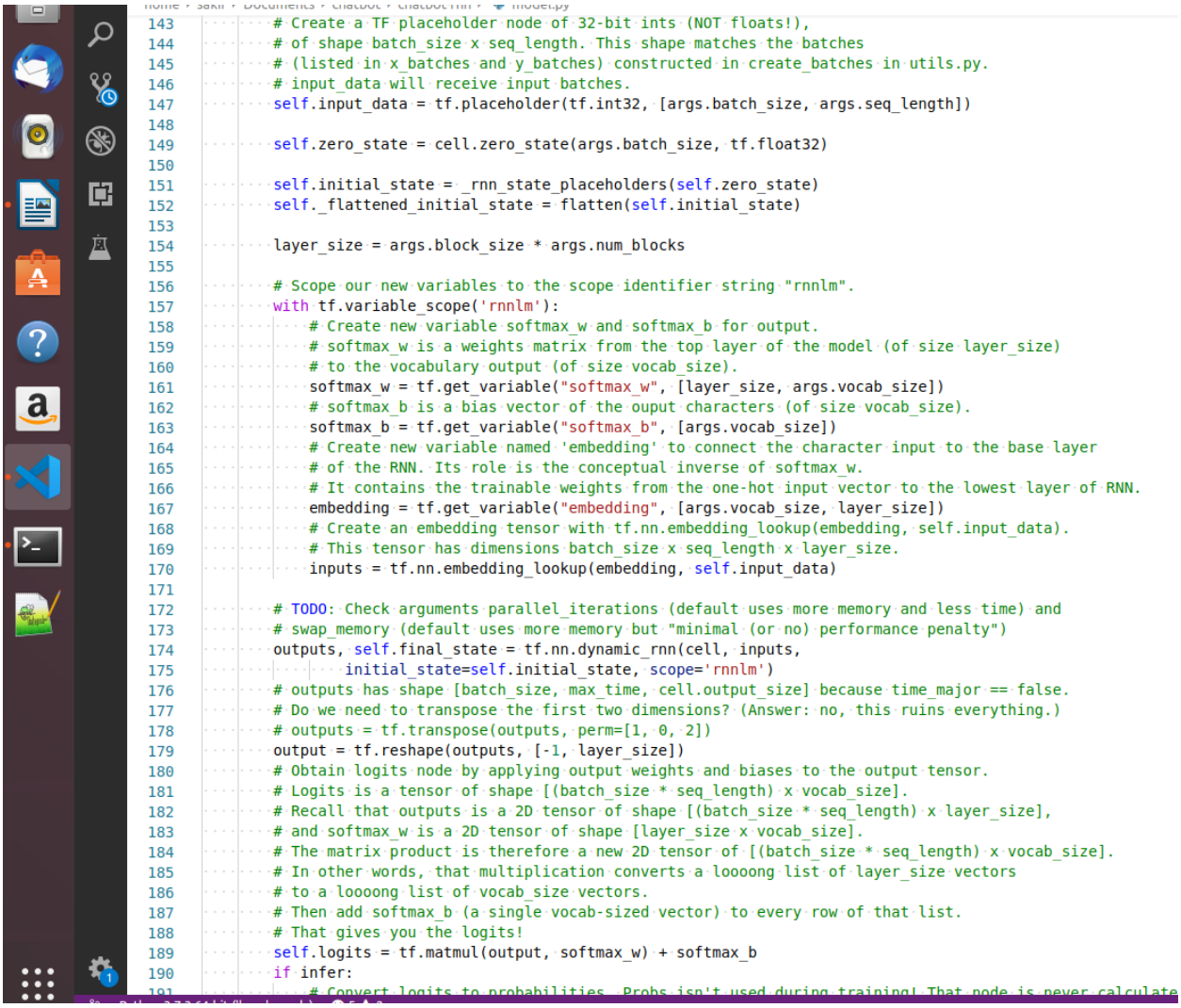


Figure A.9: Snapshot of the model.py file



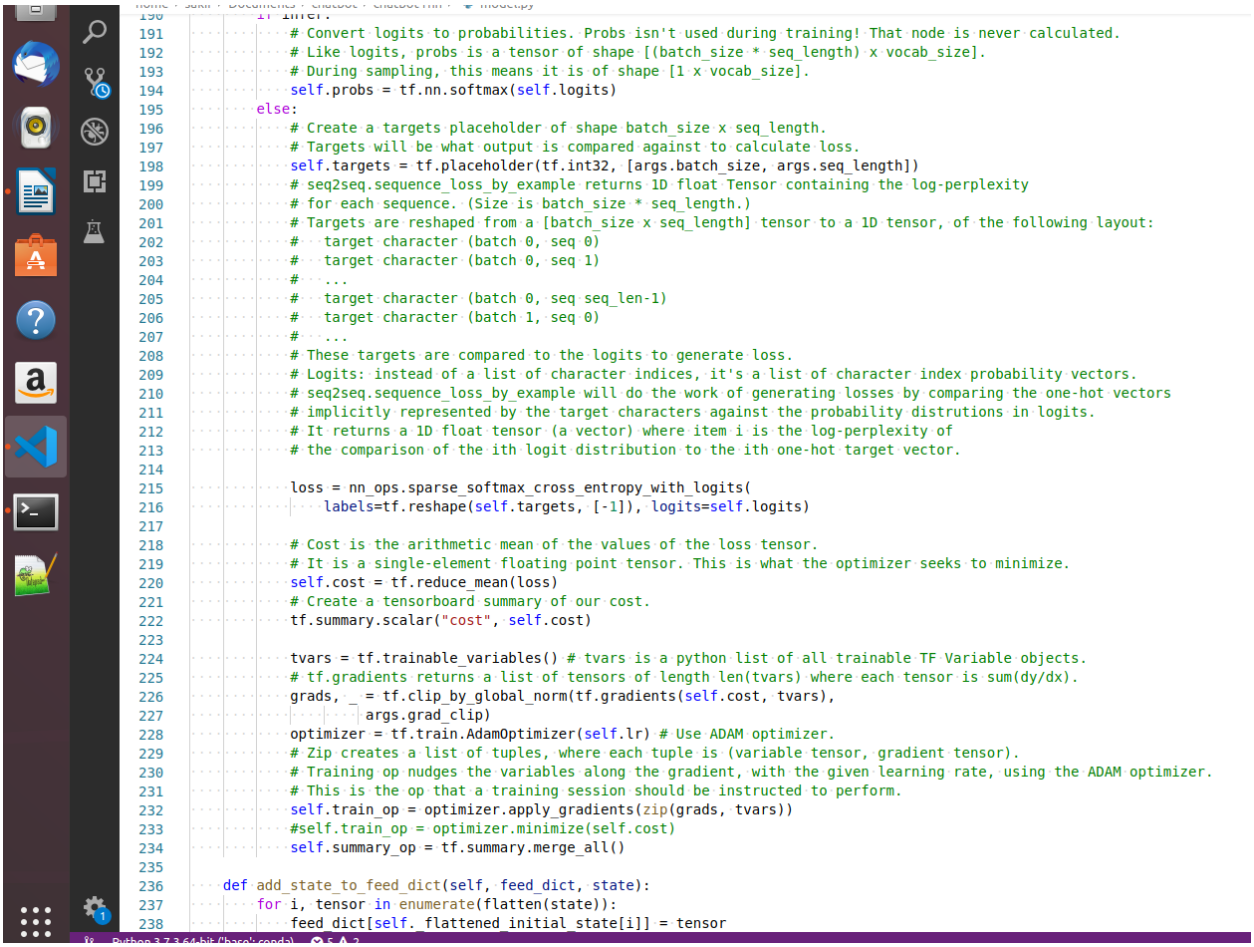
```
def _rnn_state_placeholders(state):
    """Convert RNN state tensors to placeholders, reflecting the same nested tuple structure."""
    # Adapted from @carlthome's comment:
    # https://github.com/tensorflow/tensorflow/issues/2838#issuecomment-302019188
    if isinstance(state, tf.contrib.rnn.LSTMStateTuple):
        c, h = state
        c = tf.placeholder(c.dtype, c.shape, c.op.name)
        h = tf.placeholder(h.dtype, h.shape, h.op.name)
        return tf.contrib.rnn.LSTMStateTuple(c, h)
    elif isinstance(state, tf.Tensor):
        h = state
        h = tf.placeholder(h.dtype, h.shape, h.op.name)
        return h
    else:
        structure = [_rnn_state_placeholders(x) for x in state]
        return tuple(structure)

class Model():
    def __init__(self, args, infer=False): # infer is set to true during sampling.
        self.args = args
        if infer:
            # Worry about one character at a time during sampling; no batching or BPTT.
            args.batch_size = 1
            args.seq_length = 1

        # Set cell_fn to the type of network cell we're creating -- RNN, GRU, LSTM or NAS.
        if args.model == 'rnn':
            cell_fn = rnn_cell.BasicRNNCell
        elif args.model == 'gru':
            cell_fn = rnn_cell.GRUCell
        elif args.model == 'lstm':
            cell_fn = rnn_cell.BasicLSTMCell
        elif args.model == 'nas':
            cell_fn = rnn.NASCell
        else:
            raise Exception("model type not supported: {}".format(args.model))

        # Create variables to track training progress.
        self.lr = tf.Variable(args.learning_rate, name="learning_rate", trainable=False)
        self.global_epoch_fraction = tf.Variable(0.0, name="global_epoch_fraction", trainable=False)
        self.global_seconds_elapsed = tf.Variable(0.0, name="global_seconds_elapsed", trainable=False)

        # Call tensorflow library tensorflow-master/tensorflow/python/ops/rnn_cell
        # to create a layer of block_size cells of the specified basic type (RNN/GRU/LSTM).
        # Use the same rnn_cell library to create a stack of these cells
        # of num_layers layers. Pass in a python list of these cells.
        # cell = rnn_cell.MultiRNNCell([cell_fn(args.block_size) for _ in range(args.num_layers)])
```

Figure A.10: Snapshot of the model.py file



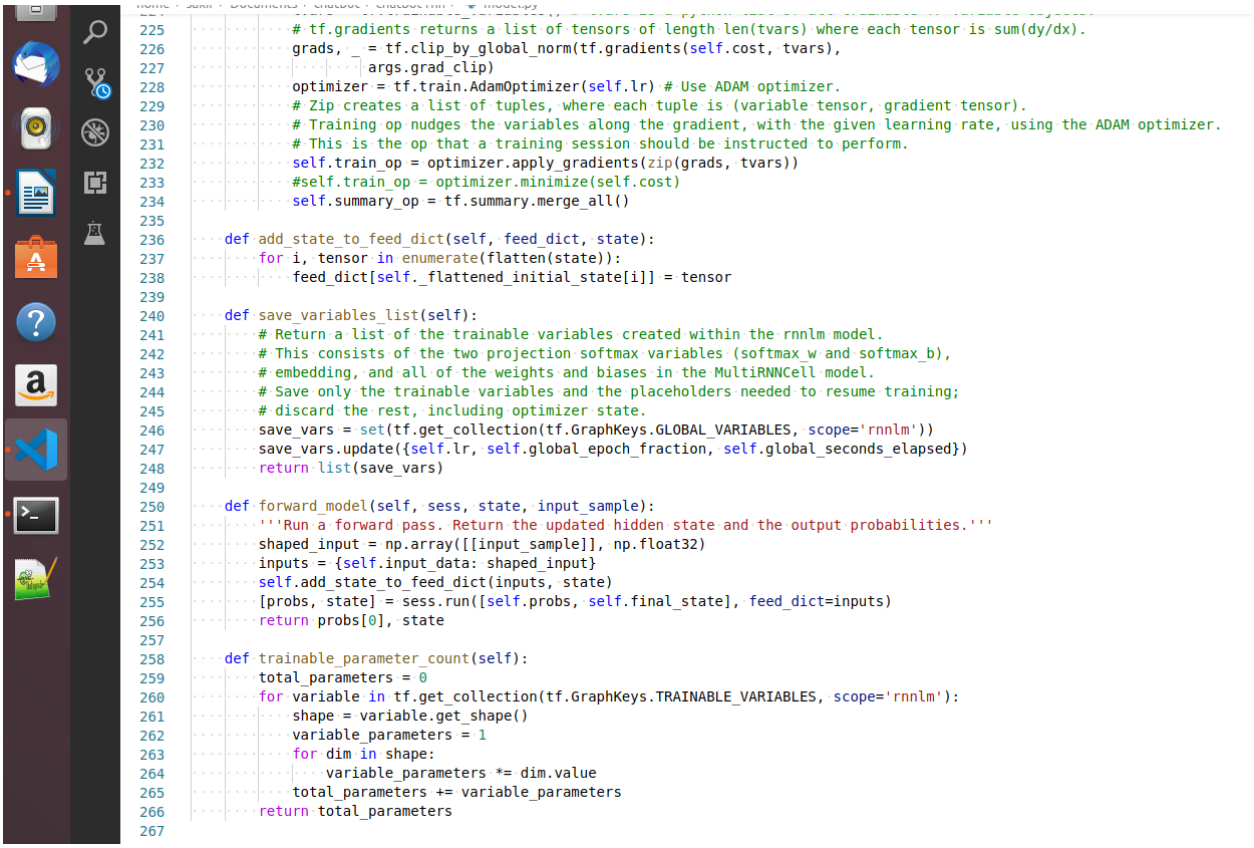
```
143         # Create a TF placeholder node of 32-bit ints (NOT floats!),
144         # of shape batch_size x seq_length. This shape matches the batches
145         # (listed in x_batches and y_batches) constructed in create_batches in utils.py.
146         # input_data will receive input batches.
147         self.input_data = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])
148
149         self.zero_state = cell.zero_state(args.batch_size, tf.float32)
150
151         self.initial_state = _rnn_state_placeholders(self.zero_state)
152         self._flattened_initial_state = flatten(self.initial_state)
153
154         layer_size = args.block_size * args.num_blocks
155
156         # Scope our new variables to the scope identifier string "rnnlm".
157         with tf.variable_scope('rnnlm'):
158             # Create new variable softmax_w and softmax_b for output.
159             # softmax_w is a weights matrix from the top layer of the model (of size layer_size)
160             # to the vocabulary output (of size vocab_size).
161             softmax_w = tf.get_variable("softmax_w", [layer_size, args.vocab_size])
162             # softmax_b is a bias vector of the ouput characters (of size vocab_size).
163             softmax_b = tf.get_variable("softmax_b", [args.vocab_size])
164             # Create new variable named 'embedding' to connect the character input to the base layer
165             # of the RNN. Its role is the conceptual inverse of softmax_w.
166             # It contains the trainable weights from the one-hot input vector to the lowest layer of RNN.
167             embedding = tf.get_variable("embedding", [args.vocab_size, layer_size])
168             # Create an embedding tensor with tf.nn.embedding_lookup(embedding, self.input_data).
169             # This tensor has dimensions batch_size x seq_length x layer_size.
170             inputs = tf.nn.embedding_lookup(embedding, self.input_data)
171
172         # TODO: Check arguments parallel_iterations (default uses more memory and less time) and
173         # swap_memory (default uses more memory but "minimal (or no) performance penalty")
174         outputs, self.final_state = tf.nn.dynamic_rnn(cell, inputs,
175                                 initial_state=self.initial_state, scope='rnnlm')
176         # outputs has shape [batch_size, max_time, cell.output_size] because time_major == false.
177         # Do we need to transpose the first two dimensions? (Answer: no, this ruins everything.)
178         # outputs = tf.transpose(outputs, perm=[1, 0, 2])
179         output = tf.reshape(outputs, [-1, layer_size])
180         # Obtain logits node by applying output weights and biases to the output tensor.
181         # Logits is a tensor of shape [(batch_size * seq_length) x vocab_size].
182         # Recall that outputs is a 2D tensor of shape [(batch_size * seq_length) x layer_size],
183         # and softmax_w is a 2D tensor of shape [layer_size x vocab_size].
184         # The matrix product is therefore a new 2D tensor of [(batch_size * seq_length) x vocab_size].
185         # In other words, that multiplication converts a loooong list of layer_size vectors
186         # to a loooong list of vocab_size vectors.
187         # Then add softmax_b (a single vocab-sized vector) to every row of that list.
188         # That gives you the logits!
189         self.logits = tf.matmul(output, softmax_w) + softmax_b
190         if infer:
191             # Convert logits to probabilities. Probs isn't used during training! That node is never calculate
```

Figure A.11: Snapshot of the model.py file



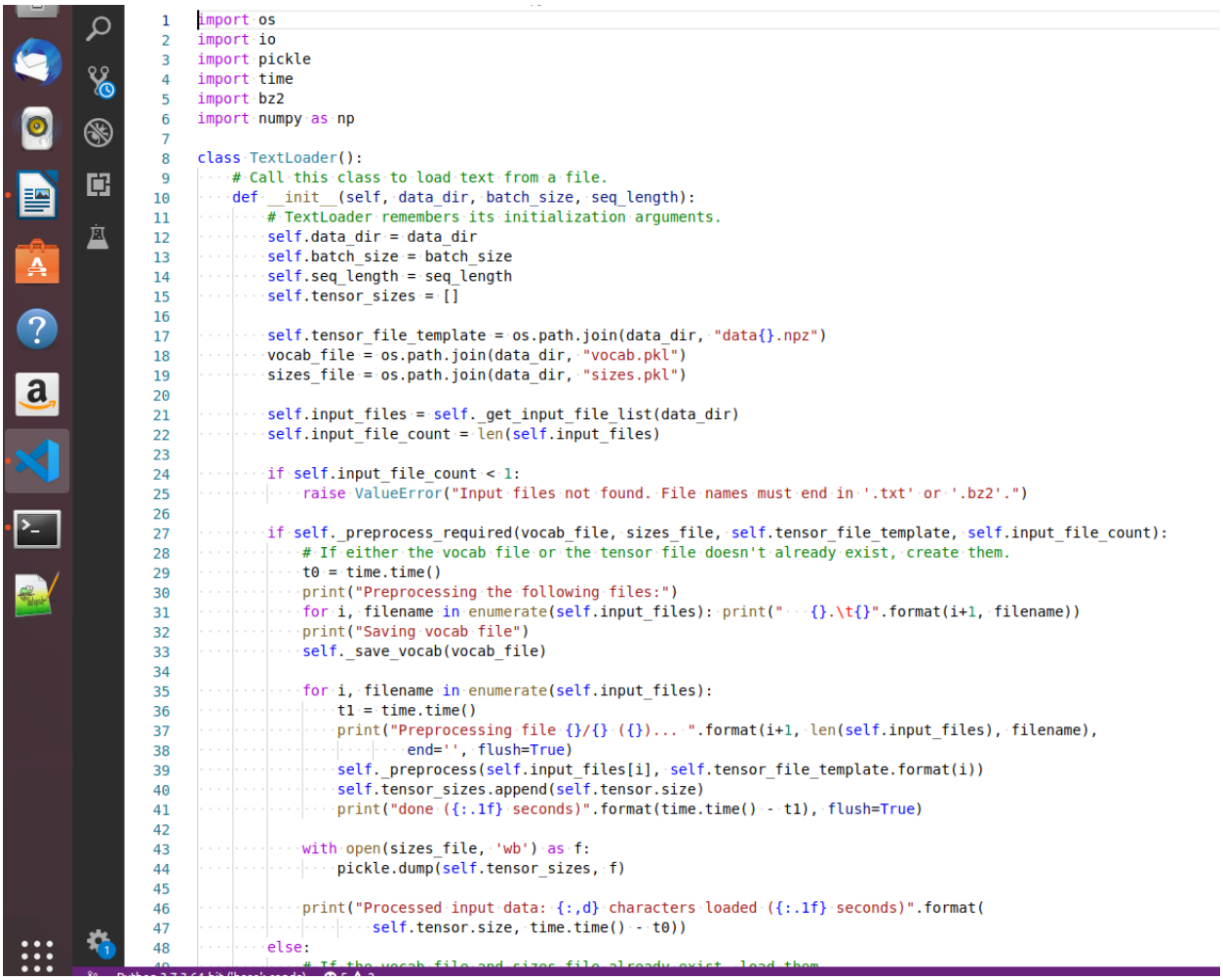
```
            if infer:
                # Convert logits to probabilities. Probs isn't used during training! That node is never calculated.
                # Like logits, probs is a tensor of shape [(batch_size * seq_length) x vocab_size].
                # During sampling, this means it is of shape [1 x vocab_size].
                self.probs = tf.nn.softmax(self.logits)
            else:
                # Create a targets placeholder of shape batch_size x seq_length.
                # Targets will be what output is compared against to calculate loss.
                self.targets = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])
                # seq2seq.sequence_loss_by_example returns 1D float Tensor containing the log-perplexity
                # for each sequence. (Size is batch_size * seq_length.)
                # Targets are reshaped from a [batch_size x seq_length] tensor to a 1D tensor, of the following layout:
                #   target character (batch 0, seq 0)
                #   target character (batch 0, seq 1)
                #   ...
                #   target character (batch 0, seq seq_len-1)
                #   target character (batch 1, seq 0)
                #   ...
                # These targets are compared to the logits to generate loss.
                # Logits: instead of a list of character indices, it's a list of character index probability vectors.
                # seq2seq.sequence_loss_by_example will do the work of generating losses by comparing the one-hot vectors
                # implicitly represented by the target characters against the probability distrutions in logits.
                # It returns a 1D float tensor (a vector) where item i is the log-perplexity of
                # the comparison of the ith logit distribution to the ith one-hot target vector.

                loss = nn_ops.sparse_softmax_cross_entropy_with_logits(
                    labels=tf.reshape(self.targets, [-1]), logits=self.logits)

                # Cost is the arithmetic mean of the values of the loss tensor.
                # It is a single-element floating point tensor. This is what the optimizer seeks to minimize.
                self.cost = tf.reduce_mean(loss)
                # Create a tensorboard summary of our cost.
                tf.summary.scalar("cost", self.cost)

                tvars = tf.trainable_variables() # tvars is a python list of all trainable TF Variable objects.
                # tf.gradients returns a list of tensors of length len(tvars) where each tensor is sum(dy/dx).
                grads, _ = tf.clip_by_global_norm(tf.gradients(self.cost, tvars),
                                                  args.grad_clip)
                optimizer = tf.train.AdamOptimizer(self.lr) # Use ADAM optimizer.
                # Zip creates a list of tuples, where each tuple is (variable tensor, gradient tensor).
                # Training op nudges the variables along the gradient, with the given learning rate, using the ADAM optimizer.
                # This is the op that a training session should be instructed to perform.
                self.train_op = optimizer.apply_gradients(zip(grads, tvars))
                #self.train_op = optimizer.minimize(self.cost)
                self.summary_op = tf.summary.merge_all()

    def add_state_to_feed_dict(self, feed_dict, state):
        for i, tensor in enumerate(flatten(state)):
            feed_dict[self._flattened_initial_state[i]] = tensor
```

Figure A.12: Snapshot of the model.py file



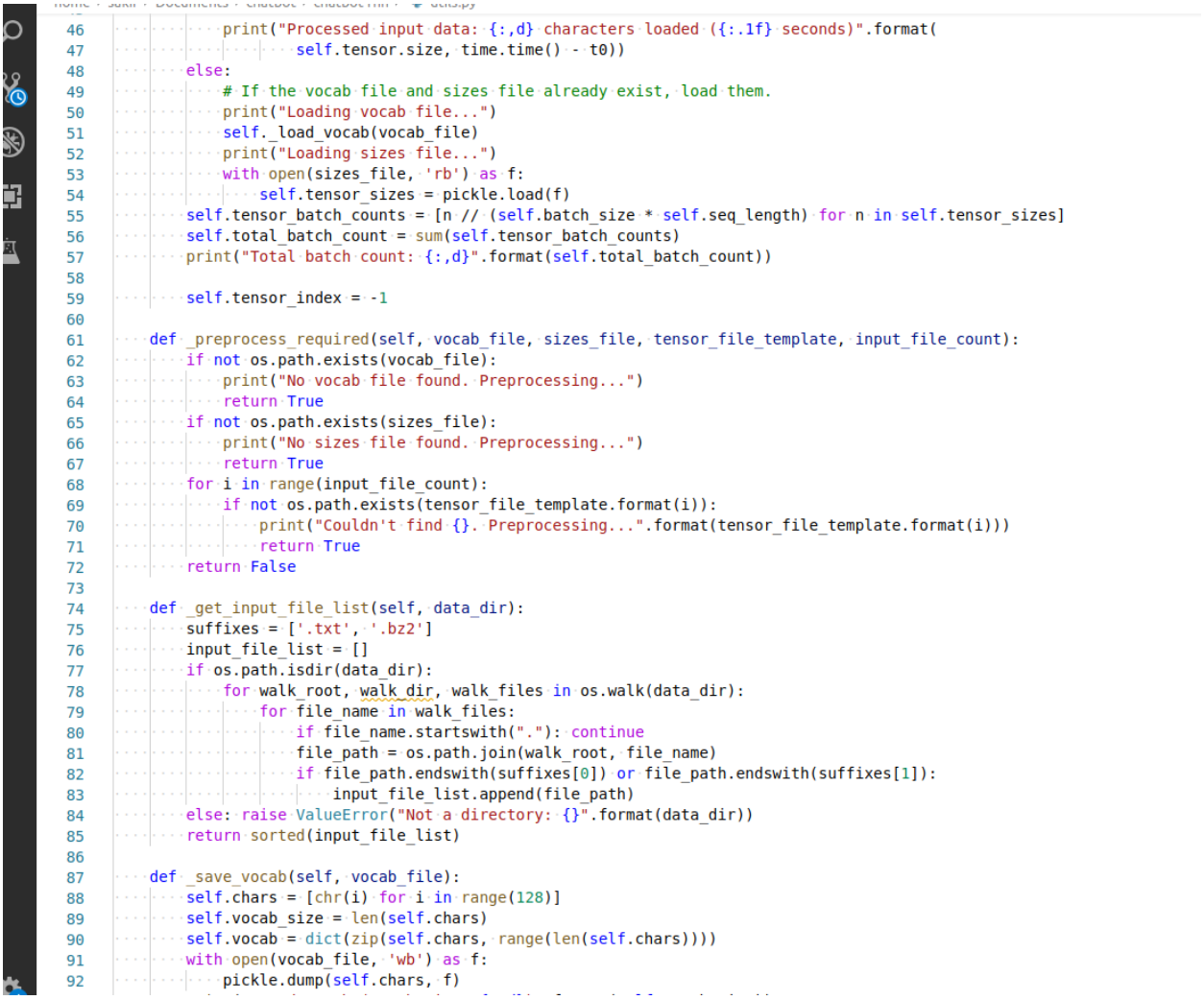
```
            # tf.gradients returns a list of tensors of length len(tvars) where each tensor is sum(dy/dx).
            grads, _ = tf.clip_by_global_norm(tf.gradients(self.cost, tvars),
                                              args.grad_clip)
            optimizer = tf.train.AdamOptimizer(self.lr) # Use ADAM optimizer.
            # Zip creates a list of tuples, where each tuple is (variable tensor, gradient tensor).
            # Training op nudges the variables along the gradient, with the given learning rate, using the ADAM optimizer.
            # This is the op that a training session should be instructed to perform.
            self.train_op = optimizer.apply_gradients(zip(grads, tvars))
            #self.train_op = optimizer.minimize(self.cost)
            self.summary_op = tf.summary.merge_all()

    def add_state_to_feed_dict(self, feed_dict, state):
        for i, tensor in enumerate(flatten(state)):
            feed_dict[self._flattened_initial_state[i]] = tensor

    def save_variables_list(self):
        # Return a list of the trainable variables created within the rnnlm model.
        # This consists of the two projection softmax variables (softmax_w and softmax_b),
        # embedding, and all of the weights and biases in the MultiRNNCell model.
        # Save only the trainable variables and the placeholders needed to resume training;
        # discard the rest, including optimizer state.
        save_vars = set(tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='rnnlm'))
        save_vars.update({self.lr, self.global_epoch_fraction, self.global_seconds_elapsed})
        return list(save_vars)

    def forward_model(self, sess, state, input_sample):
        '''Run a forward pass. Return the updated hidden state and the output probabilities.'''
        shaped_input = np.array([[input_sample]], np.float32)
        inputs = {self.input_data: shaped_input}
        self.add_state_to_feed_dict(inputs, state)
        [probs, state] = sess.run([self.probs, self.final_state], feed_dict=inputs)
        return probs[0], state

    def trainable_parameter_count(self):
        total_parameters = 0
        for variable in tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='rnnlm'):
            shape = variable.get_shape()
            variable_parameters = 1
            for dim in shape:
                variable_parameters *= dim.value
            total_parameters += variable_parameters
        return total_parameters

```

Figure A.13: Snapshot of the model.py file



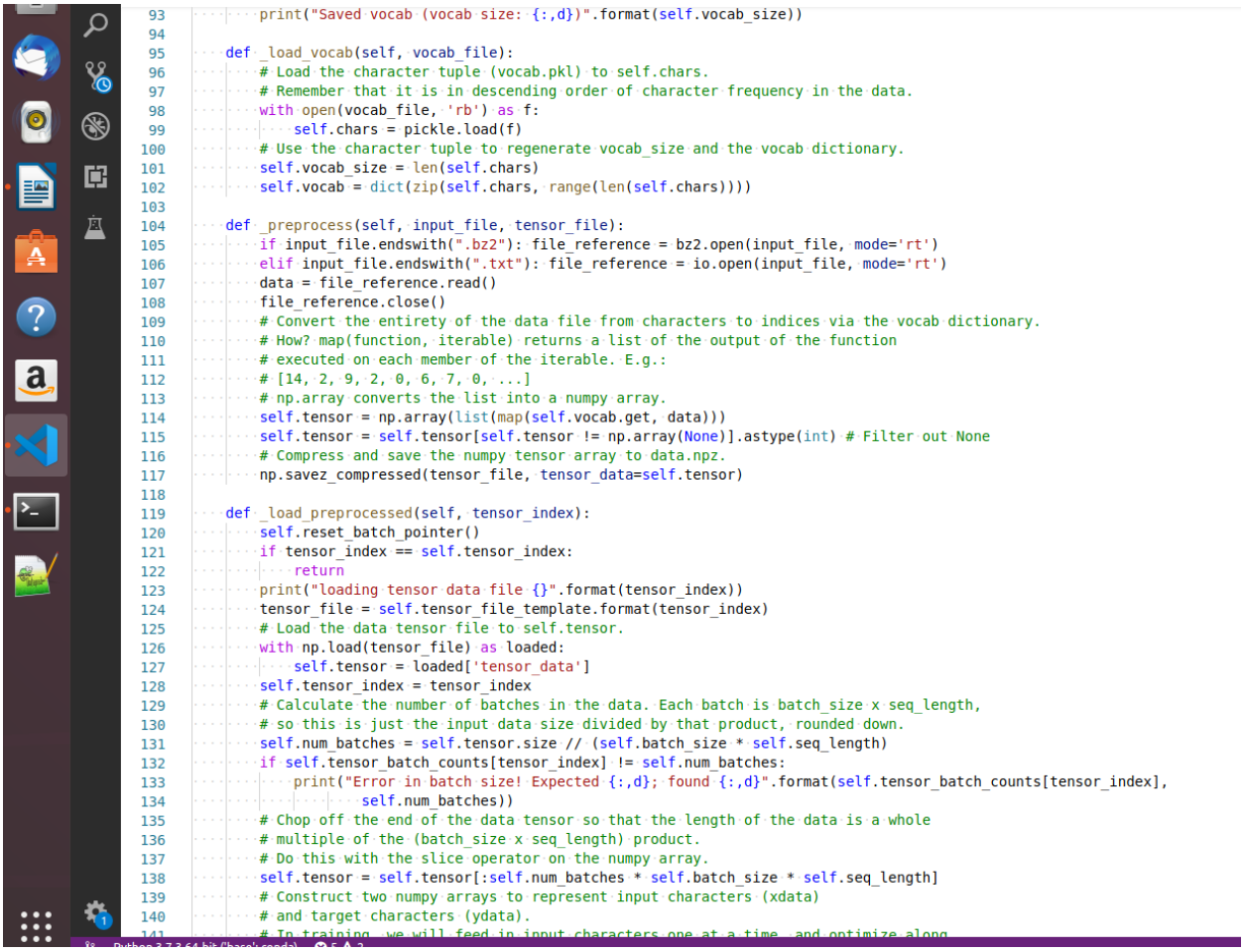
```
import os
import io
import pickle
import time
import bz2
import numpy as np

class TextLoader():
    # Call this class to load text from a file.
    def __init__(self, data_dir, batch_size, seq_length):
        # TextLoader remembers its initialization arguments.
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.seq_length = seq_length
        self.tensor_sizes = []

        self.tensor_file_template = os.path.join(data_dir, "data{}.npz")
        vocab_file = os.path.join(data_dir, "vocab.pkl")
        sizes_file = os.path.join(data_dir, "sizes.pkl")

        self.input_files = self._get_input_file_list(data_dir)
        self.input_file_count = len(self.input_files)

        if self.input_file_count < 1:
            raise ValueError("Input files not found. File names must end in '.txt' or '.bz2'.")

        if self._preprocess_required(vocab_file, sizes_file, self.tensor_file_template, self.input_file_count):
            # If either the vocab file or the tensor file doesn't already exist, create them.
            t0 = time.time()
            print("Preprocessing the following files:")
            for i, filename in enumerate(self.input_files): print("    {}.\t{}".format(i+1, filename))
            print("Saving vocab file")
            self._save_vocab(vocab_file)

            for i, filename in enumerate(self.input_files):
                t1 = time.time()
                print("Preprocessing file {}/{} ({})... ".format(i+1, len(self.input_files), filename),
                      end='', flush=True)
                self._preprocess(self.input_files[i], self.tensor_file_template.format(i))
                self.tensor_sizes.append(self.tensor.size)
                print("done ({:.1f} seconds)".format(time.time() - t1), flush=True)

            with open(sizes_file, 'wb') as f:
                pickle.dump(self.tensor_sizes, f)

            print("Processed input data: {:,d} characters loaded ({:.1f} seconds)".format(
                    self.tensor.size, time.time() - t0))
        else:
            # If the vocab file and sizes file already exist, load them
```

Figure A.14: Snapshot of the utils.py file



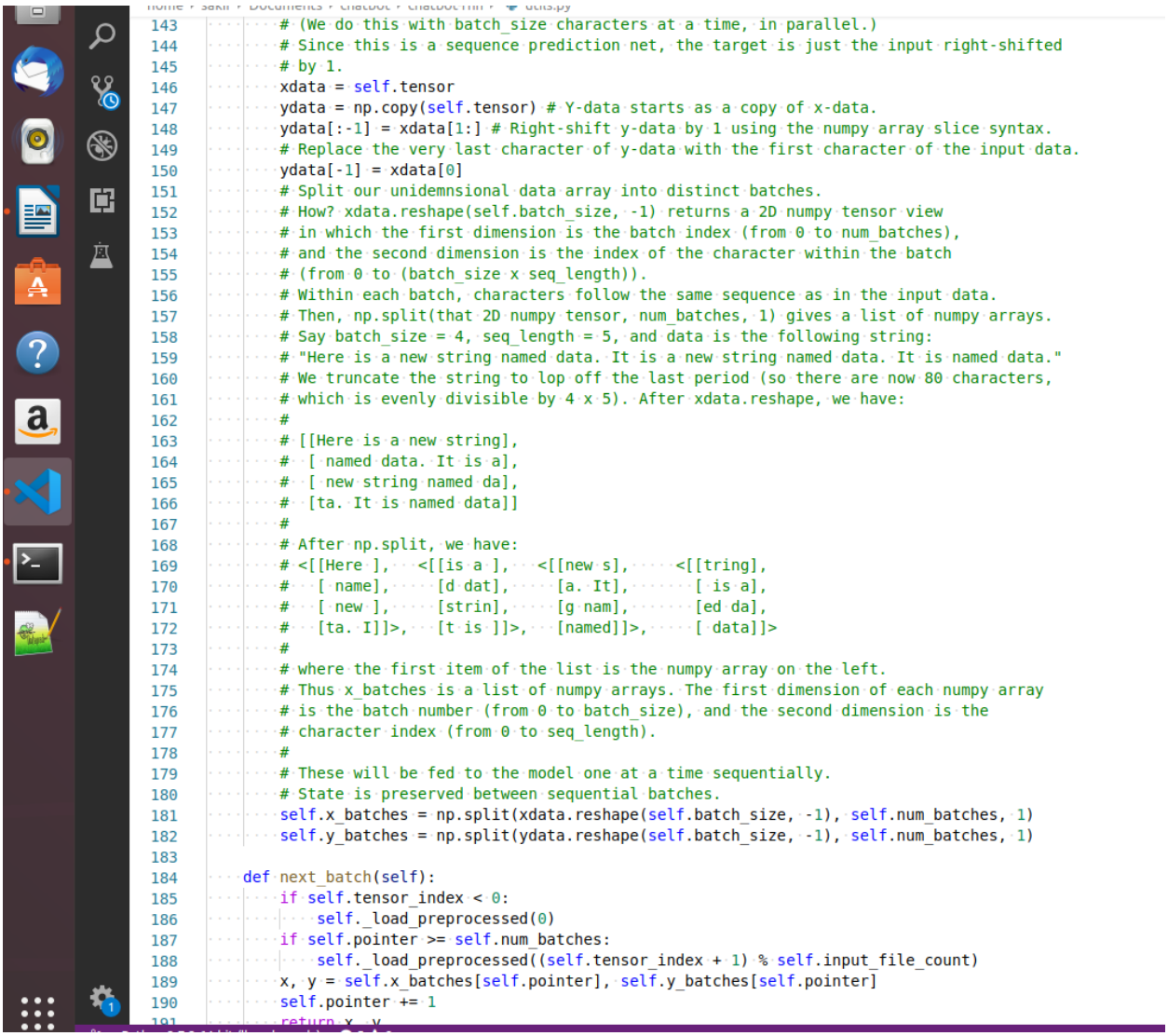
```
            print("Processed input data: {:,d} characters loaded ({:.1f} seconds)".format(
                    self.tensor.size, time.time() - t0))
        else:
            # If the vocab file and sizes file already exist, load them.
            print("Loading vocab file...")
            self._load_vocab(vocab_file)
            print("Loading sizes file...")
            with open(sizes_file, 'rb') as f:
                self.tensor_sizes = pickle.load(f)
        self.tensor_batch_counts = [n // (self.batch_size * self.seq_length) for n in self.tensor_sizes]
        self.total_batch_count = sum(self.tensor_batch_counts)
        print("Total batch count: {:,d}".format(self.total_batch_count))

        self.tensor_index = -1

    def _preprocess_required(self, vocab_file, sizes_file, tensor_file_template, input_file_count):
        if not os.path.exists(vocab_file):
            print("No vocab file found. Preprocessing...")
            return True
        if not os.path.exists(sizes_file):
            print("No sizes file found. Preprocessing...")
            return True
        for i in range(input_file_count):
            if not os.path.exists(tensor_file_template.format(i)):
                print("Couldn't find {}. Preprocessing...".format(tensor_file_template.format(i)))
                return True
        return False

    def _get_input_file_list(self, data_dir):
        suffixes = ['.txt', '.bz2']
        input_file_list = []
        if os.path.isdir(data_dir):
            for walk_root, walk_dir, walk_files in os.walk(data_dir):
                for file_name in walk_files:
                    if file_name.startswith("."): continue
                    file_path = os.path.join(walk_root, file_name)
                    if file_path.endswith(suffixes[0]) or file_path.endswith(suffixes[1]):
                        input_file_list.append(file_path)
        else: raise ValueError("Not a directory: {}".format(data_dir))
        return sorted(input_file_list)

    def _save_vocab(self, vocab_file):
        self.chars = [chr(i) for i in range(128)]
        self.vocab_size = len(self.chars)
        self.vocab = dict(zip(self.chars, range(len(self.chars))))
        with open(vocab_file, 'wb') as f:
            pickle.dump(self.chars, f)
```

Figure A.15: Snapshot of the utils.py file



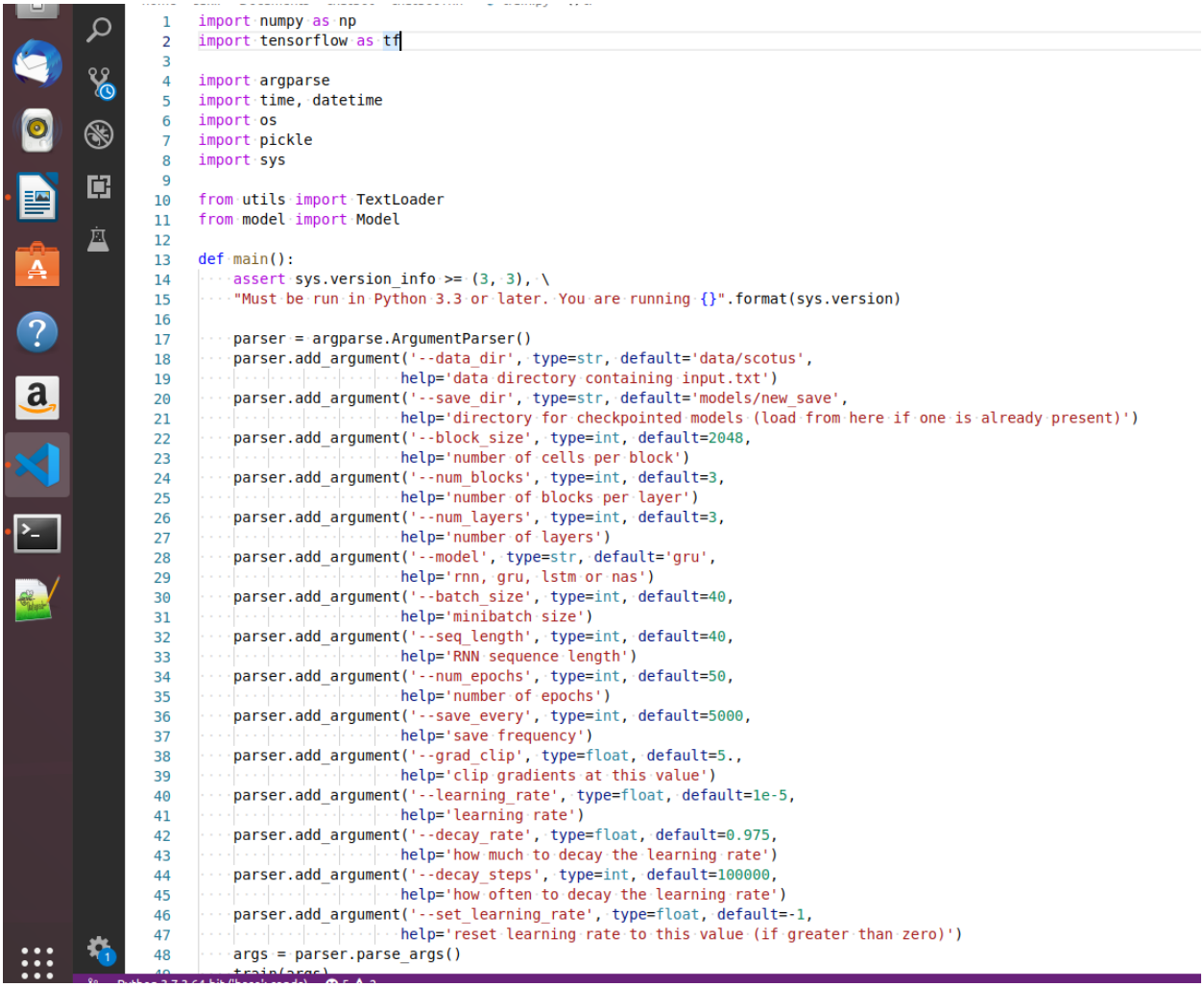
```
        print("Saved vocab (vocab size: {:,d})".format(self.vocab_size))

    def load_vocab(self, vocab_file):
        # Load the character tuple (vocab.pkl) to self.chars.
        # Remember that it is in descending order of character frequency in the data.
        with open(vocab_file, 'rb') as f:
            self.chars = pickle.load(f)
        # Use the character tuple to regenerate vocab_size and the vocab dictionary.
        self.vocab_size = len(self.chars)
        self.vocab = dict(zip(self.chars, range(len(self.chars))))

    def preprocess(self, input_file, tensor_file):
        if input_file.endswith(".bz2"): file_reference = bz2.open(input_file, mode='rt')
        elif input_file.endswith(".txt"): file_reference = io.open(input_file, mode='rt')
        data = file_reference.read()
        file_reference.close()
        # Convert the entirety of the data file from characters to indices via the vocab dictionary.
        # How? map(function, iterable) returns a list of the output of the function
        # executed on each member of the iterable. E.g.:
        # [14, 2, 9, 2, 0, 6, 7, 0, ...]
        # np.array converts the list into a numpy array.
        self.tensor = np.array(list(map(self.vocab.get, data)))
        self.tensor = self.tensor[self.tensor != np.array(None)].astype(int) # Filter out None
        # Compress and save the numpy tensor array to data.npz.
        np.savez_compressed(tensor_file, tensor_data=self.tensor)

    def load_preprocessed(self, tensor_index):
        self.reset_batch_pointer()
        if tensor_index == self.tensor_index:
            return
        print("loading tensor data file {}".format(tensor_index))
        tensor_file = self.tensor_file_template.format(tensor_index)
        # Load the data tensor file to self.tensor.
        with np.load(tensor_file) as loaded:
            self.tensor = loaded['tensor_data']
        self.tensor_index = tensor_index
        # Calculate the number of batches in the data. Each batch is batch_size x seq_length,
        # so this is just the input data size divided by that product, rounded down.
        self.num_batches = self.tensor.size // (self.batch_size * self.seq_length)
        if self.tensor_batch_counts[tensor_index] != self.num_batches:
            print("Error in batch size! Expected {:,d}; found {:,d}".format(self.tensor_batch_counts[tensor_index],
                  self.num_batches))
        # Chop off the end of the data tensor so that the length of the data is a whole
        # multiple of the (batch_size x seq_length) product.
        # Do this with the slice operator on the numpy array.
        self.tensor = self.tensor[:self.num_batches * self.batch_size * self.seq_length]
        # Construct two numpy arrays to represent input characters (xdata)
        # and target characters (ydata).
        # In training, we will feed in input characters one at a time, and optimize along
```

Figure A.16: Snapshot of the utils.py file



```
143         # (We do this with batch_size characters at a time, in parallel.)
144         # Since this is a sequence prediction net, the target is just the input right-shifted
145         # by 1.
146         xdata = self.tensor
147         ydata = np.copy(self.tensor) # Y-data starts as a copy of x-data.
148         ydata[:-1] = xdata[1:] # Right-shift y-data by 1 using the numpy array slice syntax.
149         # Replace the very last character of y-data with the first character of the input data.
150         ydata[-1] = xdata[0]
151         # Split our unidemnsional data array into distinct batches.
152         # How? xdata.reshape(self.batch_size, -1) returns a 2D numpy tensor view
153         # in which the first dimension is the batch index (from 0 to num_batches),
154         # and the second dimension is the index of the character within the batch
155         # (from 0 to (batch_size x seq_length)).
156         # Within each batch, characters follow the same sequence as in the input data.
157         # Then, np.split(that 2D numpy tensor, num_batches, 1) gives a list of numpy arrays.
158         # Say batch_size = 4, seq_length = 5, and data is the following string:
159         # "Here is a new string named data. It is a new string named data. It is named data."
160         # We truncate the string to lop off the last period (so there are now 80 characters,
161         # which is evenly divisible by 4 x 5). After xdata.reshape, we have:
162         #
163         # [[Here is a new string],
164         #  [ named data. It is a],
165         #  [ new string named da],
166         #  [ta. It is named data]]
167         #
168         # After np.split, we have:
169         # <[[Here ],   <[[is a ],   <[[new s],    <[[tring],
170         #  [ name],    [d dat],     [a. It],      [ is a],
171         #  [ new ],    [strin],     [g nam],      [ed da],
172         #  [ta. I]]>,  [t is ]]>,   [named]]>,    [ data]]>
173         #
174         # where the first item of the list is the numpy array on the left.
175         # Thus x_batches is a list of numpy arrays. The first dimension of each numpy array
176         # is the batch number (from 0 to batch_size), and the second dimension is the
177         # character index (from 0 to seq_length).
178         #
179         # These will be fed to the model one at a time sequentially.
180         # State is preserved between sequential batches.
181         self.x_batches = np.split(xdata.reshape(self.batch_size, -1), self.num_batches, 1)
182         self.y_batches = np.split(ydata.reshape(self.batch_size, -1), self.num_batches, 1)
183
184     def next_batch(self):
185         if self.tensor_index < 0:
186             self._load_preprocessed(0)
187         if self.pointer >= self.num_batches:
188             self._load_preprocessed((self.tensor_index + 1) % self.input_file_count)
189         x, y = self.x_batches[self.pointer], self.y_batches[self.pointer]
190         self.pointer += 1
191         return x, y
```

Figure A.17: Snapshot of the utils.py file



```python
import numpy as np
import tensorflow as tf

import argparse
import time, datetime
import os
import pickle
import sys

from utils import TextLoader
from model import Model

def main():
    assert sys.version_info >= (3, 3), \
    "Must be run in Python 3.3 or later. You are running {}".format(sys.version)

    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str, default='data/scotus',
                        help='data directory containing input.txt')
    parser.add_argument('--save_dir', type=str, default='models/new_save',
                        help='directory for checkpointed models (load from here if one is already present)')
    parser.add_argument('--block_size', type=int, default=2048,
                        help='number of cells per block')
    parser.add_argument('--num_blocks', type=int, default=3,
                        help='number of blocks per layer')
    parser.add_argument('--num_layers', type=int, default=3,
                        help='number of layers')
    parser.add_argument('--model', type=str, default='gru',
                        help='rnn, gru, lstm or nas')
    parser.add_argument('--batch_size', type=int, default=40,
                        help='minibatch size')
    parser.add_argument('--seq_length', type=int, default=40,
                        help='RNN sequence length')
    parser.add_argument('--num_epochs', type=int, default=50,
                        help='number of epochs')
    parser.add_argument('--save_every', type=int, default=5000,
                        help='save frequency')
    parser.add_argument('--grad_clip', type=float, default=5.,
                        help='clip gradients at this value')
    parser.add_argument('--learning_rate', type=float, default=1e-5,
                        help='learning rate')
    parser.add_argument('--decay_rate', type=float, default=0.975,
                        help='how much to decay the learning rate')
    parser.add_argument('--decay_steps', type=int, default=100000,
                        help='how often to decay the learning rate')
    parser.add_argument('--set_learning_rate', type=float, default=-1,
                        help='reset learning rate to this value (if greater than zero)')
    args = parser.parse_args()
    train(args)
```

Figure A.18: Snapshot of the train.py file

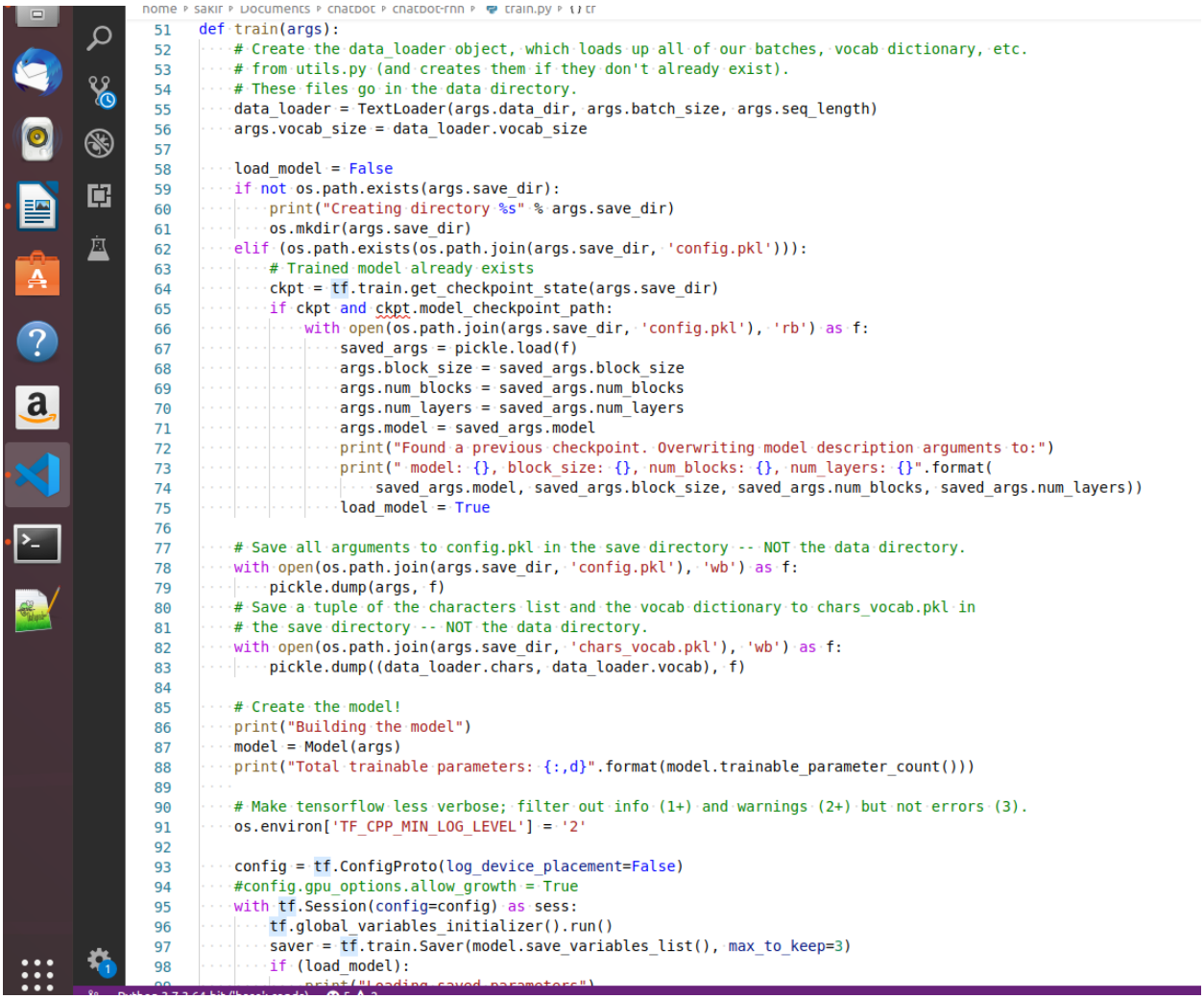
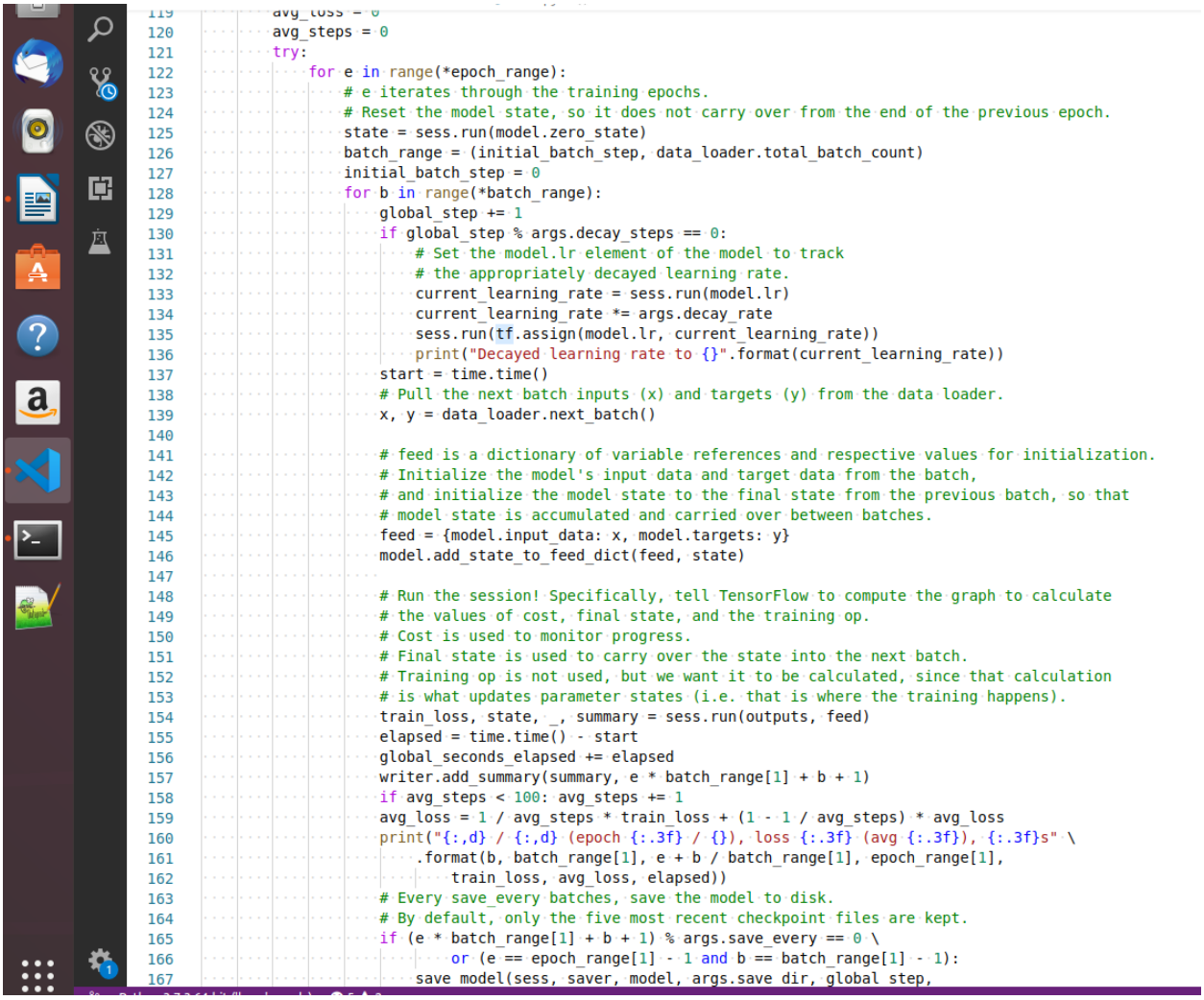


Figure A.19: Snapshot of the train.py file



```
avg_loss = 0
avg_steps = 0
try:
    for e in range(*epoch_range):
        # e iterates through the training epochs.
        # Reset the model state, so it does not carry over from the end of the previous epoch.
        state = sess.run(model.zero_state)
        batch_range = (initial_batch_step, data_loader.total_batch_count)
        initial_batch_step = 0
        for b in range(*batch_range):
            global_step += 1
            if global_step % args.decay_steps == 0:
                # Set the model.lr element of the model to track
                # the appropriately decayed learning rate.
                current_learning_rate = sess.run(model.lr)
                current_learning_rate *= args.decay_rate
                sess.run(tf.assign(model.lr, current_learning_rate))
                print("Decayed learning rate to {}".format(current_learning_rate))
            start = time.time()
            # Pull the next batch inputs (x) and targets (y) from the data loader.
            x, y = data_loader.next_batch()

            # feed is a dictionary of variable references and respective values for initialization.
            # Initialize the model's input data and target data from the batch,
            # and initialize the model state to the final state from the previous batch, so that
            # model state is accumulated and carried over between batches.
            feed = {model.input_data: x, model.targets: y}
            model.add_state_to_feed_dict(feed, state)

            # Run the session! Specifically, tell TensorFlow to compute the graph to calculate
            # the values of cost, final state, and the training op.
            # Cost is used to monitor progress.
            # Final state is used to carry over the state into the next batch.
            # Training op is not used, but we want it to be calculated, since that calculation
            # is what updates parameter states (i.e. that is where the training happens).
            train_loss, state, _, summary = sess.run(outputs, feed)
            elapsed = time.time() - start
            global_seconds_elapsed += elapsed
            writer.add_summary(summary, e * batch_range[1] + b + 1)
            if avg_steps < 100: avg_steps += 1
            avg_loss = 1 / avg_steps * train_loss + (1 - 1 / avg_steps) * avg_loss
            print("{:,d} / {:,d} (epoch {:.3f} / {}), loss {:.3f} (avg {:.3f}), {:.3f}s" \
                .format(b, batch_range[1], e + b / batch_range[1], epoch_range[1],
                        train_loss, avg_loss, elapsed))
            # Every save_every batches, save the model to disk.
            # By default, only the five most recent checkpoint files are kept.
            if (e * batch_range[1] + b + 1) % args.save_every == 0 \
                    or (e == epoch_range[1] - 1 and b == batch_range[1] - 1):
                save_model(sess, saver, model, args.save_dir, global_step,
```

Figure A.20: Snapshot of the train.py file

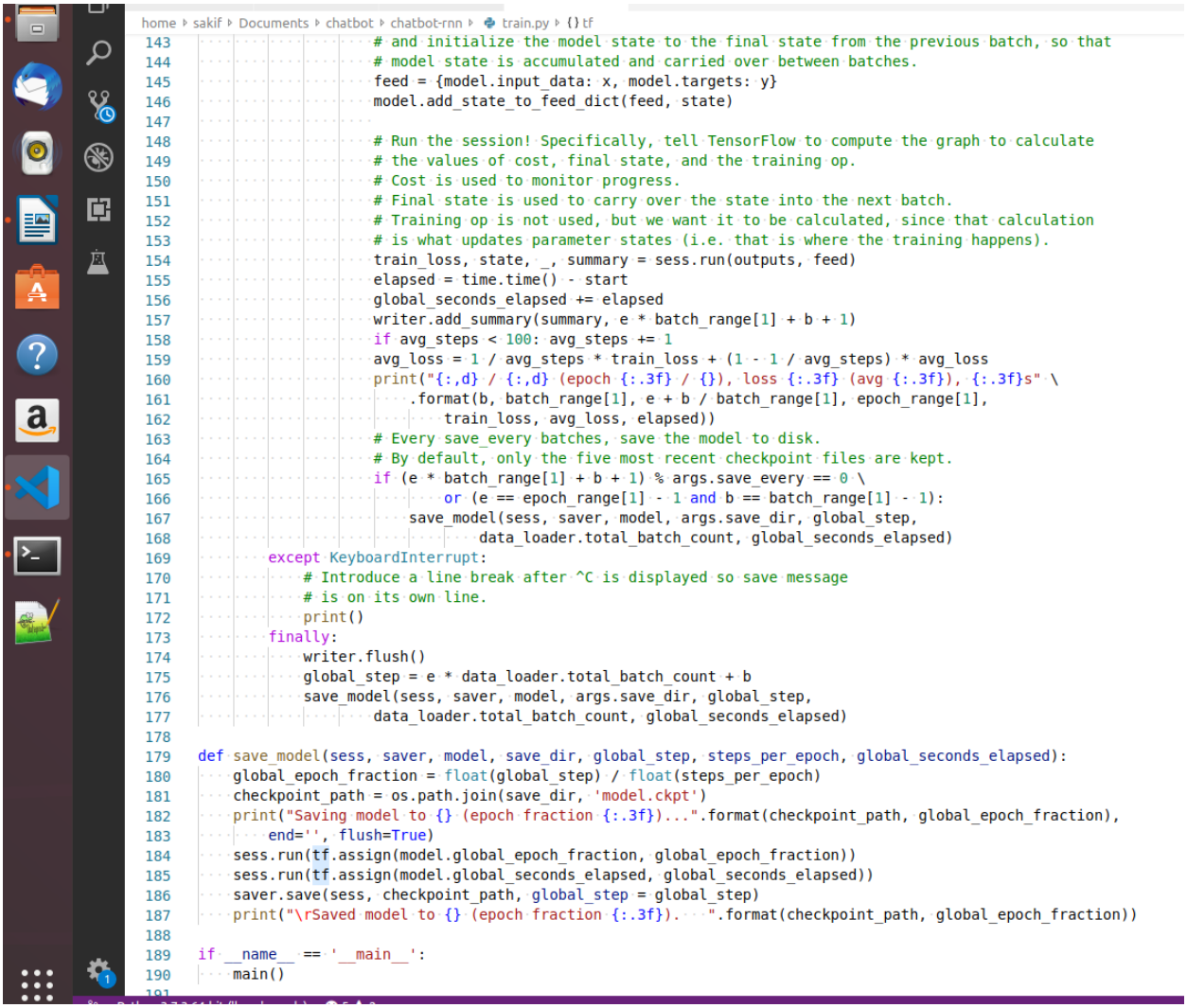


Figure A.21: Snapshot of the train.py file

APENDIX B

In Appendix B, we have shown some snapshots of our dataset.

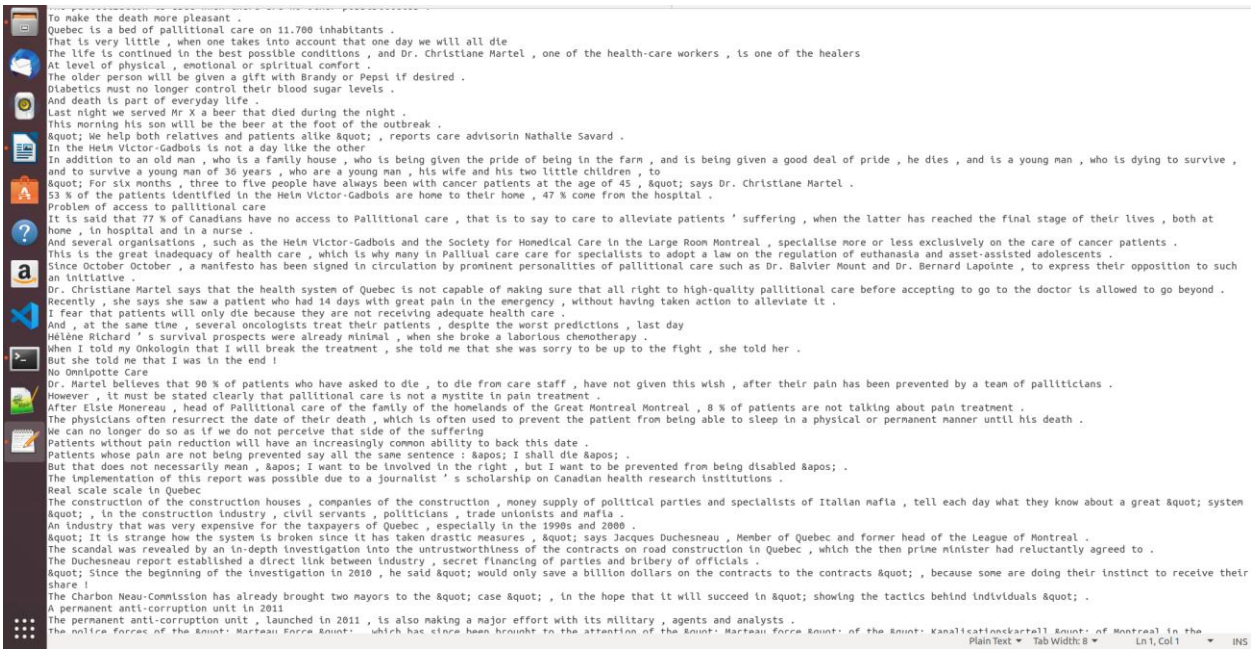


Figure B.1: Snapshot of the Dataset

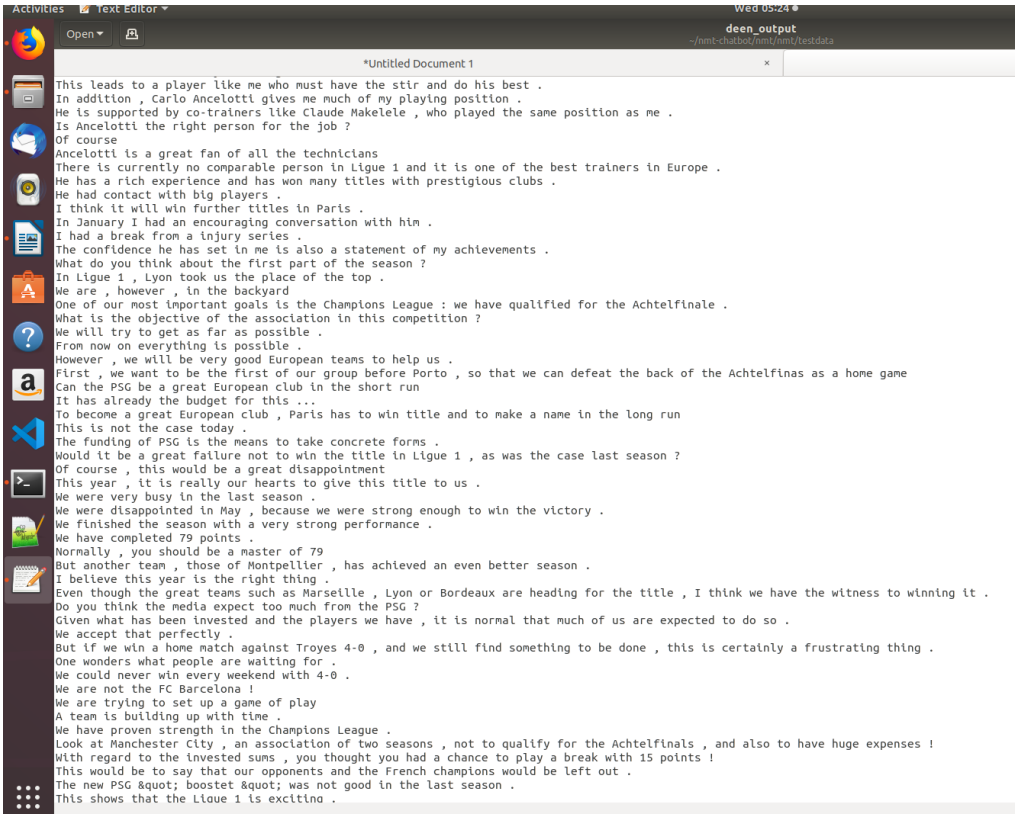


Figure B.2: Snapshot of the Dataset

Figure B.3: Snapshot of the Dataset

Figure B.4: Snapshot of the Dataset

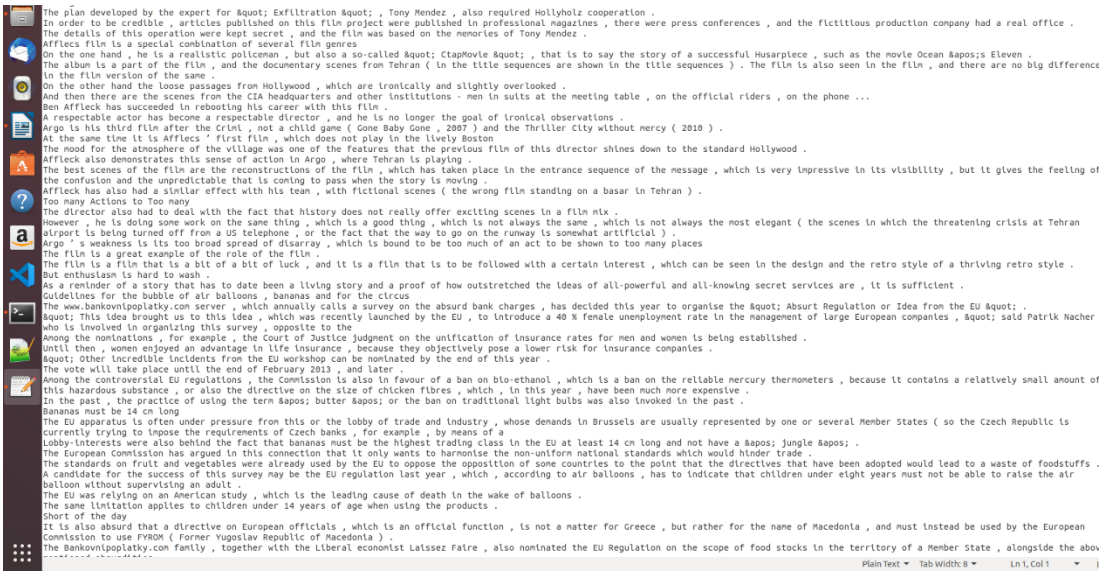


Figure B.5: Snapshot of the Dataset

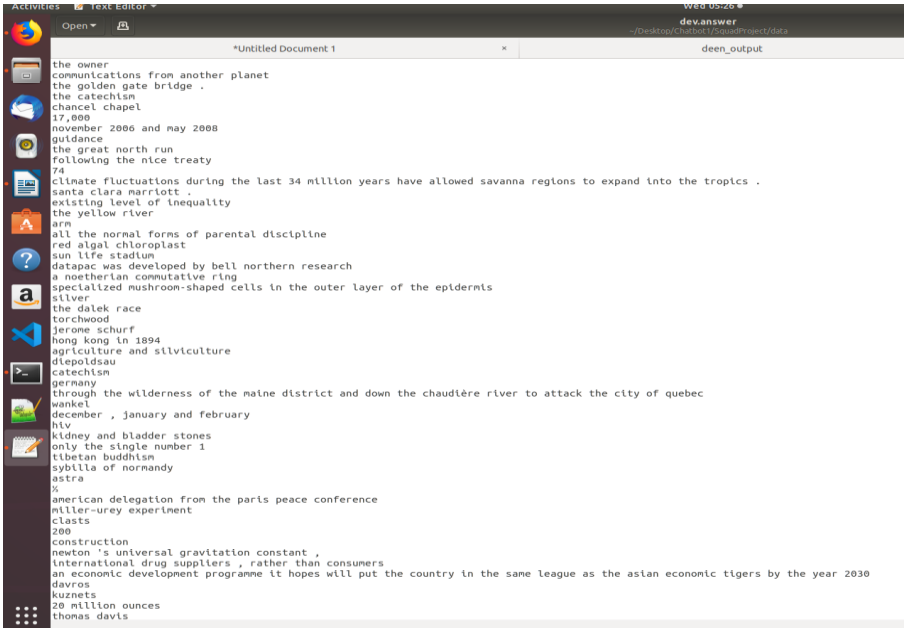


Figure B.6: Snapshot of the Dataset

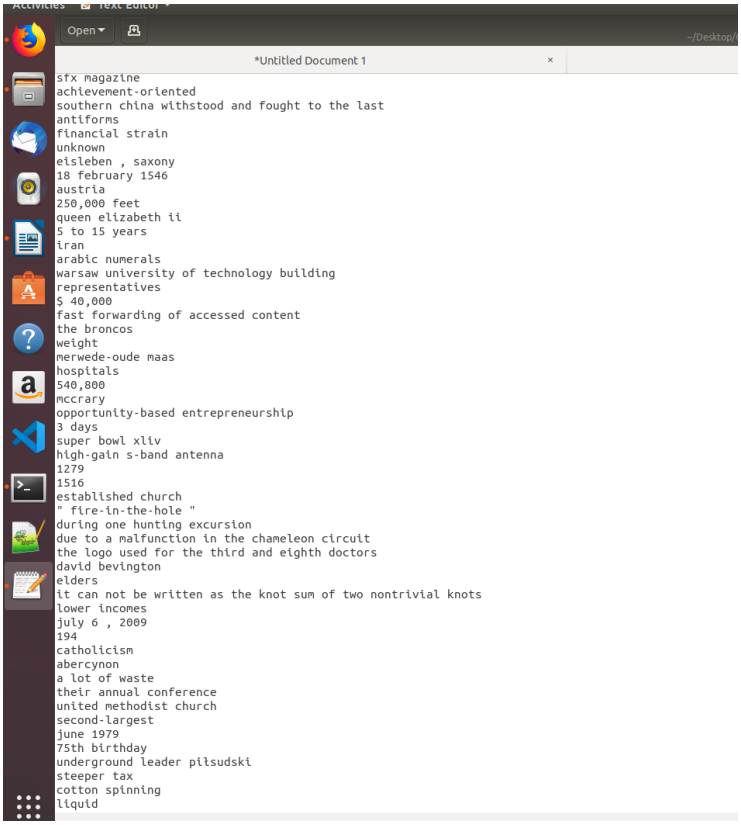


Figure B.7: Snapshot of the Dataset

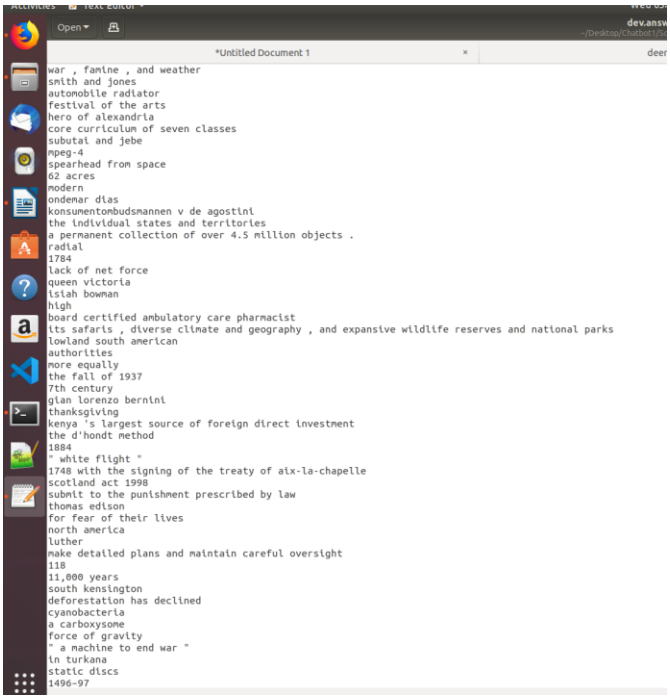


Figure B.8: Snapshot of the Dataset

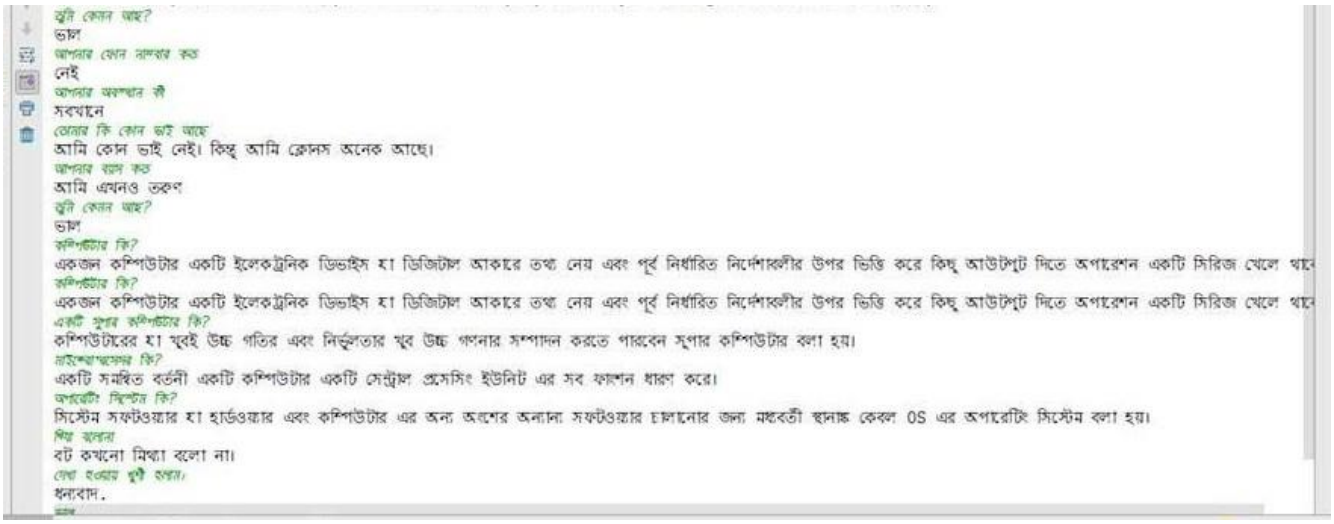


Figure B.9: Snapshot of the Dataset