Ain Shams University
Faculty of Computer & Information Sciences
Computer Science Department

A Conversational Dialogue engine (ChatterBot)

July 2017

Ain Shams University
Faculty of Computer & Information Sciences
Computer Science Department

A Conversational Dialogue engine (ChatterBot)

By:

Abdelrahman Hamdy [Computer Science]
Abdelhameed Hamed [Computer Science]
Ahmed Gamal [Computer Science]
Ibrahim Sharaf [Computer Science]
Yousr Ahmed [Computer Science]

Under Supervision of:

[Professor. Mostafa Aref],
Computer Science Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

[D. Wael Hamdy],
Computer Science Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

[TA. Zainab Fouad],
Computer Science Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

Acknowledgement:

We wish to express our thanks to the Head of our Computer Science Department **Prof. El-Sayed Mohamed El-Sayed El-Horbaty** for accepting our graduation project proposal, also we thank **professor. Mostafa Aref** for helping us to make a good, clear and well formatted proposal and documentation supporting us in many things related to project management that were one of the main factors to accept and finalize our project.

As we are working on such project that involves many researches, we needed some help and advices from experts who got involved and worked on the research field in general and on Machine Learning and Natural Language Processing specifically. **Dr. Ahmed Hani**, who was graduated from our college in 2015, helped us with his professional advices that helped when we stuck in a problem.

We want also to thank **ITWORX company**, the sponsors of our project for their help and support with valuable advices and support sessions.

We also very grateful to everyone who supported us.

ChatterBot Team.

Abstract:

The role of the system is to provide a chatbot that will be able to answer questions related to the admission procedure and it can ask the user questions too. It will provide a web interface for the users to interact with the system. A user is anyone who would like to visit the website and engage in a conversation.

The system is a machine learning based system which is trained on many variant datasets to be able to **generate** the answer for the questions whatever their domain (to be **open domain**).

The system is trying to solve the Turing test (imitation game) by creating a chatter bot which can pretend to be a human.

Another feature of the system is that you can train it language independent based on any language (Arabic, English, German …) and it can be trained on any data set (you can train it on your own chat corpus and expecting that it will act just like you).

The main idea of the system is based on paper published by Google in 2015 in which they used the sequence to sequence model (created by Google in 2014) to transform/translate statement after encoding it to another statement; suggesting that, despite optimizing the wrong objective function, the model is able to extract knowledge from both a domain specific dataset, and from a large, noisy, and general domain datasets.

After we get the answer for the given question, we train the model on this answer, so it can be able to gain the knowledge represented in the last conversation.

Table of Contents:

Acknowledgement 1

Abstract 2

List of Figures 4

List of Abbreviations 5

1- Introduction 6

1.1 Motivation 6

1.2 Problem Definition 6

1.3 Objective 7

1.4 Time Plan 8

1.5 Document Organization 9

2- Background 10

3- Analysis and Design 14

3.1 System Overview 14

3.1.1 System Architecture 15

3.1.2 System Users 18

3.2 System Analysis & Design 19

3.2.1 Use Case Diagram 19

3.2.2 Class Diagram 19

3.2.3 Sequence Diagram 20

3.2.4 Database Diagram 21

4- Implementation and Testing 26

5- User Manual 59

6- Conclusion and Future Work 63

6.1 Conclusion 63

6.2 Future Work 65

References 67

List of Figures:

Figure 1- Sequence to Sequence Model..... 17
Figure 2 - Sequence to sequence model with the LSTM part……..……18

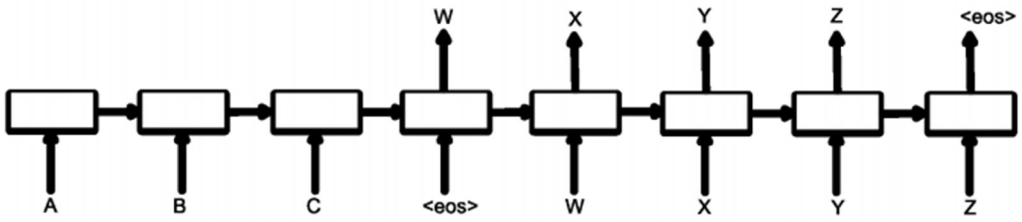


Figure 1-Sequence to sequence model

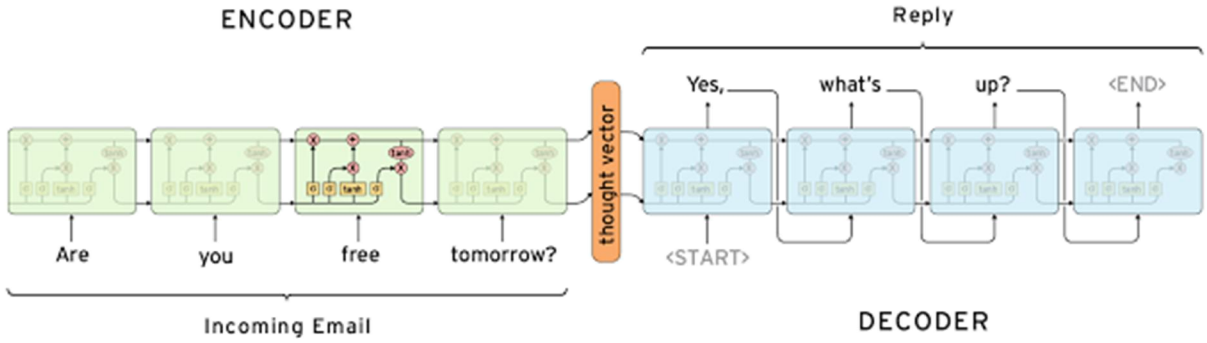


Figure 2-Sequence to sequence model with the LSTM part

List of Abbreviations:

Abbreviation:	Meaning:
ANN	Artificial Neural Networks
DNN	Deep Neural Networks
NLP	Natural Language Processing
NN	Neural Networks
LSTM	Long Short-Term memory
RNN	Recurrent Neural Networks
Seq2Seq	Sequence to Sequence model

List of Definitions:

Expression	Definition
DNN	Deep Neural Networks
response	A single string of text that is uttered as an answer, a reply or an acknowledgement to a statement.
statement	A single string of text representing something that can be said.
Untrained instance	An untrained instance of the chat bot has an empty database.

1- Introduction:

1.1 Motivation:

Conversational modelling is an important task in natural language understanding and machine intelligence. Although previous approaches exist, they are often restricted to specific domains (e.g., booking an airline ticket) and require handcrafted rules. In this project, we present an approach for this task which is sequence to sequence framework. Our model converses by predicting the next sentence given the previous sentence or sentences in a conversation. The strength of our model is that it can be trained end-to-end and thus requires much fewer hand-crafted rules.

1.2 Problem Definition:

Conversation with machines has been an important topic in both research and fiction for a long time. The possibility of having conversations with machines has always captured the imagination of people.
The idea of talking with machines has become popular with movies like 2001: Space Odyssey and Her, and with product like Apple's Siri, Microsoft's Cortana, Facebook's M and Google Now. It makes sense for natural language to become the primary way in which we interact with devices in the future, because that's how humans communicate with each other. Thus, the possibility of having conversations with machines would make our interaction much more smooth, natural and human-like. In this project, we explore different methods that model distinct aspects of language.
There are 4 types of chatbots according to their usage:

1. **Simple Basic chatbot:**
   Ask people a single question before connecting them with a human ("Hey, just in case we get disconnected, what's your email address?").

2. **Intelligent switchboard for live chat:**
   It asks you questions not so it can attempt to resolve your issue on its own, but so it can figure out who the best human is for you to talk to.

3. **Close domain assistant chatbot:**
   minimizing human-to-human interaction,
   try to help in solving simple problems.

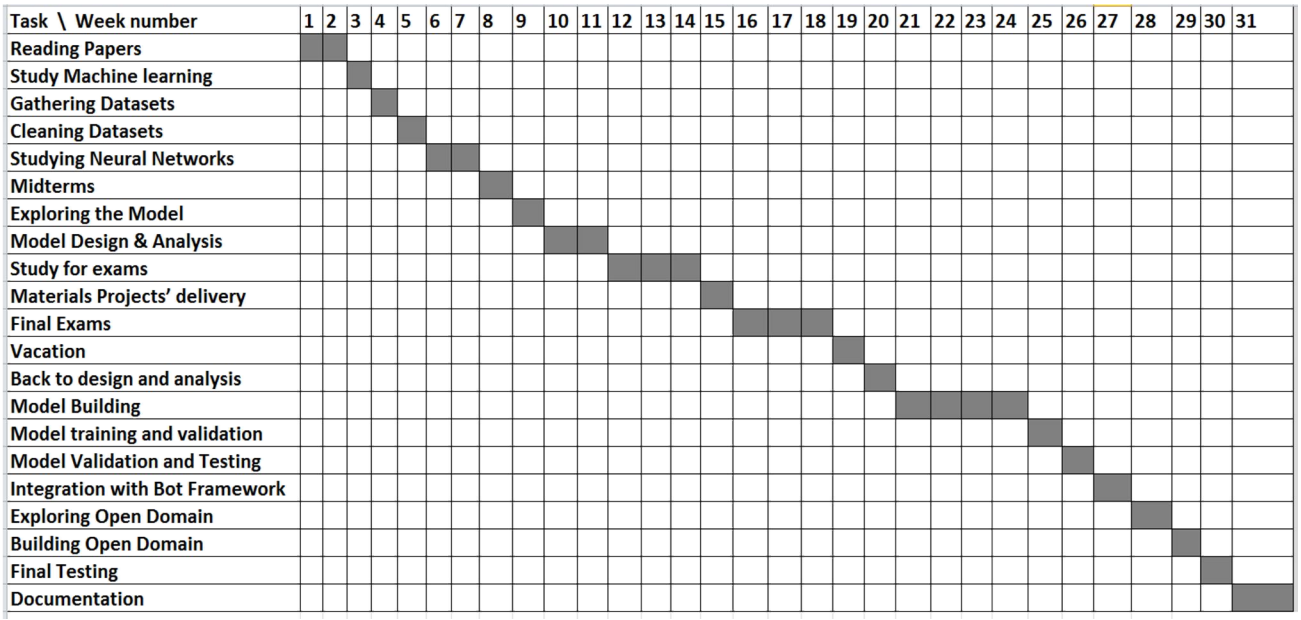
4. **Strictly human-to-chatbot open domain conversation:**
   To beat the Turing test — to become
   indistinguishable from humans during conversation.

1.3 Objectives:

- We aim to build a closed domain, conversational dialogue system based on large dialogue corpora using generative models which produce system responses that are autonomously generated word by word, opening up the possibility for realistic, flexible interactions.
- Extend the Chatbot to be an open domain one.
- We aim to build an open domain, conversational dialogue system based on large dialogue corpora using generative models.
- We aim to create a bot which can beat the Imation game.
- Find new ways to improve interactions between human and machine.

1.4 Time Plan:

Start Date	Weeks	Task Name
24/9/2016	Week 1	Reading Papers
1/10/2016	Week 2	Reading Papers
8/10/2016	Week 3	Study Machine learning
15/10/2016	Week 4	Gathering Datasets
22/10/2016	Week 5	Cleaning Datasets
29/10/2016	Week 6	Studying Neural Networks
5/11/2016	Week 7	Studying Neural Networks
12/11/2016	Week 8	Midterms
19/11/2016	Week 9	Exploring the Model
26/11/2016	Week 10	Model Design & Analysis
3/12/2016	Week 11	Model Design & Analysis
10/12/2016	Week 12	Study for exams
17/12/2016	Week 13	Study for exams
24/12/2016	Week 14	Study for exams
31/12/2016	Week 15	Materials Projects' delivery
7/1/2017	Week 16	Final Exams
14/1/2017	Week 17	Final Exams
21/1/2017	Week 18	Final Exams
28/1/2017	Week 19	Vacation
4/2/2017	Week 20	Back to design and analysis
11/2/2017	Week 21	Model Building
18/2/2017	Week 22	Model Building
25/2/2017	Week 23	Model Building
4/3/2017	Week 24	Model Building
11/3/2017	Week 25	Model training and validation
18/3/2017	Week 26	Model Validation and Testing
25/3/2017	Week 27	Integration with Bot Framework
1/4/2017	Week 28	Exploring Open Domain
8/4/2017	Week 29	Building Open Domain
15/4/2017	Week 30	Final Testing
22/4/2017	Week 31	Documentation



1.5 Document Organization:

In the **second** chapter we will talk about the field under which our project fall, overview of the science, problem and the work and researches done before in this field.

In the **third** chapter we will talk about the system analysis and design.

In the fourth chapter we will talk about overview of the code and what does each class and model do.

In the **fifth** chapter we will talk about how to install and use the system with all its options and features.

In the **sixth** chapter we will talk about the **results** we got, and the **further work**.

2- Background:

2.1 Project Initiation:

In Computer Science, an implementation word means a realization of prespecified technical functionalities and no functionalities requirements of an algorithm or system and software modules. An Example for that is Web Browsers. We use web browsers to be able to access the World Wide Web (WWW), we can illustrate the idea of the web browsers that we want to make an application that helps us to access the World Wide Web, so, before we begin the implementation we want to get general idea of how to access the World Wide Web with some functional and nonfunctional requirements. After we reach the idea, we begin the implementation which will be an easy task if the idea is fully illustrated and handling many issues that may appear when implementing it.

In this chapter we illustrate the idea of our project and mention the main factors that we took into consideration before we began the implementation.
We can conclude this chapter in 6 main points or factors, **The field of the project**, in which we discuss the fields which the project is going under them and need them and can add to them, **The science background**, in which we discuss the science we used and needed, **The work done before** and the **current / existing systems**, in which we show the papers and ideas similar to ours and the existing systems that do the same idea of our project with mentioning its functionalities, **The Technologies** we used, in which we discuss the technologies we used with the science to implement out idea and project.

The remaining point is **Stakeholder Analysis**, in which we discuss who can use our project and get benefit from it as the users are the main motivation for doing any application.

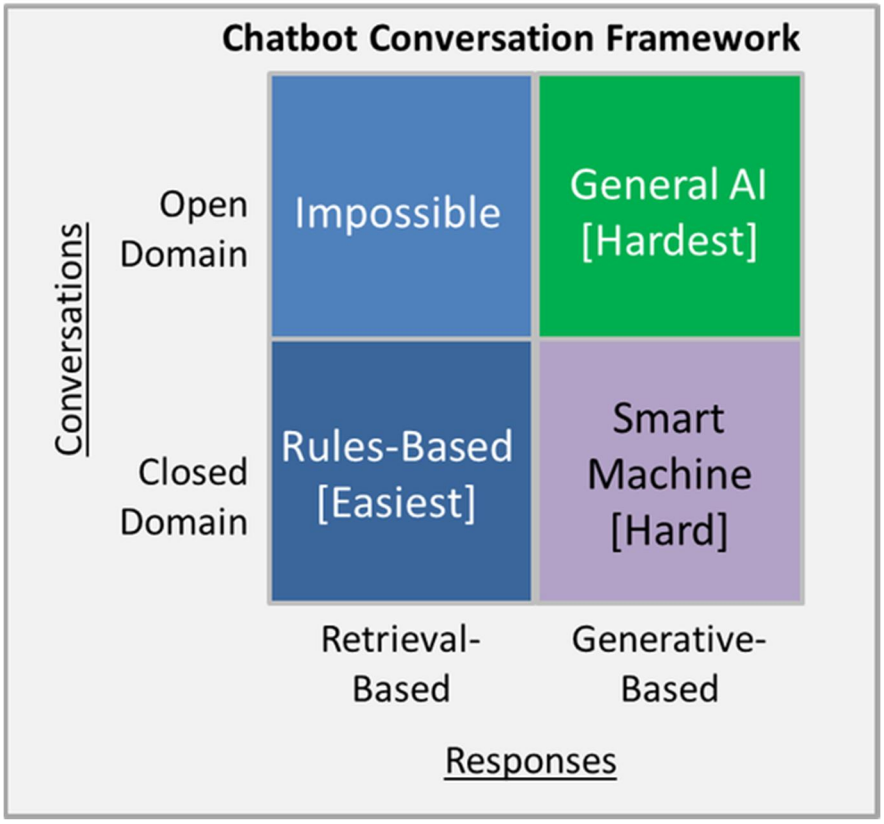
As we mention that we concern on the user's need, we tried to make the system efficient and easy as possible to use by creating a friendly user interface. Also, we are planning to add voice recognition to the system to make the user uses his/her voice to ask a question without needing to type on the keyboard.

2.2 Field of the project:

The project goes under the chatting bots category of science, which has roots to the Turing test, and it is a subfield of the natural language processing and understanding problem, which is attacked recently in efficient way using machine learning specially the neural networks and deep learning.

2.3 The Science background:

As we said before that our system falls under the topic of chatter bot creating, but there are many types of chatter bots, based on the way they get the response, and their domain.



Closed domain question answering deals with questions under a specific domain (for example, medicine or automotive maintenance), and can be seen as an easier task because NLP systems can exploit domain specific knowledge frequently formalized in ontologies.

Open domain question answering deals with questions about nearly anything, and can only rely on general ontologies and world knowledge. On the other hand, these systems usually have much more data available from which to extract the answer.

Retrieval-based models produce system use a repository of predefined responses .The heuristic could be as simple as a rule-based expression match they just pick a response from a fixed set.

Generative models produce system don't rely on pre-defined responses, they generate new responses from scratch. We translate from an input to an output.

2.4 The work done in the field:

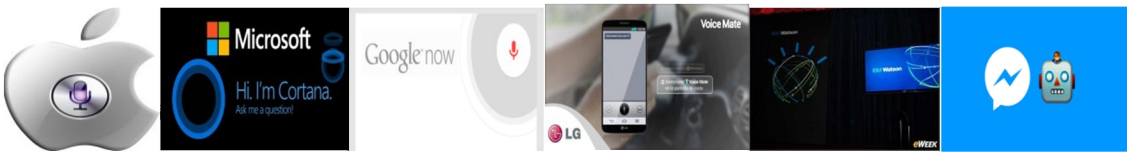
Many papers discussed how to create a close domain chatter bot based which generates the answers based on some answers saved to further use, which in this case the problem of creating chatter bot is directly transformed to a classification naïve problem. On the other side, no many papers discuss our system which is open domain chatter bot which can learn from the experience of the used, and it's replying based on generating the replies not retrieving them.

The most well-known paper in this field is a paper created by Google in 2015 called "**A Neural Conversational Model**".
Another well-known paper in the field is a paper created by Microsoft in 2016 called "**A Persona-Based Neural Conversation Model**".

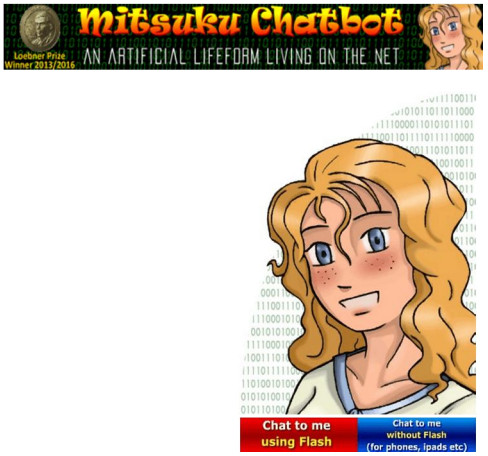
2.5 Evaluation of current and existing systems:

As we mention on the previous section, there are many existing chatbots like:

1. Google Now
2. Apple Siri
3. IBM Watson
4. Microsoft Cortana
5. Facebook M
6. LG Voice Mate



But the one we are trying to create at least a similar to it is Mitsuku:



Mitsuku is a Chatterbot created from AIML technology by Steve Worswick. Mitsuku won the 2013 and 2016 Loebner prize. Mitsuku is available as a flash game on Mouse breaker Games as well as on skype and on Kik Messenger under the username "Pandorabots."

2.6 Technology used:

We didn't use technologies except some external libraries supported by Python programming language like Tensorflow library created by Google in 2015 (Initial release).

2.7 Stakeholder Analysis:

The project mainly concerns on the users need, if the system is available for them, they would use it for asking what they want to know then the system gives back the answer. So, the users can use the system for educational or entertainment purposes. Imagine that you have an application that can answer all of your questions, it would be fantastic thing!

3- Analysis and Design:

3.1 System Overview:

The role of the system is to provide a chatbot that will be able to answer questions related to the admission procedure and it can ask the user questions too. It will provide a web interface for the users to interact with the system. A user is anyone who would like to visit the website and engage in a conversation.

The system is a machine learning based system which is trained on many variant datasets to be able to **generate** the answer for the questions whatever their domain (to be **open domain**).

The system is trying to solve the Turing test (imitation game) by creating a chatter bot which can pretend to be a human.

Another feature of the system is that you can train it language independent based on any language (Arabic, English, German …) and it can be trained on any data set (you can train it on your own chat corpus and expecting that it will act just like you).

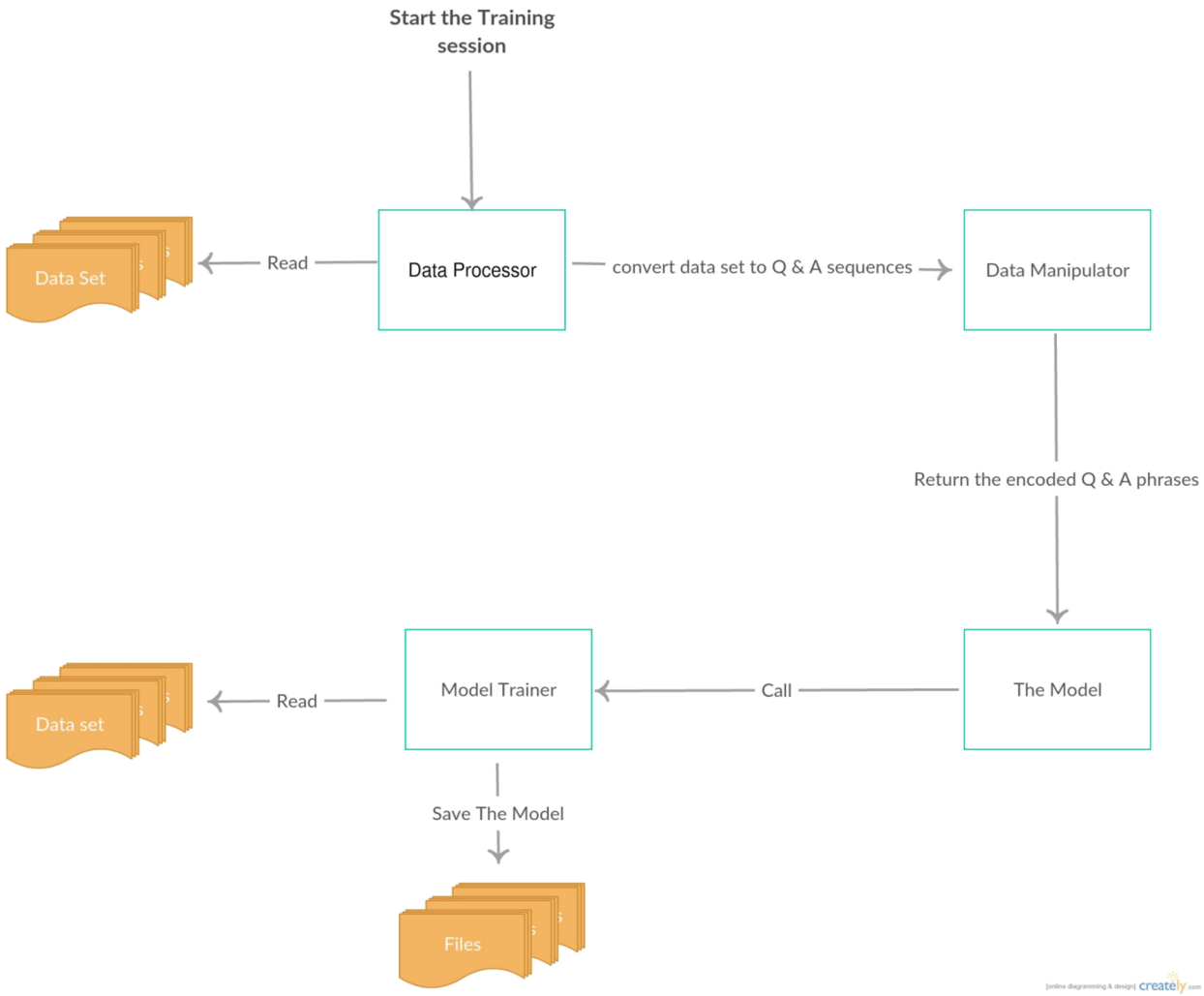
The main idea of the system is based on paper published by Google in 2015 in which they used the sequence to sequence model (created by Google in 2014) to transform/translate statement after encoding it to another statement; suggesting that, despite optimizing the wrong objective function, the model is able to extract knowledge from both a domain specific dataset, and from a large, noisy, and general domain datasets.

After we get the answer for the given question, we train the model on this answer, so it can be able to gain the knowledge represented in the last conversation.

3.1.1 System Architecture:

As long as the system is a machine learning based system, so it should has two architectures, one of them is the training architecture, and the other is the testing one.

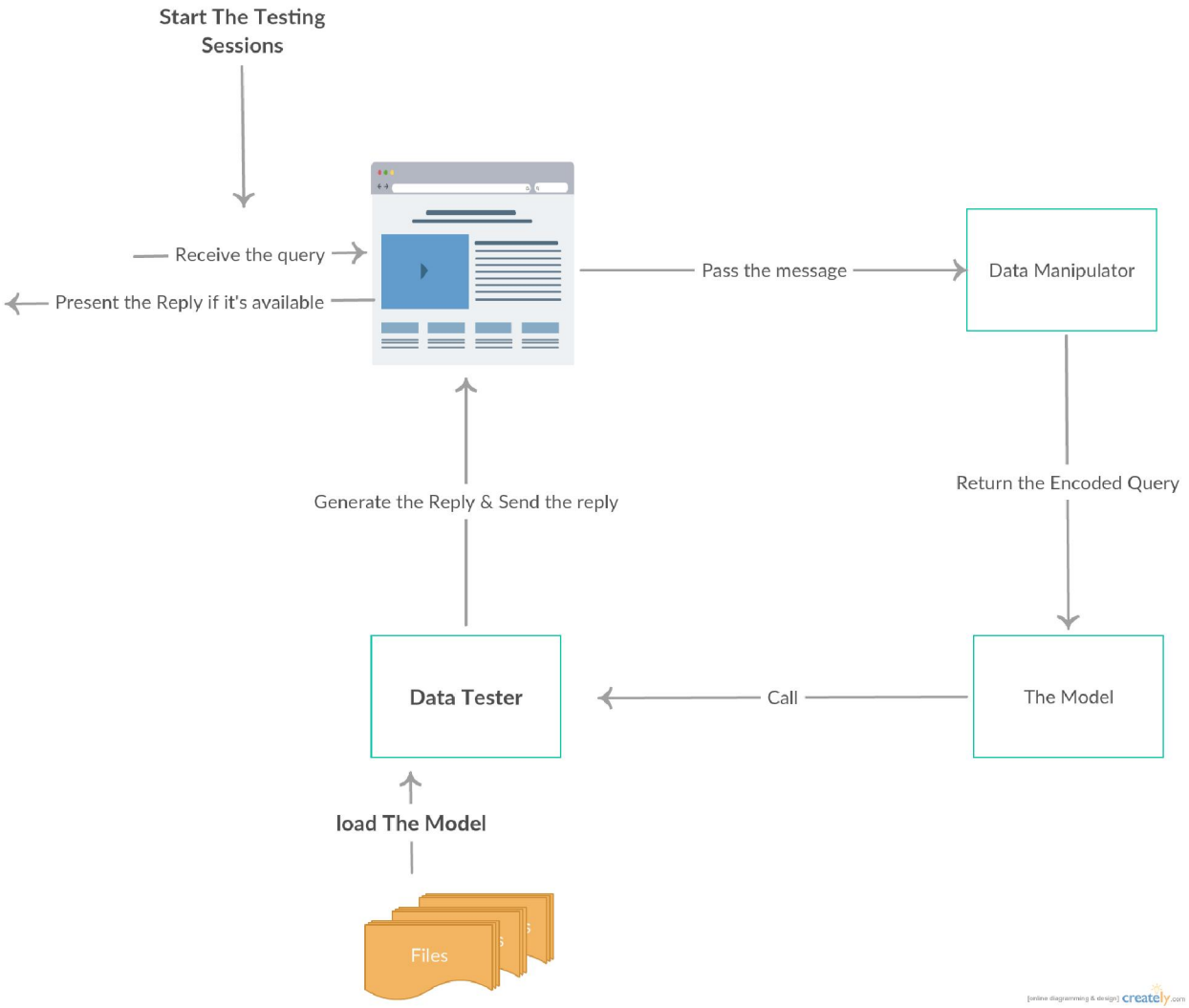
3.1.1.1 System Architecture: Training architecture:



The architecture flow is that, the data preprocessor reads the dataset corpuses, and converts them into Question and Answer style, so it will be able to use it later as training instances, and then it passes these instances to the data manipulator which encodes these sequences using the word emeddings made by Google in 2015. The model then call the training module to train itself on the encoded dataset, and try to adapt the weights of the model in order to make it better and more accurate in the

generalization phase, then it saves the weights in a file to read them back while testing.

3.1.1.2 System Architecture: Testing architecture:



The architecture flow is that, the website (the UI) receives a query (after logging in and so), the interface send this query to the data manipulator to encode it, then it send this encoded query to the model which calls the tester, the tester loads the saved model which is the weights of the model, and use these weights to generate the answer for this query, then it decodes the answer into words.

3.1.3. System requirements:

3.2.1. Functional Requirements

Open Domain:

- The user can talk about most topics (personality, core knowledge, small talk, general knowledge etc.).

Chatting:

- The system should answer most of questions properly and ask questions too.
- The system should allow user to chat.
- The system should inform the user if an answer isn't available to learn.

3.2.2. Non-Functional Requirements

User Interface:

- The system shall maintain an easy to use interface across all functionality and for all users.
- The clients' user interface should be compatible with all commonly used browsers, such as Internet explorer, Firefox, Google chrome and Safari.

Ethics:

- The system shall not store or process any information about its users just learn.

Maintainability:

- The system should be easy to maintain.
- There should be a clear separation of HTML and JavaScript interface code.

3.1.4. System Users:

A. Intended Users:

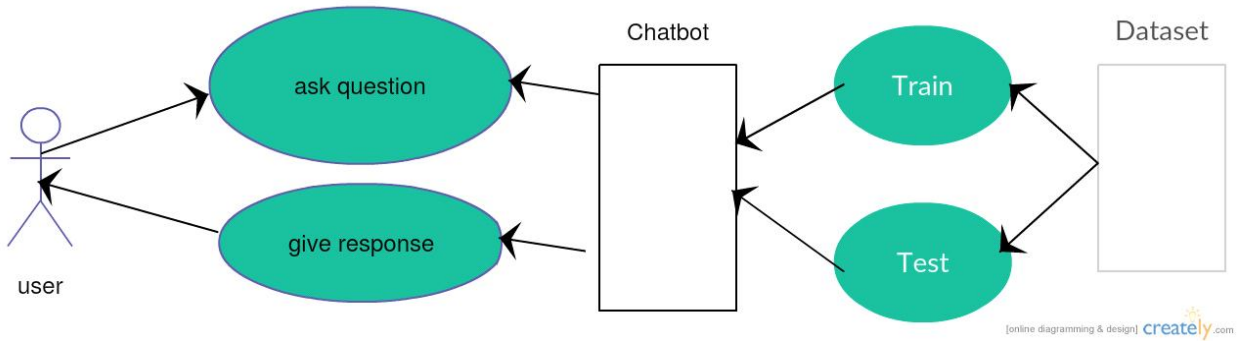
The system is represented by simple UI, and hosted on a server for anyone to use, its user specific so your experience with your it will not affect other users.

B. User Characteristics:

The user need no specific skills, you interact with the bot via chatting, so you need only to be able to write in English with fair grammar and expressions.

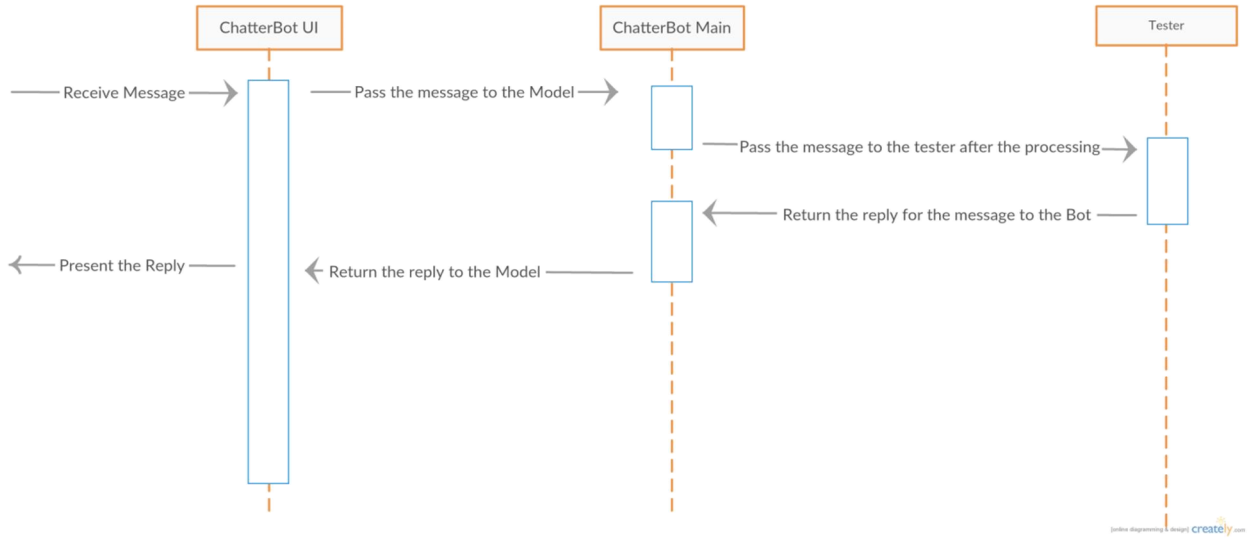
3.2 System Analysis & Design

3.2.1 Use Case Diagram:



The user will ask a question to the system, the system will be trained or tested based on if the weights are ready to be used or this is the first time to use the system, if the first time the system will be trained, but if it's not the first time, the system will load the specific weights specified for this user.

3.2.2 Sequence Diagram:

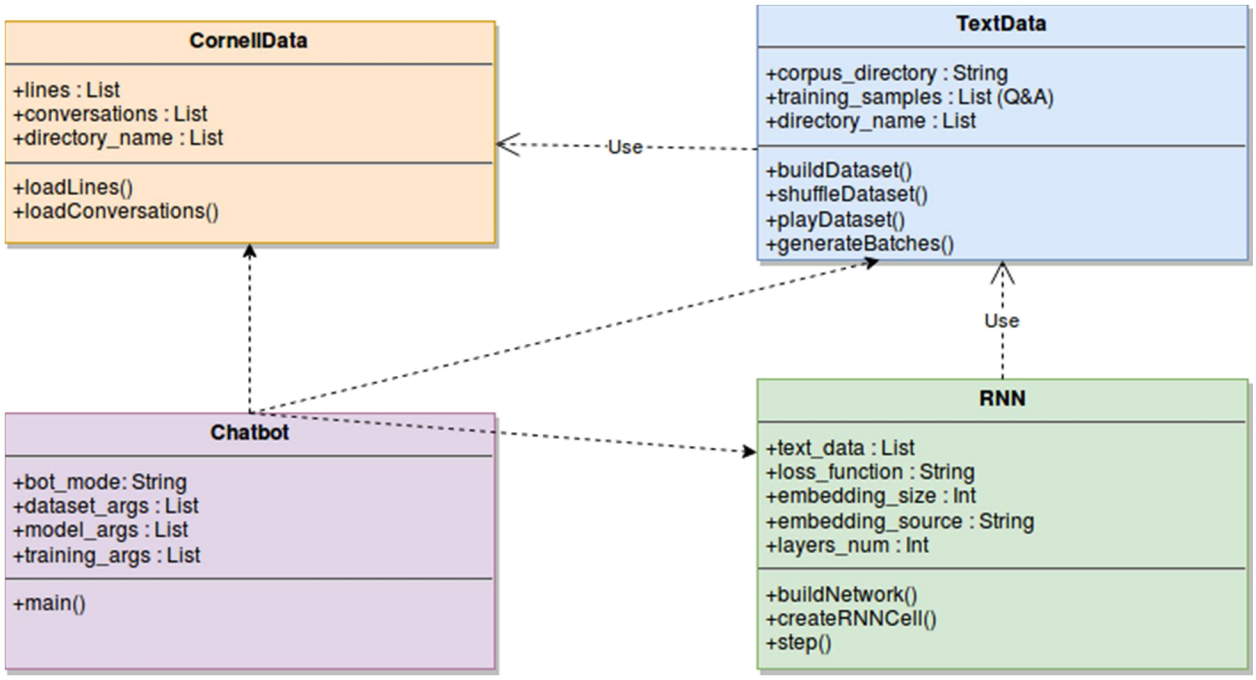


The user will ask question via the UI, the UI will send this message to the bot, then encoding of the question is done, and it goes to the trainer which will generate the answer for the question or the reply.

3.2.3 Database Diagram:

The system has no databases, the weights of the model is updated and saved in a file inside the project itself to make it faster and easier accessing it to write/read.

3.2.4 Class Diagram:



The model consists of 4 classes:

Class 1: Data Manipulation:

1. **Purpose:**

   This class is used to read the dataset, rearrange it to be suitable for further uses, gets word embedding for each conversation, generate training batches.

2. **What it uses:**

   it uses the Dataset, tensorflow, NLTK, SpaCy.

3. Functions Overview:

3.1. Function (get_conversations) :

[Gets dataset conversations]

[return: list of lists, contains related conversation lines]

3.2. Function (word_to_vec):

[Takes the conversations as an input data and split them in array of conversations then we take each one of this array's indexes and split them to sentences and words , we take each word and use the tensorflow's functions to extract the vector from it.]

3.3. Function (doc_to_vec):

[Takes a conversation line, split words then calls the word_to_vec function for each word,].

[Calculates the average of all vectors to get the doc vec].

3.4. Function (generate_batches):

[Prepares the batches for the current epoch, get a list of the batches for the next epoch].

3.5. Function (load_lines):

[Populate lines dictionary]

[Return: Dict of line id with each utterance]

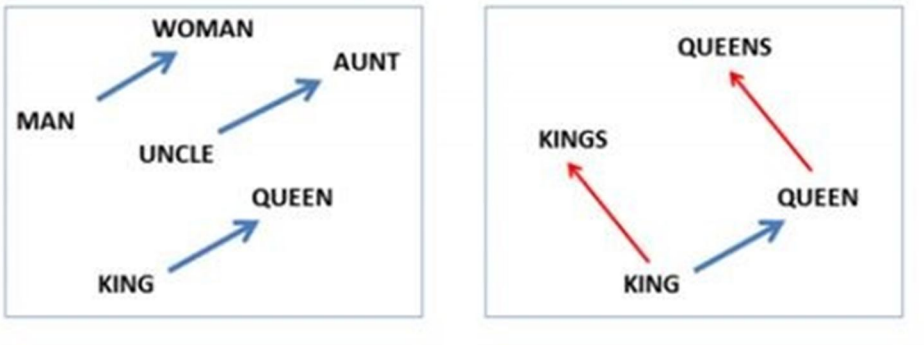
3.6. Function (load_conversations):

[Get ordered conversation lines]

[return: list of lists, contains related conversation lines]

4. Science beyond what the class does:

Representing the words using Word2Vec:



Tomas Mikolov, COLING 2014

In order to keep the close words close in their vector values.

Class 2: The Neural Network:

1. Purpose:

This class is used to prepare the neural networks layers, to create the sequence to sequence model, and to prepare it to be trained.

2. What it uses:

it uses the Dataset represented in vectors via the Data Manipulation class

3. Functions Overview:

3.1. Function (Prepare_DataSet) :

[Gets dataset conversations]

[return: vector feature to each phrase]

3.2. Function (Prepare_RNN_Layers) :

[Gets dataset conversations converted to vectors]

[Create the two-layer models of RNN(decoder and encoder)]

3.3. Function (Prepare_Seq2seq):

[Gets the RNNs]

[Returns the seq2seq Model]

4. Science beyond what the class does:

Represented in the Model file.

Class 3: Model Training:

1- Purpose:

Train the model to be able to be tested or used.

2- What it uses:

The Model created in the Neural Network class.

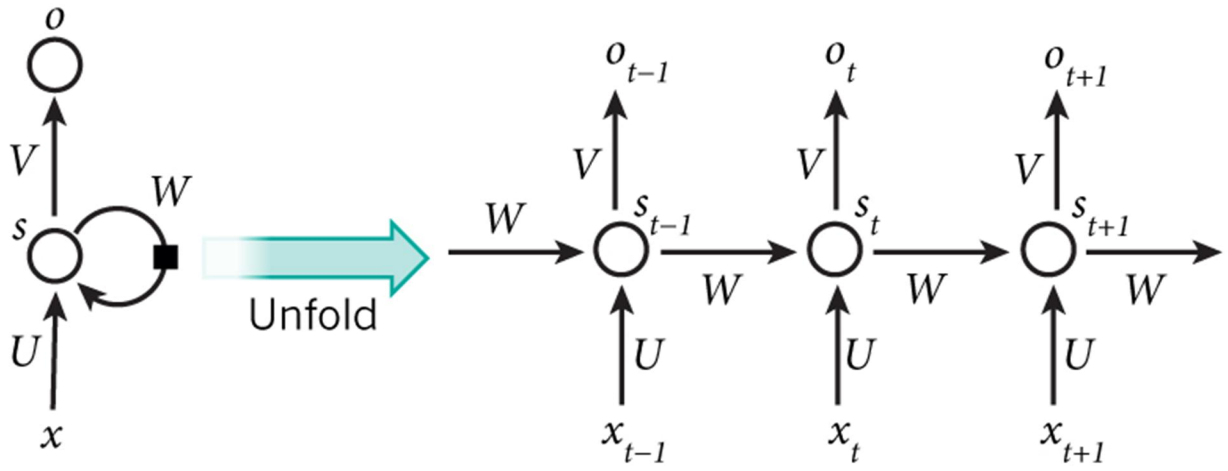
3- Functions Overview:

a. Function (Train_The_Model) :
   [Gets The Model]
   [return The Model Trained]

b. Function (Save_The_Model):
   [Saves the gotten weights via the training process.]

c. Function (Load_The_Model):
   [Return the trained model].

4- Science beyond what The class does:



a. Given a loss function, our objective is to find the parameters U, V and W that minimize it for the given data during the training phase.
b. Traditionally, we'd use the SGD method with BPTT to optimize the parameters, but this method is slow and suffers from the vanishing gradient problem.
c. We use LSTM to fix the vanishing gradient problem.
d. We use a variation of SGD (RMSPROP) to increase the performance.

Class 4: Tester:

1- Purpose:

Use the chatbot to chat, test it and calculate the BLEU.

2- What it uses:

The trained model from the Model training class.

3- Functions Overview:

a. Function (Read_The_Query) :
[Gets a query from the user]

[return The query enhanced]

b. Function (Reply):
[Take the enhanced query]
[Call the Model with this Query]
[Return: the Model output]

4- Science beyond what each function does:

After calculating the weights in the trainer class, we can use them to predict the output via the query passing by the weights [Explained more in the model file]

Class 5: The interface:

It consists of 2 classes:

- **Front_End_Handler class**

  A class to manage the web app user interface.
- **Back_End_Handler class**

  A class to prepare the data to be shown, and manipulate the queries and chatting process.

4- Implementation and Testing

4.1. Implementation Approach

In this chapter, we talk about the implementation phases for each module, the talk contains deep technicalities that has been used and tried on our system.
We also mentioned a lot of references that we have used during the implementation phases, so, you can check them if you want to find more details about something not clear in the illustration.

Knowledgebase

We used 2 open source data sets Cornell movie-dialogs corpus and Open Subtitles 2016.

Data cleaning:

We cleaned the datasets using BeautifulSoup and NLTK.

Word representation: Word Embedding:

Usually known as Word embedding or word vectors, this is based on representing a word as a vector. Word vectors can be trained on huge text datasets; they provide generalization for systems trained with limited amount of supervised data. More complex model architectures can be used for obtaining the word vectors (Neural net language model with multitask learning (Collobert & Weston, 2008)).

4.2. Programming Language and packages:

Languages:

- Python, R to clean data.
- JavaScript to UI and interface.

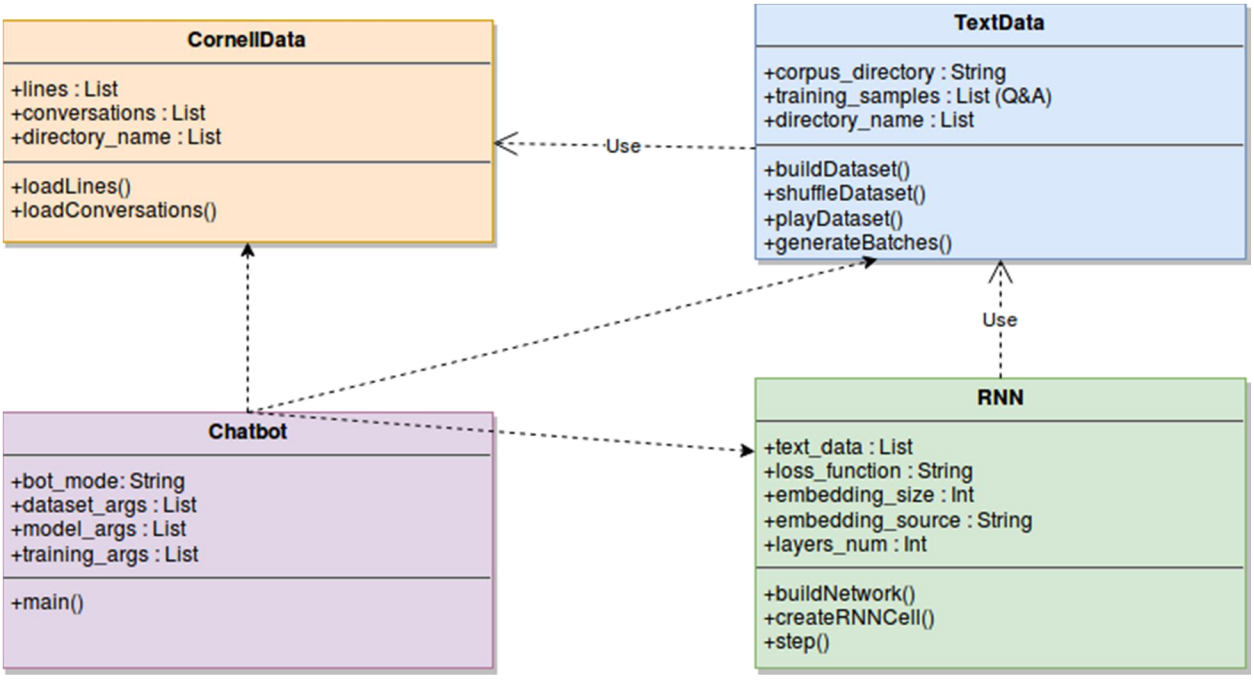
Packages:

- Anaconda, NLTK to clean data.
- TensorFlow to build the chat Bot.

4.3. User Interface

We will build a web application for the Bot, which allows the user to interact with the Bot and start a conversation Using Python's framework (Django) with CSS and Javascript.

4.4. Classes overview (The Implementation):



The model consists of 4 classes:

Class 1: Data Manipulation & Preprocessing

Extracting Answers based on Word vector space.

Problem definition:

One of the biggest problem in the Natural language Processing field is : 1) Selecting the features that represent the sentence in a digitized model that a computer can understand.
2) Unlike the image recognition Most of the features that can represent the sentence in a semantic manner is variable length size (eg. Parse tree, dependency graph, etc..) which is needed to be represented in a vectorized manner to be able to be used.

Solution:

To solve the previous stated problem we need first a way to represent a word in a digitized model, there are three basic common solutions to do so :

1) Ngrams :

Standard approach to language modeling
Task: compute probability of a sentence W

$$P(W) = \prod_i P(w_i|w_1 \dots w_{i-1})$$

And can be simplified to trigrams

$$P(W) = \prod_i P(w_i|w_{i-2},w_{i-1})$$

2) Word Classes:

It's one of the most successful NLP concepts in practice.
Similar words should share parameter estimation, which leads to generalization.
Example:
Class1 = (Yellow, green, blue, red)
Class2 = (Italy, Germany, Spain, France)
Usually, each vocabulary word is mapped to a single class (similar words share the same class).
There are many ways how to compute the classes – usually, it is assumed that similar words appear in similar contexts
It's an alternative way instead of just counting the words, we can use also counts of classes, which leads to generalization (better performance on

novel data). There are many ways how to compute the classes – usually, it is assumed that similar words appear in similar contexts.

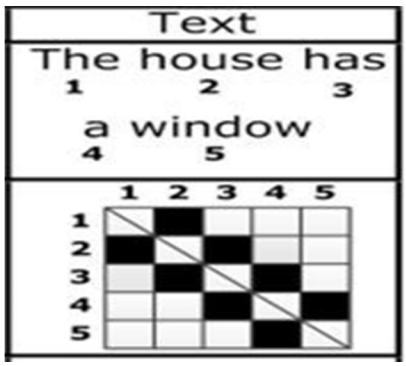
3) BagofWords (BoW) :

Simple way to encode discrete concepts such as words, generally converts the sentence into a vector of frequency of N dimensions, where N is the vocabulary size.

```
vocabulary = (Monday, Tuesday, is, a, today)
Monday Monday      = [2 0 0 0 0]
today is a Monday  = [1 0 1 1 1]
today is a Tuesday = [0 1 1 1 1]
is a Monday today  = [1 0 1 1 1]
```

Dimensional embeddings; the latter are then combined componentwise with an operation such as summation. The resulting combined vector is classified through one or more fully connected layers.

One of the disadvantages in this model it can cause a confliction in a sentence representation because of the loss of order of the words.
The loss of order problem can be solved by either bagofNgrams or Adjacency Matrix of bag of words. Example:



These are the main features that can be extended to give a high level of sentence representation (eg. Named Entity Recognition, Parse Tree, Text completion, Word embeddings)
Question Answering

After getting the top candidates from the Information retrieval based on sentence shallow processing, in order to get the right answer we should look at the problem from the other point of view from the Semantic perspective and this is the main focus of this module to answer the question
The main characteristics this problem to be able to answer the question is :

1) **Recognizing context focus.**
2) **Question type focus.**

1) Recognizing context focus :

1.1) Choosing the headwords of the context

The context headwords is the most important words of the sentence that controls the sentence context. can be easily extracted and learned by a simple machine learning algorithm such as Support vector machine or Neural network based on the following features :

1) Part of Speech Tagging:

Usually NN, JJ, VB, NNS, RB, VBN have high weights to be headwords.

2) Dependency graph:

The dependency relation between the current word and the headwords.

3) NER: if the current word is a Named Entity.

this method should be applied to the question and the answer to extract the context head words.

1.2) Finding the similarity

Calculate the context similarity between the question and the answer. To solve this problem, we should first solve its subproblem which is finding the similarity between two words.
There's two common solutions to do so:

1.2.1) Word Hypernyms:

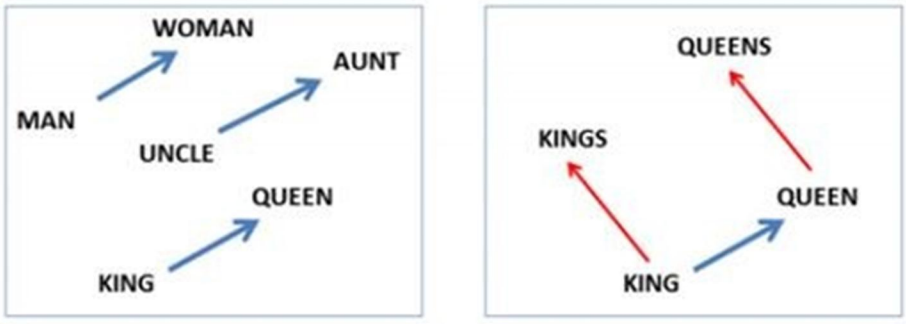
Each word has its hypernyms which is extracted from a dictionary (eg. Wordnet), Hypernyms of a single word share the same semantic meaning, this method is about to group them in a single class to be able to be used in the calculation of the similarity. very simple way if compared to the other methods but not very efficient.

1.2.2) Word Embedding

Usually known as Word embedding or word vectors, this is based on representing a word as a vector. Word vectors can be trained on huge text datasets; they provide generalization for systems trained with limited amount of supervised data. More complex model architectures can be used for obtaining the word vectors (neural net language model with multitask learning (Collobert & Weston, 2008)).

1.2.2.1) Word vectors – linguistic regularities

Recently, it was shown that word vectors capture many linguistic properties (gender, tense, plurality, even semantic concepts like "**capital city of**"). We can do nearest neighbor search around result of vector operation "**King – man + woman**" and obtain "**Queen**" (Linguistic regularities in continuous space word representations (Mikolov et al, 2013)).



Tomas Mikolov, COLING 2014

1.2.2.2) Word vectors – various architectures

Neural net based word vectors were traditionally trained as part of neural network language model (Bengio et al, 2003).

- Bigram NNLM

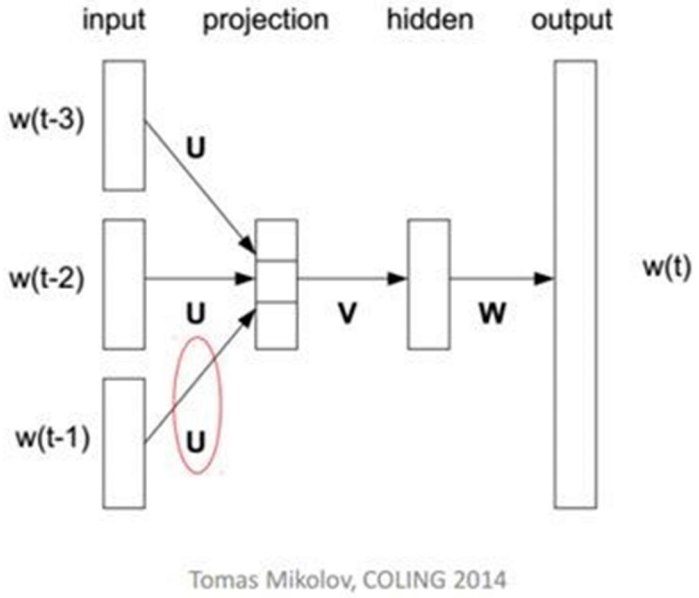
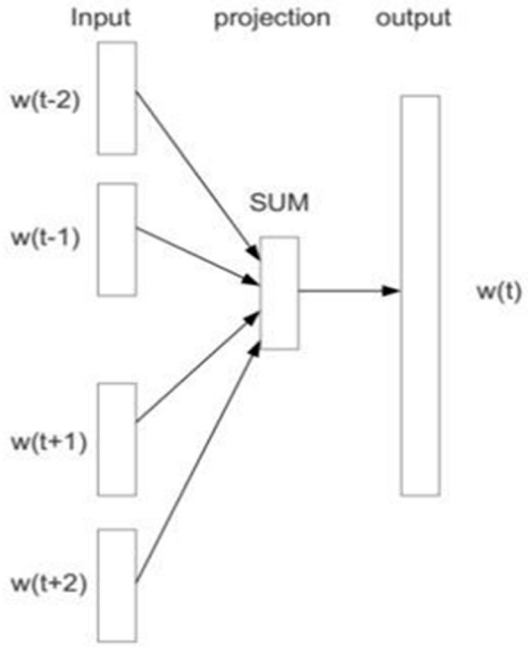


figure 5.1.9

Neural Network Language Modeling models consists of input layer, projection layer, hidden layer and output layer, this model can be extended by adding more context to the bigram NNLM. the projection layer will be explained in the next section.

- **CBOW**



Continuous bagofwords model adds inputs from words within short window to predict the current word. This model differs from the NNLM in the following characteristics :
The weights for different positions are shared.
Computationally much more efficient than normal NNLM.
The hidden layer is just linear.

- skipgram NNLM

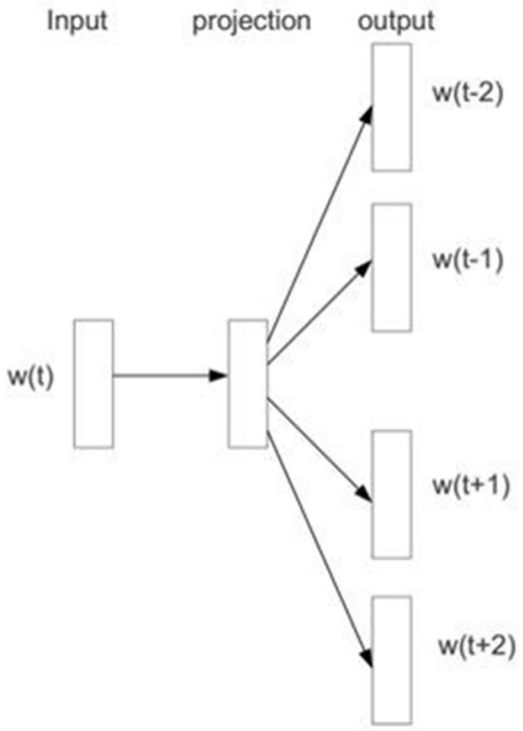


figure 5.1.11

We can reformulate the CBOW model by predicting surrounding words using the current word, performance is almost similar to CBOW if both architectures trained for sufficient number of epochs.

The projection layer :

The projection layer maps the discrete word indices of an ngram context to a continuous vector space.
Each neuron in the projection layer is represented by a number of weights equal to the size of the vocabulary. The projection layer differs from the

hidden and output layers by not using a nonlinear activation function. Its purpose is simply to provide an efficient means of projecting the given ngram context onto a reduced continuous vector space for subsequent processing by hidden and output layers trained to classify such vectors. Given the oneorzero nature of the input vector elements, the output for a particular word with index i is simply the i th column of the trained matrix of projection layer weights (where each row of the matrix represents the weights of a single neuron).

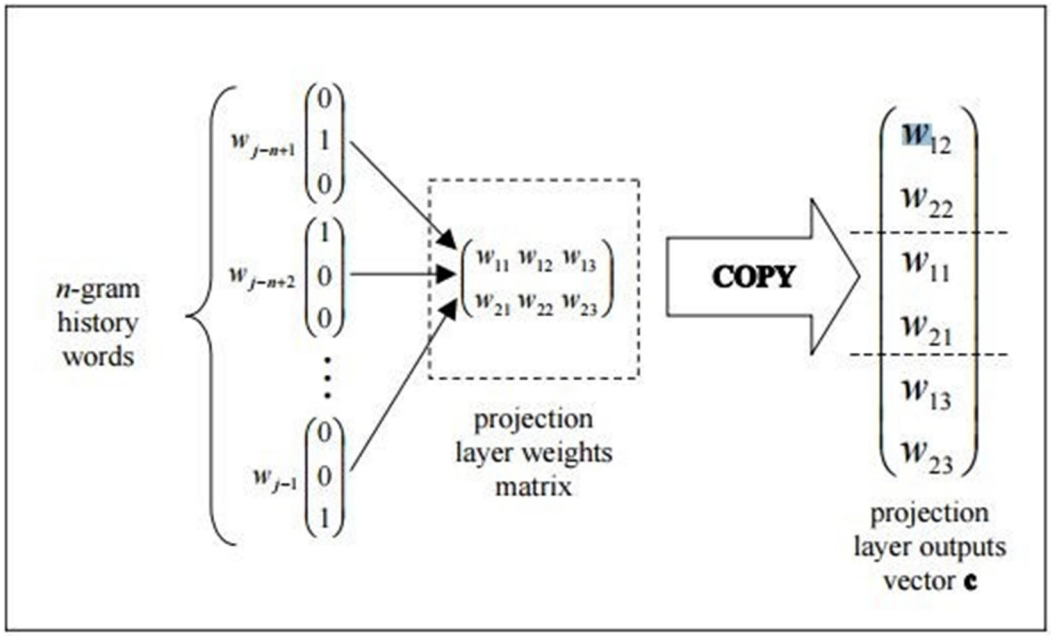


Figure 2 – Formatting the projection vector from an *n*-gram context

figure 5.1.12

For an ngram history of length h n 1 and a projection layer size of P neurons, the final output from the projection layer is a vector of real values of length h * P. The resulting projection vector is used as the input to the next layer in the network: the hidden layer.

1.2.1.3) Word vectors – training:

Output layer size is very large – equal to the vocabulary size can be easily in order of millions. Two methods are used to solve this problem:

1) Negative sampling 2) Hierarchical softmax

Before the training it's very useful to sub sample the frequent words (such as 'the', 'is', 'a', etc…) during training.

Also Nonlinearity does not seem to improve performance of these models, thus the hidden layer does not use activation function.

· Negative sampling

Instead of propagating signal from the hidden layer to the whole output layer, only the output neuron that represents the positive class + few randomly sampled neurons are evaluated.
The output neurons are treated as independent logistic regression classifiers. this method makes the training speed independent on the vocabulary size.

· Hierarchical Softmax

In traditional NNLM, we need to compute the conditional probabilities of all vocab words given a history:
p(w|history) $w \in Vocabulary$,
finally perform a normalization (namely softmax). Obviously, such kind of operation requires huge computation, especially on the whole vocab.

To solve this problem this solution was proposed with the following processes:

1) First, build a Huffman tree based on word frequencies. As a result, each word is a leaf of that tree, and enjoys a path from the root to itself;

2) Now, each step in the search process from root to the target word is a normalization. For example, Root chooses left subtree with probability 0.7 and chooses right subtree with probability 0.3. This is a normalization step in this tree level. Naturally, the final probability for finding the target word is the continuous multiplication of probabilities in each search step.

3) To compute the Hierarchical Softmax probability from this equation

$$p(w|w_I) = \prod_{j=1}^{L(w)-1} \sigma([n(w,j+1) = ch(n(w,j))] \cdot {v'_{n(w,j)}}^T v_{w_I})$$

Some comments to understand this formula:

1） The whole process is a continuous multiplication. Product ${v'_{n(w,j)}}^T v_{w_I}$ can be treated as a kind of "**similarity** " between history (also called "**Input** " or "**context** ") and the jth ancestor. Note that all the leaves and nonleaves (i.e., ancestors) have initialized embeddings which will be updated gradually. So, above "similarity " is computed using embeddings.

Its intuition is that if given context is more similar with the ancestors of certain word, then the word has higher probability to be the target word.

2) Given "**similarity** " score, Hierarchical Softmax converts it to two probabilities to search left subtree and right subtree. That's achieved by using

$\sigma([n(w,j+1)=ch(n(w,j))]\cdot v'^{T}_{n(w,j)} v_{w_I})$. Note that the equation in the symbol " $[\cdot]$" is 1 or +1. It makes $\sigma(+x)+\sigma(-x)=1$. This is a normalization step. The original

sentence says "**let ch(n) be an arbitrary fixed child of n**". Hence, we can always think that ch(n) means the left child. So, if the j+1 ancestor of target word w is the left child of its j ancestor, then the probability of j choosing j+1 is

$\sigma(+0.78234)$ (let's assume the similarity between context and ancestor j is 0.78234), otherwise, the probability is $\sigma(-0.78234)$.

3) Now, we know how to compute the probability of ancestor j choosing j+1 ($j \in \{1,2,\cdots,L(w)-1\}$). Multiplying them one by one equals to the normalized probability of target word given context or history.

1.2.2.4) Comparison of performance

Model	*Vector Dimensionality*	*Training Words*	*Training Time*	*Accuracy [%]*
Collobert NNLM	50	660M	2 months	11
Turian NNLM	200	37M	few weeks	2
Mnih NNLM	100	37M	7 days	9
Mikolov RNNLM	640	320M	weeks	25
Huang NNLM	50	990M	weeks	13
Skip-gram (hier.s.)	1000	6B	hours	66
CBOW (negative)	300	1.5B	**minutes**	**72**

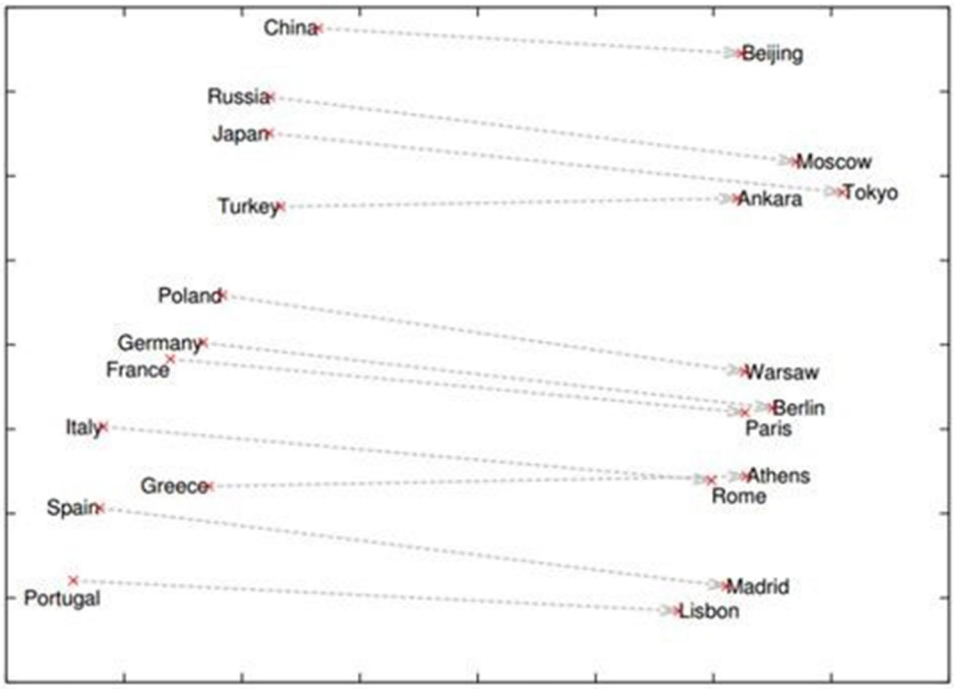
1.2.2.5) Scaling up

The choice of training corpus is usually more important than the choice of the technique itself. The crucial component of any successful model thus should be low computational complexity.

1.2.2.6) Examples

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

1.2.2.7) Visualization using PCA



The same technique that's used to represent the words can be used to represent the sentence but it's very complicated and should have imaginary number of training data.

By the end of this module each word has a corresponding vector representing it in a semantic manner and sure of the question and answer share the same context.

ChatterBot's preprocessors are simple functions that modify the input statement that a chat bot receives before the statement gets processed by the logic adapter.

Here is an example of how to set preprocessors. The preprocessors parameter should be a list string of the import paths to your preprocessors.

```
chatbot = ChatBot(
    'Bob the Bot',
    preprocessors=[
        'chatterbot.preprocessors.clean_whitespace'
    ]
)
```

Preprocessor functions:

ChatterBot comes with several preprocessors build in.

chatterbot.preprocessors.clean_whitespace(*chatbot, statement*) [source]

Remove any consecutive whitespace characters from the statement text.

chatterbot.preprocessors.unescape_html(*chatbot, statement*) [source]

Convert escaped html characters into unescaped html characters. For example: “<b>” becomes “<b>”.

chatterbot.preprocessors.convert_to_ascii(*chatbot, statement*) [source]

Converts unicode characters to ASCII character equivalents. For example: “på fédéral” becomes “pa federal”.

Creating new preprocessors:

It is simple to create your own preprocessors. A preprocessor is just a function with a few requirements.

It must take two parameters, the first is a ChatBot instance, the second is a Statement instance.

It must return a statement instance.

Class 2: The Model The Neural Network:

Motivation:

Deep Neural Networks (DNNs) are powerful models that have achieved excellent performance on difficult learning tasks. Although DNNs work well whenever large labeled training sets are available, they cannot be used to map sequences to sequences. In this model, we present a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure. Our method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector.

The paper presents an LSTM (long short term memory) model for sequence to sequence learning. It consists of 2 LSTM models that are used for encoding and decoding processes. Link to the paper

Our model is a sequence to sequence model, which is a direct implementation of the end to end principle. Our model converses by predicting the next sentence given the previous sentence or sentences in conversation. The strength of our model is that it can be trained end-to-end and thus requires much fewer hand-crafted rules. We find that this straightforward model can generate simple conversations given a large conversational training dataset. Our preliminary suggest that, despite optimizing the wrong objective function, the model is able to extract knowledge from both a domain specific dataset, and from a large, noisy, and general domain dataset of movie subtitles. Our approach makes use of the sequence to sequence (seq2seq) model. The model is based on a recurrent neural network which reads the input sequence one token at a time, and predicts the output sequence, also one token at a time. During training, the true output sequence is given to the model, so learning can be done by backpropagation. The model is trained to maximize the cross entropy of the correct sequence given its context. During inference, given that the true output sequence is not observed, we simply feed the predicted output token as input to predict the next output. This is a "greedy" inference approach. A less greedy approach would be to use beam search, and feed several candidates at the previous step to the next step. The predicted sequence can be selected based on the probability of the sequence. Concretely, suppose that we observe a conversation with two turns: the first person utters "ABC", and another replies "WXYZ". We can use a recurrent neural network, and train to map "ABC" to "WXYZ" as shown in Figure 1.

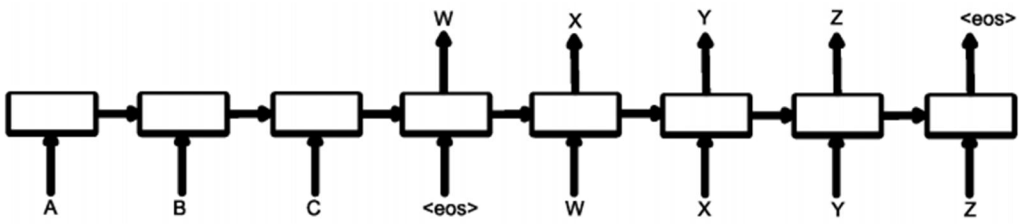
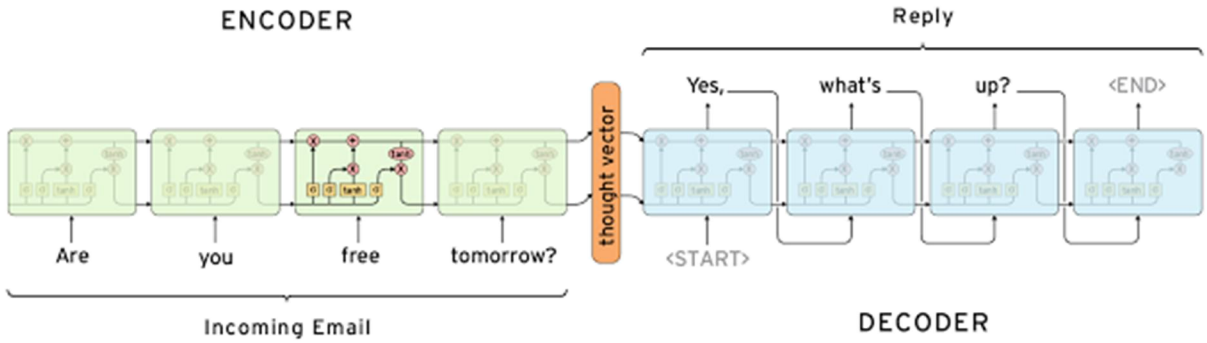


Figure 1. Using the seq2seq framework for modeling conversations.

The hidden state of the model when it receives the end of sequence symbol "<eos>" can be viewed as the thought vector because it stores the information of the sentence, or thought, "ABC". The strength of this model lies in its simplicity and generality limitation of a purely unsupervised model.

The idea beyond the model:

The idea is to use one LSTM to read the input sequence, one time step at a time, to obtain large fixed dimensional vector representation, and then to use another LSTM to extract the output sequence from that vector (fig. 1). The second LSTM is essentially a recurrent neural network language model except that it is conditioned on the input sequence. The LSTM's ability to successfully learn on data with long range temporal dependencies makes it a natural choice for this application due to the considerable time lag between the inputs and their corresponding outputs (fig. 1).



The Recurrent Neural Network (RNN) is a natural generalization of feedforward neural networks to sequences. Given a sequence of inputs (x1, . . . , xT), a standard RNN computes a sequence of outputs (y1, . . . , yT) by iterating the following equation:

$$h_t = \text{sigm}\left(W^{\text{hx}} x_t + W^{\text{hh}} h_{t-1}\right)$$
$$y_t = W^{\text{yh}} h_t$$

The RNN can easily map sequences to sequences whenever the alignment between the inputs and the outputs is known ahead of time. However, it is not clear how to apply an RNN to problems whose input and the output sequences have different lengths with complicated and non-monotonic relationships.

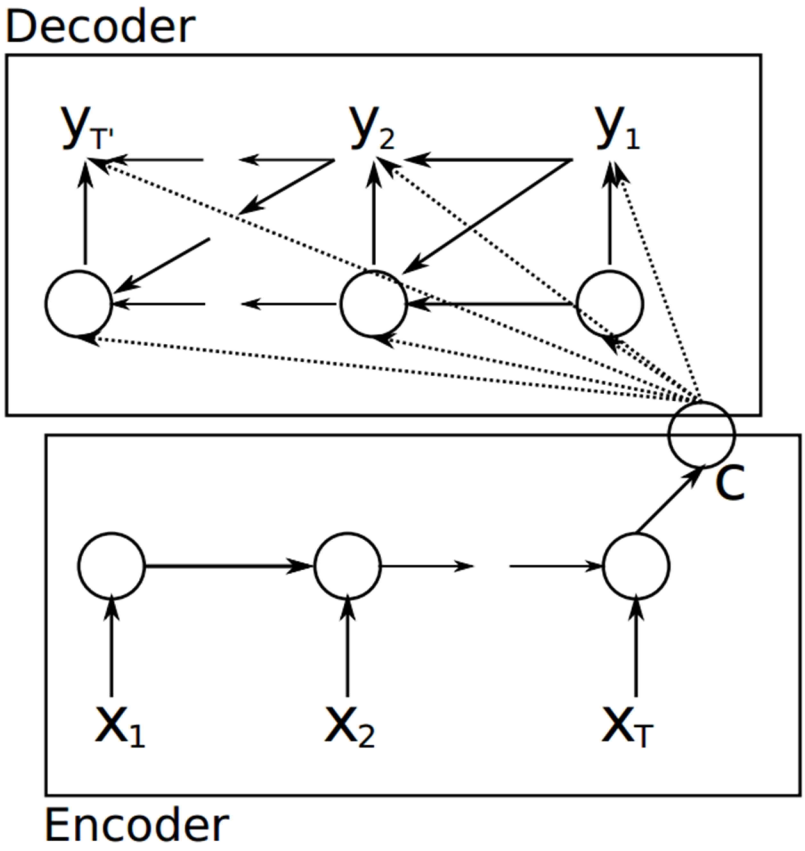


Figure 2: An illustration of the proposed RNN Encoder–Decoder.

The simplest strategy for general sequence learning is to map the input sequence to a fixed sized vector using one RNN, and then to map the vector to the target sequence with another RNN (figure2). While it could work in principle since the RNN is provided with all the relevant information, it would be difficult to train the RNNs due to the resulting long term dependencies (figure 1). However, the Long Short-Term Memory (LSTM) is known to learn problems with long range temporal dependencies, so an LSTM may succeed in this setting. The goal of the LSTM is to estimate the conditional probability p(y1, . . . , yT′ |x1, . . . , xT) where(x1, . . . , xT) is an input sequence and (y1, . . . ,

yT′)is its corresponding output sequence whose length T′ may differ from T . The LSTM computes this conditional probability by first obtaining the fixed dimensional representation v of the input sequence (x1, . . . , xT) given by the last hidden state of the LSTM, and then computing the probability of (y1, . . . , yT′) with a standard LSTM-LM formulation whose initial hidden state is set to the representation v of (x1, . . . , xT):

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1}) \quad (1)$$

In this equation, each p (y_t |v, y_1, . . . , y_{t-1}) distribution is represented with a soft max over all the words in the vocabulary. We use the LSTM formulation from Graves. Note that we require that each sentence ends with a special end-of-sentence symbol "<EOS>", which enables the model to define a distribution over sequences of all possible lengths. The overall scheme is outlined in figure1, where the shown LSTM computes the representation of "A", "B", "C", "<EOS>" and then uses this representation to compute the probability of "W", "X", "Y", "Z", "<EOS>". Our actual models differ from the above description in three important ways. First, we used two different LSTMs: one for the input sequence and another for the output sequence, because doing so increases the number model parameters at negligible computational cost and makes it natural to train the LSTM on multiple language pairs simultaneously. Second, we found that deep LSTMs significantly outperformed shallow LSTMs, so we chose an LSTM with four layers. Third, we found it extremely valuable to reverse the order of the words of the input sentence. So for example, instead of mapping the sentence a, b, c to the sentence α, β, γ, the LSTM is asked to map c, b, a to α, β, γ, where α, β, γ is the translation of a, b, c. This way, a is in close proximity to α, b is fairly close to β, and so on, a fact that makes it easy for SGD to "establish communication" between the input and the output. We found this simple data transformation to greatly improve the performance of the LSTM.

Once the RNN Encoder–Decoder is trained, the model can be used in two ways. One way is to use the model to generate a target sequence given an input sequence. On the other hand, the model can be used to score a given pair of input and output sequences, where the score is simply a probability p (y | x).

The model building begins with assigning each word a long sequence of numbers in the form of a vector. Word vectors are useful because you can calculate the distance between them. The word vector for the word "run" will be pretty close to

the word vector for the word "jog," but both of those word vectors will be far from the vector for "Chicago." This is a big improvement over symbols, where all we can say is that going for a run is not the same as going for a jog, and neither are the same as Chicago.

The word vector for each word has the same dimension. The dimension is usually around 300, and unlike tf-idf document vectors, word vectors are dense, meaning that most values are not 0. To learn the word vectors, the Skip-gram algorithm first initializes each word vector to a random value. It then essentially loops over each word w1 in all of the documents, and for each word w2 around word w1, it pushes the vectors for w1 and w2 closer together, while simultaneously pushing the vector for w1 and the vectors for all other words farther apart.

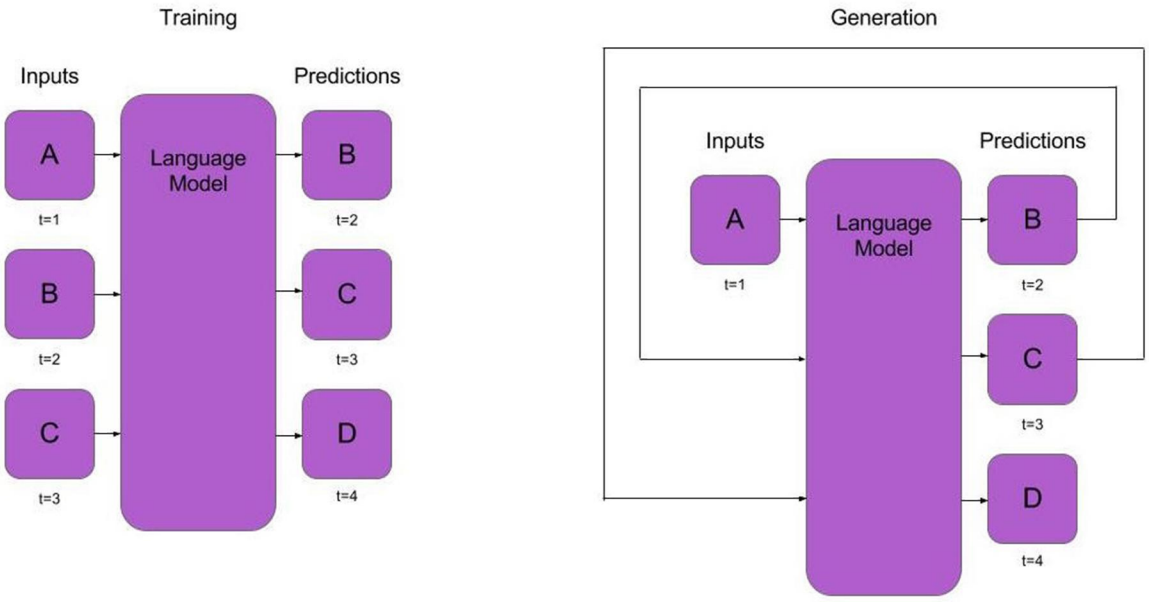
The quote we often see associated with word vectors is "You shall know a word by the company it keeps" by J. R. Firth (1957). This seems to be true, at least to a degree, because word vectors have surprising internal structure. For example, the classic result is that if you take the word vector for the word "Italy" and subtract the word vector for the word "Rome," you get something very similar to what you get when you subtract the word vector for the word "Paris" from the word vector for the word "France." This internal structure of word vectors is impressive, but these vectors are not grounded in experience in the world—they are only grounded in being around other words. As we saw previously, people only express what the reader does not know, so the amount of world knowledge that can be contained in these vectors is limited.

The Model Input:

Just as we can encode words into vectors, we can encode whole sentences into vectors. This encoding is done using a recurrent neural network (RNN). An RNN takes a vector representing its last state and a word vector representing the next word in a sentence, and it produces a new vector representing its new state. It can keep doing this until the end of the sentence, and the last state represents the encoding of the entire sentence.

A sentence encoded into a vector using an encoder RNN can then be decoded into a different sentence. This decoding uses another RNN and goes in reverse. This decoder RNN begins with its state vector being the last state vector of the encoder RNN, and it produces the first word of the new sentence. The decoder RNN then takes that produced word and the RNN vector itself as input and returns a new RNN vector, which allows it to produce the next word. This process can continue until a special stop symbol is produced, such as a period.

These encoder-decoder (sequence-to-sequence) models are trained on a corpus consisting of source sentences and their associated target sentences, these sentences are run through the model until it learns the underlying patterns.



This encoder-decoder models work on many kinds of sequences, but this generality highlights their limitation for use as language understanding agents, such as chatbots. Noam Chomsky proposed that the human brain contains a specialized universal grammar that allows us to learn our native language. In conversations, sequences of words are just the tip of the meaning iceberg, and it is unlikely that a general method run over the surface words in communication could capture the depth of language. Language allows for infinite combinations of concepts, and any training set, no matter how large, will represent only a finite subset.

Beyond the fact that the sequence-to-sequence models are too general to fully capture language, you may wonder how a fixed-length vector can store a growing amount of information as the model moves from word to word. This is a real problem, and it was partially solved by adding attention to the model. During decoding, for example when doing language translation, before outputting the next word of the target sentence, the model can look back over all of the encoded states of the source sentence to help it determine what the next word should be. The model learns what kinds of information it needs in different situations. It treats the encodings of the input sentence as a kind of memory.

This attention mechanism can be generalized to enable computers to answer questions based on text. The question itself can be converted into a vector, and instead of looking back through words in the source sentence, the neural network can look at encoded versions of the facts it has seen, and it can find the best facts that will help it answer the current question.

Benefits of the model:

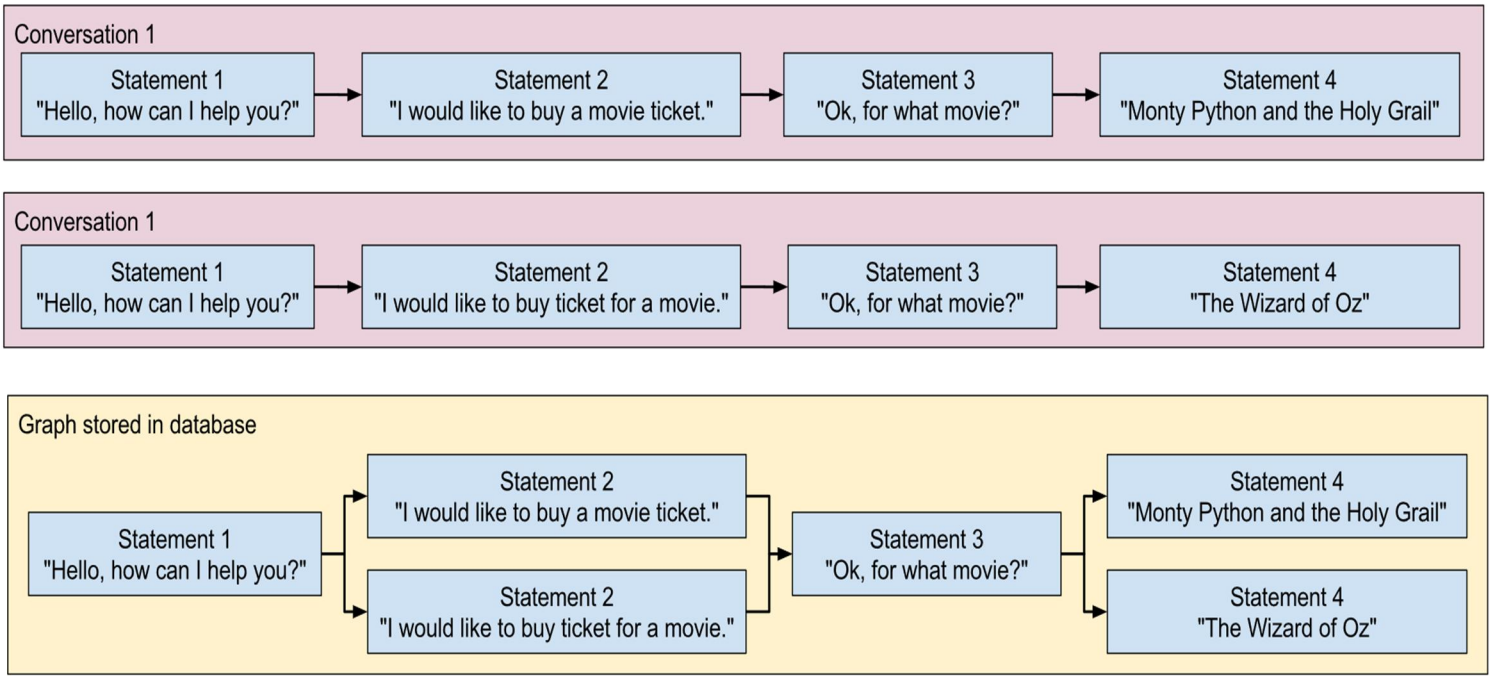
- A new way to map an input to an output that are commonly used in Machine Translation and Question Answering tasks
- Another evidence that shows the advantage of reading the input in reverse order instead of the original order
- A parallelized model that works efficiently in training phase

Model Specifications:

- Two LSTMs that are connected together, each is two layers.
- The first LSTM target is to encode the given input to a fixed-size vector that are fed to the second LSTM
- The second LSTM target is to decode the given vector from the previous LSTM layer and produce a sequence that are closely related to the input
- The first LSTM reading the input in a reverse order. Experiments showed that this boosted up the accuracy of the training
- The first LSTM keeps reading the input words till it finds token < EOS >
- The seconds LSTM produces words till it finds an output of < EOS > which means that the LSTM shall stop produce words
- The optimization algorithm used is the SGD

Class 3: Model 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.

Several training classes come built-in with ChaterBot. 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 pre-loaded training data.

You can also create your own training class. This is recommend if you wish to train your bot with data you have stored in a format that is not already supported by one of the pre-built classes listed below.

Setting the training class:

ChatterBot comes with training classes built in, or you can create your own if needed. To use a training class you must import it and pass it to the *set_trainer()* method before calling *train()*.

Training classes:

Training via list data:

```
chatterbot.trainers.ListTrainer(storage, **kwargs)    [source]
```

Allows a chat bot to be trained using a list of strings where the list represents a conversation.

For the training, process, you will need to pass in a list of statements where the order of each statement is based on it's placement in a given conversation.

For example, if you were to run bot of the following training calls, then the resulting chatterbot would respond to both statements of "Hi there!" and "Greetings!" by saying "Hello".

```
from chatterbot.trainers import ListTrainer

chatterbot = ChatBot("Training Example")
chatterbot.set_trainer(ListTrainer)

chatterbot.train([
    "Hi there!",
    "Hello",
])

chatterbot.train([
    "Greetings!",
    "Hello",
])
```

You can also provide longer lists of training conversations. This will establish each item in the list as a possible response to it's predecessor in the list.

```
chatterbot.train([
    "How are you?",
    "I am good.",
    "That is good to hear.",
    "Thank you",
    "You are welcome.",
])
```

Training with corpus data

chatterbot.trainers.ChatterBotCorpusTrainer(*storage, **kwargs*) [source]

Allows the chat bot to be trained using data from the ChatterBot dialog corpus.

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.

```
from chatterbot.trainers import ChatterBotCorpusTrainer

chatterbot = ChatBot("Training Example")
chatterbot.set_trainer(ChatterBotCorpusTrainer)

chatterbot.train(
    "chatterbot.corpus.english"
)
```

Specifying corpus scope

It is also possible to import individual subsets of ChatterBot's at once. For example, if you only wish to train based on the english greetings and conversations corpora then you would simply specify them.

```
chatterbot.train(
    "chatterbot.corpus.english.greetings",
    "chatterbot.corpus.english.conversations"
)
```

You can also specify file paths to corpus files or directories of corpus files when calling the train method.

```
chatterbot.train(
    "./data/greetings_corpus/custom.corpus.json",
    "./data/my_corpus/"
)
```

Training with the Twitter API

chatterbot.trainers.TwitterTrainer(*storage, **kwargs*) [source]

Purpose: Allows the chat bot to be trained using data gathered from Twitter.

Parameters: **random_seed_word** – The seed word to be used to get random tweets from the Twitter API. This parameter is optional. By default it is the word 'random'.

Create an new app using your twitter account. Once created, it will provide you with the following credentials that are required to work with the Twitter API.

Parameter	Description
twitter_consumer_key	Consumer key of twitter app.
twitter_consumer_secret	Consumer secret of twitter app.
twitter_access_token_key	Access token key of twitter app.
twitter_access_token_secret	Access token secret of twitter app.

Twitter training example:

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot
from settings import TWITTER
import logging


'''
This example demonstrates how you can train your chat bot
using data from Twitter.

To use this example, create a new file called settings.py.
In settings.py define the following:

TWITTER = {
    "CONSUMER_KEY": "my-twitter-consumer-key",
    "CONSUMER_SECRET": "my-twitter-consumer-secret",
    "ACCESS_TOKEN": "my-access-token",
    "ACCESS_TOKEN_SECRET": "my-access-token-secret"
}
'''

# Comment out the following line to disable verbose logging
logging.basicConfig(level=logging.INFO)

chatbot = ChatBot("TwitterBot",
    logic_adapters=[
        "chatterbot.logic.BestMatch"
    ],
    input_adapter="chatterbot.input.TerminalAdapter",
    output_adapter="chatterbot.output.TerminalAdapter",
    database="./twitter-database.db",
    twitter_consumer_key=TWITTER["CONSUMER_KEY"],
    twitter_consumer_secret=TWITTER["CONSUMER_SECRET"],
    twitter_access_token_key=TWITTER["ACCESS_TOKEN"],
    twitter_access_token_secret=TWITTER["ACCESS_TOKEN_SECRET"],
    trainer="chatterbot.trainers.TwitterTrainer"
)

chatbot.train()

chatbot.logger.info('Trained database generated successfully!')
```

Training with the Ubuntu dialog corpus

chatterbot.trainers.UbuntuCorpusTrainer(*storage*, ***kwargs*) [source]

Allow chatbots to be trained with the data from the Ubuntu Dialog Corpus.

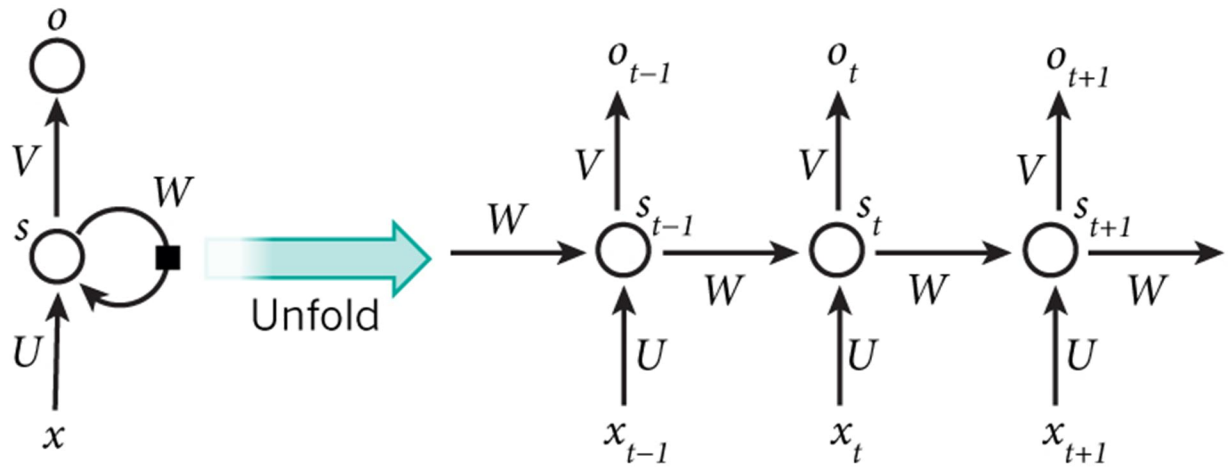
This training class makes it possible to train your chat bot using the Ubuntu dialog corpus. Because of the file size of the Ubuntu dialog corpus, the download and training process may take a considerable amount of time.

This training class will handle the process of downloading the compressed corpus file and extracting it. If the file has already been downloaded, it will not be downloaded again. If the file is already extracted, it will not be extracted again.

Creating a new training class

You can create a new trainer to train your chat bot from your own data files. You may choose to do this if you want to train your chat bot from a data source in a format that is not directly supported by ChatterBot.

Your custom trainer should inherit *chatterbot.trainers.Trainer* class. Your trainer will need to have a method named *train*, that can take any parameters you choose. **Train the model to be able to be tested or used.**



Class 4: ChatterBot:

The main class 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.

A conversational dialog chat bot.

class **chatterbot.ChatBot**(*name, **kwargs*) [source]

Parameters:

- **name** (*str*) – A name is the only required parameter for the ChatBot class.
- **storage_adapter** (*str*) – The import path to a storage adapter class.
- **logic_adapters** (*list*) – A list of string paths to each logic adapter the bot uses.
- **input_adapter** (*str*) – The import path to an input adapter class.
- **output_adapter** (*str*) – The import path to an output adapter class.
- **trainer** (*str*) – The import path to the training class to be used with the chat bot.
- **filters** (*list*) – A list of import paths to filter classes to be used by the chat bot.
- **logger** (*logging.Logger*) – A `Logger` object.

classmethod **from_config**(*config_file_path*) [source]

Create a new ChatBot instance from a JSON config file.

generate_response(*input_statement, session_id*) [source]

Return a response based on a given input statement.

get_response(*input_item*, *session_id=None*) [source]

Return the bot's response based on the input.

Parameters: **input_item** – An input value.
Returns: A response to the input.
Return type: Statement

initialize() [source]

Do any work that needs to be done before the responses can be returned.

learn_response(*statement*, *previous_statement*) [source]

Learn that the statement provided is a valid response.

set_trainer(*training_class*, ***kwargs*) [source]

Set the module used to train the chatbot.

Parameters:

- **training_class** (*Trainer*) – The training class to use for the chat bot.
- ****kwargs** – Any parameters that should be passed to the training class.

train

Proxy method to the chat bot's trainer class.

Example chat bot parameters

```
ChatBot(
    'Northumberland',
    storage_adapter='my.storage.AdapterClass',
    logic_adapters=[
        'my.logic.AdapterClass1',
        'my.logic.AdapterClass2'
    ],
    input_adapter='my.input.AdapterClass',
    output_adapter='my.output.AdapterClass',
    trainer='my.trainer.TrainerClass',
    filters=[
        'my.filter.FilterClass1',
        'my.filter.FilterClass2'
    ],
    logger=custom_logger
)
```

Example expanded chat bot parameters

It is also possible to pass parameters directly to individual adapters. To do this, you must use a dictionary that contains a key called `import_path` which specifies the import path to the adapter class.

```
ChatBot(
    'Leander Jenkins',
    storage_adapter={
        'import_path': 'my.storage.AdapterClass',
        'database_name': 'my-database'
    },
    logic_adapters=[
        {
            'import_path': 'my.logic.AdapterClass1',
            'statement_comparison_function': 'chatterbot.comparisons.levenshtein_distance'
            'response_selection_method': 'chatterbot.response_selection.get_first_response'
        },
        {
            'import_path': 'my.logic.AdapterClass2',
            'statement_comparison_function': 'my.custom.comparison_function'
            'response_selection_method': 'my.custom.selection_method'
        }
    ],
    input_adapter={
        'import_path': 'my.input.AdapterClass',
        'api_key': '0000-1111-2222-3333-DDDD'
    },
    output_adapter={
        'import_path': 'my.output.AdapterClass',
        'api_key': '0000-1111-2222-3333-DDDD'
    }
)
```

Enable logging:

ChatterBot has built in logging. You can enable ChatterBot's logging by setting the logging level at the top of your python code.

```
import logging

logging.basicConfig(level=logging.INFO)

ChatBot(
    # ...
)
```

Using a custom logger:

You can choose to use your own custom logging class with your chat bot. This can be useful when testing and debugging your code.

```
import logging

custom_logger = logging.getLogger(__name__)

ChatBot(
    # ...
    logger=custom_logger
)
```

Adapters

ChatterBot uses adapter modules to control the behavior of specific types of tasks. There are four distinct types of adapters that ChatterBot uses, these are storage adapters, input adapters, output adapters and logic adapters.

Adapters types:

1. Storage adapters - Provide an interface for ChatterBot to connect to various storage systems such as MongoDB or local file storage.
2. Input adapters - Provide methods that allow ChatterBot to get input from a defined data source.
3. Output adapters - Provide methods that allow ChatterBot to return a response to a defined data source.
4. Logic adapters - Define the logic that ChatterBot uses to respond to input it receives.

Accessing the chatbot instance:

When ChatterBot initializes each adapter, it sets an attribute named chatbot. The chatbot variable makes it possible for each adapter to have access to all of the other adapters being used. Suppose two input and output adapters need to share some information or perhaps you want to give your logic adapter direct access to the storage adapter. These are just a few cases where this functionality is useful.

Each adapter can be accessed on the chatbot object from within an adapter by referencing *self.chatbot*. Then, self.chatbot.storage refers to the storage adapter, self.chatbot.input refers to the input adapter, self.chatbot.output refers to the current output adapter, and self.chatbot.logic refers to the logic adapters.

Adapter defaults:

By default, ChatterBot uses the *JsonFileStorageAdapter* adapter for storage, the *BestMatch* for logic, the *VariableInputTypeAdapter* for input and the *OutputAdapter* for output.

Each adapter can be set by passing in the dot-notated import path to the constructor as shown.

```
bot = ChatBot(
    "Elsie",
    storage_adapter="chatterbot.storage.JsonFileStorageAdapter",
    input_adapter="chatterbot.input.VariableInputTypeAdapter",
    output_adapter="chatterbot.output.OutputAdapter",
    logic_adapters=[
        "chatterbot.logic.BestMatch"
    ],
)
```

Testing:

Measuring the system's accuracy and its level of humanness, using the Granular Conversation Test **(GCT), that tests a reply given back by the computer in 4 areas:**

- Semantics: "Does the response make sense by itself?"
- Structure: "Is it understandable and does it follows a structure a human would use?"
- Context: "Does the response make sense in the context?"
- Feeling: "Does the response feel human?"

Each component receives a score of 0 or 1 ("no" or "yes," respectively). The result of the test is the sum of the ones, and then we will calculate the average of each of the 4 components.

4- User Manual

Presentation:

This work tries to reproduce the results of A Neural Conversational Model (aka the Google chatbot). It uses a RNN (seq2seq model) for sentence predictions. It is done using python and TensorFlow.

For now, DeepQA support the following dialog corpus:

- Cornell Movie Dialogs corpus (default). Already included when cloning the repository.
- OpenSubtitles. Much bigger corpus (but also noisier). To use it, use the flag --corpus opensubs.
- Supreme Court Conversation Data. Available using --corpus scotus.
- Ubuntu Dialogue Corpus. Available using --corpus ubuntu.
- Your own data, by using a simple custom conversation format.

To speedup the training, it's also possible to use pre-trained word embeddings.

Installation:

The program requires the following dependencies (easy to install using pip):

- python 3.5
- tensorflow (tested with v1.0)
- numpy
- CUDA (for using GPU)
- nltk (natural language toolkit for tokenized the sentences)
- tqdm (for the nice progression bars)

You might also need to download additional data to make nltk work.

```
python3 -m nltk.downloader punkt
```

The Cornell dataset is already included. For the other datasets, look at the readme files into their respective folders (inside data/).

The web interface requires some additional packages:

- django (tested with 1.10)
- channels
- Redis (see here)
- asgi_redis (at least 1.0)

Running:

Chatbot:

To train the model, simply run main.py. Once trained, you can test the results with main.py --test (results generated in 'save/model/samples_predictions.txt') or main.py --test interactive (more fun).

Here are some flags which could be useful. For more help and options, use python main.py -h:

- --modelTag <name>: allow to give a name to the current model to differentiate between them when testing/training.
- --keepAll: use this flag when training if when testing, you want to see the predictions at different steps (it can be interesting to see the program changes its name and age as the training progress). Warning: It can quickly take a lot of storage space if you don't increase the --saveEvery option.
- --filterVocab 20 or --vocabularySize 30000: Limit the vocabulary size to and optimize the performances and memory usage. Replace the words used less than 20 times by the <unknown> token and set a maximum vocabulary size.
- --verbose: when testing, will print the sentences as they are computed.
- --playDataset: show some dialogue samples from the dataset (can be use conjointly with --createDataset if this is the only action you want to perform).

To visualize the computational graph and the cost with TensorBoard, just run tensorboard --logdir save/.

By default, the network architecture is a standard encoder/decoder with two LSTM layers (hidden size of 256) and an embedding size for the vocabulary of 32. The network is trained using ADAM. The maximum sentence length is set to 10 words, but can be increased.

Web interface:

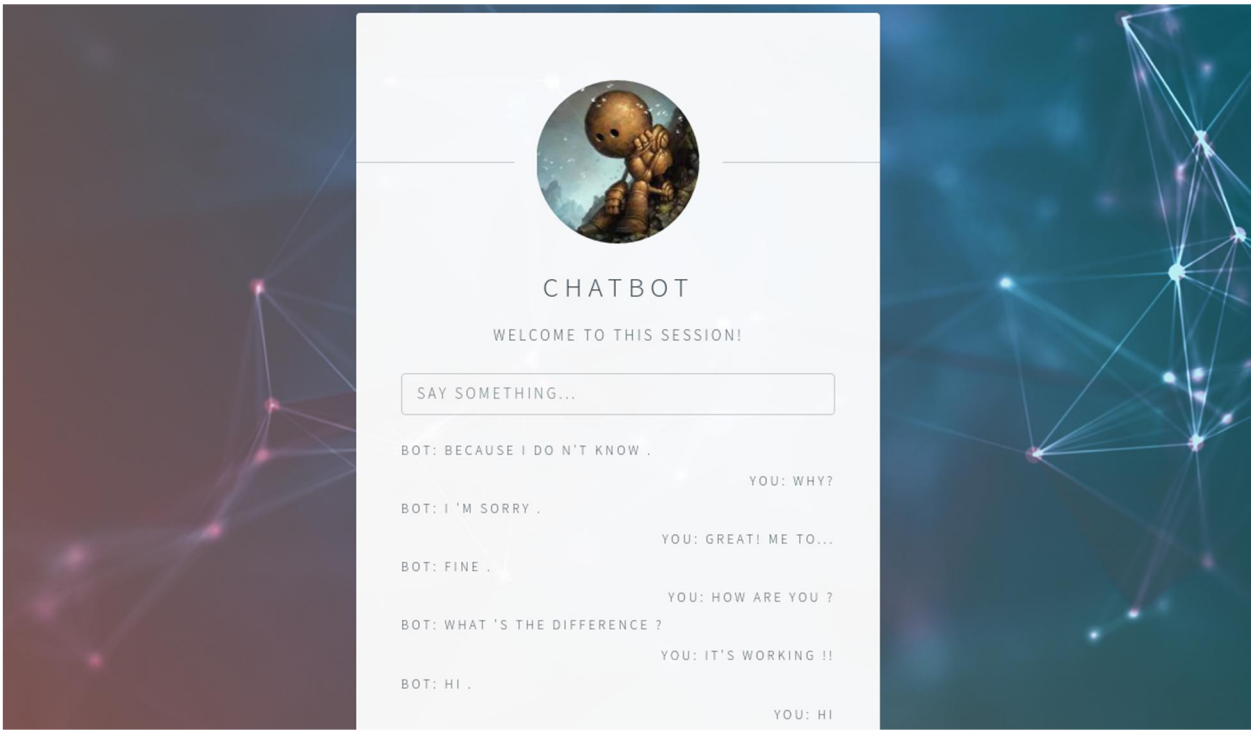
Once trained, it's possible to chat with it using a more user friendly interface. The server will look at the model present on save/model-server/model.ckpt. The first time you want to use it, you'll need to configure it with:

```
export CHATBOT_SECRET_KEY="my-secret-key"
cd chatbot_website/
python manage.py makemigrations
python manage.py migrate
```

Then, to launch the server locally, use the following commands:

```
cd chatbot_website/
redis-server &  # Launch Redis in background
python manage.py runserver
```

After launch, the interface should be available on http://localhost:8000/. If you want to deploy the program on a server, use python manage.py runserver 0.0.0.0 instead. More info here.



Pretrained model:

You can find a pre-trained model here, trained of the default corpus. To use it:

1. Extract the zip file inside DeepQA/save/
2. Copy the preprocessed dataset from save/model-pretrainedv2/dataset-cornell-old-lenght10-filter0-vocabSize0.pkl to data/samples/.
3. Run ./main.py --modelTag pretrainedv2 --test interactive.

If you have a high-end GPU, don't hesitate to play with the hyper-parameters/corpus to train a better model. From my experiments, it seems that the learning rate and dropout rate have the most impact on the results. Also if you want to share your models, don't hesitate to contact me and I'll add it here.

5- Conclusion and Future Work

6.1 Conclusion

Surprisingly, it's possible to get some results after only 1 or 2 hours of training (on a GeForce GT 740M), by drastically reducing the input sequence to 5 words and output to 3 (plus the `go` and `eos` tokens) and by using a small embedding size (something like 25). Since then I modified the code and now the output length has to match the input one but you can still reproduce the original results, using that version. Of course, the network won't be really chatty:

```
Q: Hi
A: Hi.

Q: What is your name ?
A: Laura.

Q: What does that mean ?
A: I dunno.

Q: How old are you ?
A: thirty-five.

Q: Will Google hire me ?
A: No.

Q: Tell me the alphabet
A: Fuck you.

Q: That's not nice
A: Yeah.
```

Here is some case where it fails:

```
Q: Who is Laura ?
A: My brother.

Q: Say 'goodbye'
A: Alright.

Q: What is cooking ?
A: A channel.

Q: Can you say no ?
A: No.

Q: Two plus two
A: Manny...
```

With longer sentences, the network is much slower to train. After 250000 iterations with maxLength=10, the network still mainly gives evasive answers (by answering most of the open ended questions by 'I don't know') but it seems to have integrated some basic semantic:

```
Q: When are the fireworks ?
A: Tomorrow.

Q: What color ?
A: Gray.

Q: Which apple ?
A: That's the first one.

Q: What time is it ?
A: Eleven o'clock.

Q: Who is Skywalker ?
A: She's a freelance bio-exorcist.

Q: Who is John ?
A: The old man.

Q: Who is Jenny ?
A: The man who killed herself.

Q: When were you born ?
A: Soon.
```

As shown on the last questions, the network still does some confusion between the genders and the present/future. I also tried some deeper philosophical questions with more or less success.

```
Q: What is the purpose of being intelligent ?
A: The man are obsolete.

Q: What is immoral ?
A: The truth.

Q: What is morality ?
A: The worst.

Q: What is the purpose of existence ?
A: It 's private.

Q: What happens if machines can think ?
A: I think I'll just lie with you.
```

The model I trained is quite limited by the embedding size I put and by its size, and by the training corpus size. Its 'thought vector' is also probably too small to answer the kind of following:

```
Q: My name is David. What is my name ?
A: Debbie.

Q: My name is John. What is my name ?
A: Carl Smith.

Q: How many legs does a cat have ?
A: Four.

Q: How many legs does a spider have ?
A: Four.

Q: Do you prefer football or soccer ?
A: Whiskey.
```

It also seems to overfit as sometimes it will just pop out sentences from its training set unrelated to the question. Adding some dropout should reduce the issue.

```
Q: I play tennis. What do I play ?
A: I'm going to get laid.

Q: What color is the sky ?
A: One of the deputies is everywhere.
```

6.2 Future Work:

In addition to trying larger/deeper model, there are a lot of small improvements which could be tested. Don't hesitate to send a pull request if you implement one of those. Here are some ideas:

- For now, the predictions are deterministic (the network just take the most likely output) so when answering a question, the network will always give the same answer. By adding a sampling mechanism, the network could give more diverse (and maybe more interesting) answers. The easiest way to do that is to sample the next predicted word from the SoftMax probability distribution. By combining that with the **loop_function** argument of **tf.nn.seq2seq.rnn_decoder**, it shouldn't be too difficult to add. After that, it should be possible to play with the SoftMax temperature to get more conservative or exotic predictions.
- Adding attention could potentially improve the predictions, especially for longer sentences. It should be straightforward by replacing **embedding_rnn_seq2seq** by **embedding_attention_seq2seq** on **model.py**.
- Having more data usually don't hurt. Training on a bigger corpus should be beneficial. Reddit comments dataset seems the biggest for now (and is too big for this program to support it). Another trick to artificially increase the

dataset size when creating the corpus could be to split the sentences of each training sample (ex: from the sample:
Q:Sentence 1. Sentence 2. => A:Sentence X. Sentence Y. we could generate 3 new samples:
Q:Sentence 1. Sentence 2. => A:Sentence X,
Q:Sentence 2. => A:Sentence X. Sentence Y and
Q:Sentence 2. => A:Sentence X
Warning: other combinations like
Q:Sentence 1. => A:Sentence X. won't work because it would break the transition **2 => X** which links the question to the answer)

- The testing curve should really be monitored as done in my other music generation project. This would greatly help to see the impact of dropout on overfitting. For now it's just done empirically by manually checking the testing prediction at different training steps.
- For now, the questions are independent from each other. To link questions together, a straightforward way would be to feed all previous questions and answer to the encoder before giving the answer. Some caching could be done on the final encoder stated to avoid recomputing it each time. To improve the accuracy, the network should be retrain on entire dialogues instead of just individual QA. Also when feeding the previous dialogue to the encoder, new tokens **<Q>** and **<A>** could be added so the encoder knows when the interlocutor is changing. I'm not sure though that the simple seq2seq model would be sufficient to capture long term dependencies between sentences. Adding a bucket system to group similar input lengths together could greatly improve training speed.
- Build a bot that produces as relevant as possible replies to different kind of questions.
- Get up to speed with the latest natural language processing and machine learning techniques.
- Build a bot that recognizes different users' personalities, and produce replies based on this property.
- Make the bot as humane as possible.

References

- Papers:

[1] Kaisheng Yao, Geoffrey Zweig, Baolin Peng (Oct 2015): Attention with Intention for a Neural Network Conversation Model.
[2] Jiwei Li, Michel Galley, Chris Brockett, Georgios P. Spithourakis, Jianfeng Gao, Bill Dolan (Jun 2016): A Persona-Based Neural Conversation Model.
[3] Oriol Vinyals, Quoc V. Le (Jun 2015): A Neural Conversational Model.
[4] Iulian V. Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, Joelle Pineau (Apr 2016): Building End-To-End Dialogue Systems
Using Generative Hierarchical Neural Network Models.
[5] Roy Chan (2016): Designer Chatbots for Lonely People.
[6] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean (Sep 2013): Efficient Estimation of Word Representations in Vector Space.
[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio (Jun 2014):
Generative Adversarial Nets.
[8] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio (Sep 2014):
Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation.
[9] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio (May 2016): Nerual Machine Translation By Jointly Learning To Align And Translate
[10] Ilya Sutskever, Oriol Vinyals, Quoc V. Le (Dec 2014): Sequence to Sequence Learning with Neural Networks.
[11] Bonnie Chantarotwong (Oct 2006): The Learning Chatbot.

- Online Blogs:

[1] http://www.wildml.com/2016/04/deep-learning-for-chatbots-part-1-introduction/
[2] http://www.wildml.com/2016/07/deep-learning-for-chatbots-2-retrieval-based-model-tensorflow/
[3] https://www.quora.com/I-want-to-create-a-machine-learning-based-chatbot-from-A-to-Z-from-where-can-I-start
[4] https://www.tensorflow.org/tutorials/seq2seq
[5] https://deeplearning4j.org/lstm
[6] https://en.wikipedia.org/wiki/Word2vec
[7] https://en.wikipedia.org/wiki/Recurrent_neural_network
[8] https://en.wikipedia.org/wiki/Turing_test
[9] https://ahmedhanibrahim.wordpress.com/2016/10/09/another-lstm-tutorial/
[10] https://ahmedhanibrahim.wordpress.com/2017/04/27/thesis-tutorials-ii-understanding-word2vec-for-word-embedding-ii/
[11] http://accord-framework.net/samples.html
[12] https://indico.io/blog/sequence-modeling-neuralnets-part1/
[13] http://colah.github.io/posts/2015-08-Understanding-LSTMs/
[14] http://karpathy.github.io/2015/05/21/rnn-effectiveness/
[15] https://blog.drift.com/all-about-chatbots/