Chatbot Installation Instruction

This section explain how IntelliBot can be installed trained.

Environment Setup:

To develop IntelliBot, this study used Anaconda environment with Python 3.6 and PyCharm as IDE. Upon installing these two software, please make sure you have set python path into environment variable PYTHONPATH. It needs to point to the project root directory, in which you have chatbot, Data, and webui folder. If you are running in PyCharm, it will create that for you. But if you run any python scripts in a command line, you have to have that environment variable, otherwise, you get module import errors.

Download Datasets:

IntelliBot were trained with publicly available various datasets as follows. Please download these datasets and extract into your project's 'data' folder.

- Cornell Movie Corpus
  http://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html
- Reddit Dataset
  https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/
- Pre-trained dataset
  https://drive.google.com/file/d/1mVWFScBHFeA7oVxQzWb8QbKfTi3TToUr/view
- Install from Terminal:

  python -m nltk.downloader punkt

  python -m nltk.downloader stopwords

Tools and Libraries Requirements:

The following python libraries must be installed from anaconda terminal using the following command:

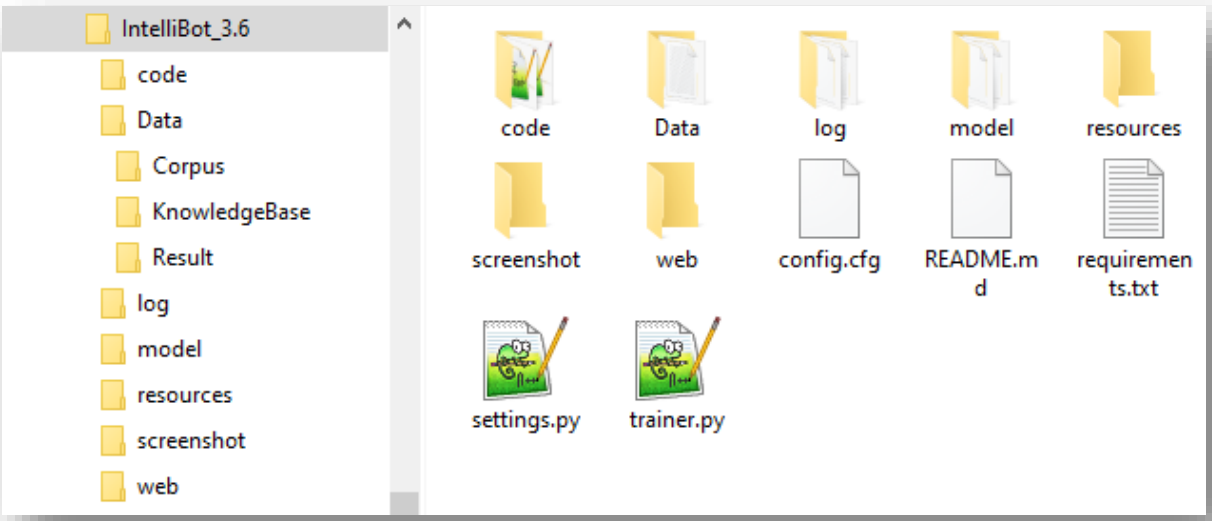
pip install -r requirements.txt

tensorflow 1.14	aiml	lxml	beautifulsoup4
numpy	mysql-connector-python	wikipedia	flask

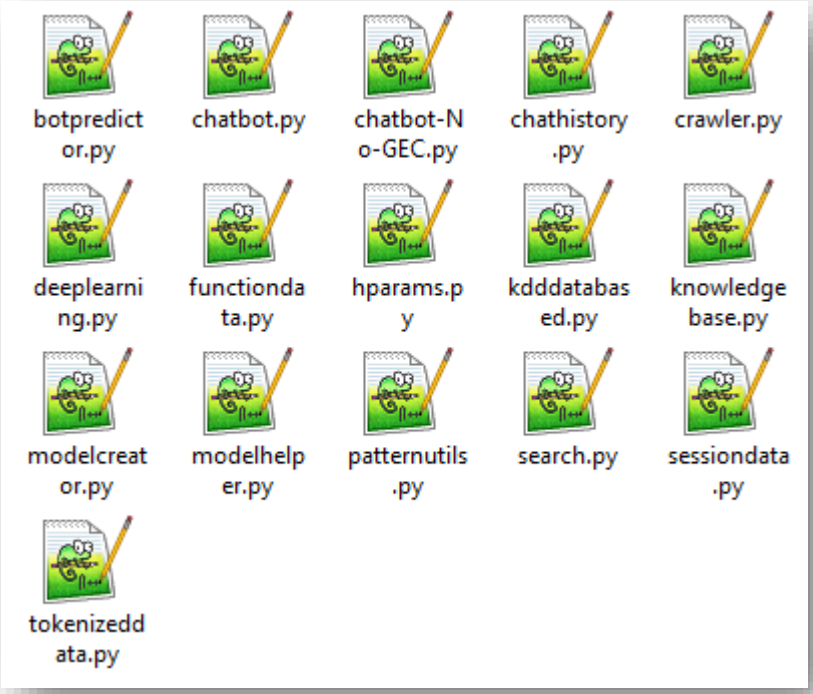
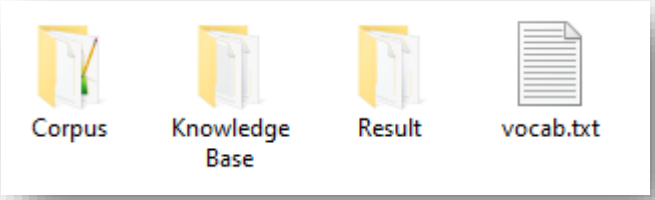
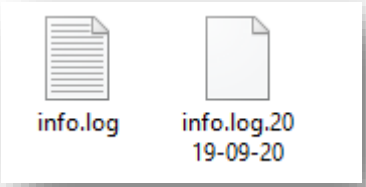
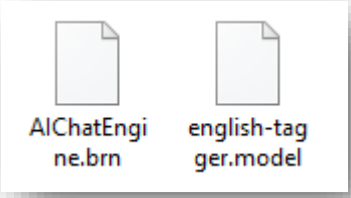
wordsegment	google-cloud-core	configparser	requests
nltk	tqdm	future	networkx
stanfordcorenlp	django	pyttsx3	pyaudio
scikit-learn	colorama	scipy	h5py
tflearn	language-check		

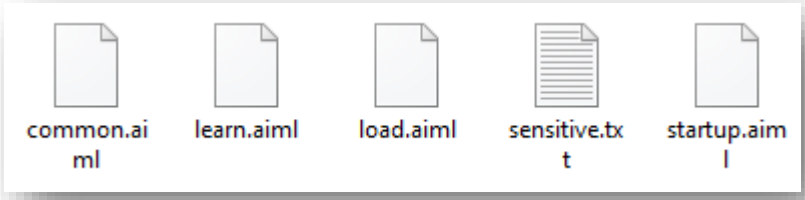
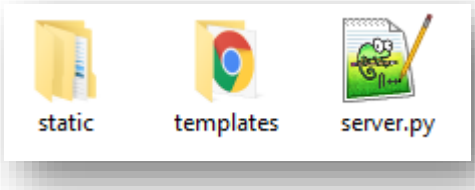
Directory Structure:

The IntelliBot project has seven main folders and their structure shown as follows:



Top Level Folders	Structures

Code	
Data	
Log	
Model	

Resources	
Web	

Hyperparameters setting:

Highly GPU is recommended for the training as it can be very time-consuming. You can adjust the *batch_size parameter* in *hparams.json* file accordingly to make full use of the memory. You will be able to see the training results under *Data/Result/* folder. Make sure the following 2 files exist as all these will be required for testing and prediction (the .meta file is optional as the inference model will be created independently):

1. *basic.data-00000-of-00001*
2. *basic.index*

Execute Training:

Training is straightforward. Remember to create a folder named Result under the Data folder first. Then run the following commands:

cd IntelliBot_3.6

python trainer.py

During the training, I really suggest you to try playing with a parameter (*colocate_gradients_with_ops*) in function *tf.gradients*. You can find a line like this in *modelcreator.py* : *gradients = tf.gradients(self.train_loss, params)*. Set *colocate_gradients_with_ops=True* (adding it) and run the training for at least one epoch, note down the time, and then set it to False (or just remove it) and run the training for at least one epoch and see if the times required for one epoch are significantly different

Execute the Server:

IntelliBot runs on two servers: NLP server and application server.

- **Running NLP Server:**

  Please download *standfordCoreNLP* from the following link and extract it:

  https://stanfordnlp.github.io/CoreNLP/

  Then, go to your '*standford-corenlp*' folder and run the following command:

  java -mx4g -cp "" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000 -timeout 15000*

- **Running Chat Server:**

  cd IntelliBot_3.6/web/

  python server.py

  open your browser and write: http://127.0.0.1:5000/

Testing/ Inference:

For testing and prediction, we provide a simple command interface and a web-based interface. Note that *vocab.txt* file (and files in KnowledgeBase, for this chatbot) is also required for inference. In order to quickly check how the trained model performs, use the following command interface:

cd IntelliBot_3.6/web/

python server.py

Source code:

settings.py

```
import os
PROJECT_ROOT = os.path.abspath(os.path.dirname(__file__))

```

trainer.py

```
import math
import os
import time
import colorama
import tensorflow as tf
import sys

from chatbot.tokenizeddata import TokenizedData
from chatbot.modelcreator import ModelCreator

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
colorama.init()
```

trainer.py

```python

# This class uses for Train the Chatbot
class BotTrainer(object):
    def __init__(self, corpus_dir):
        self.graph = tf.Graph()
        with self.graph.as_default():
            tokenized_data = TokenizedData(corpus_dir=corpus_dir)
            self.hparams = tokenized_data.hparams
            self.train_batch = tokenized_data.get_training_batch()
            self.model = ModelCreator(training=True, tokenized_data=tokenized_data,
                                      batch_input=self.train_batch)

    def train(self, result_dir, target=""):
        """Train a seq2seq model."""
        # Summary writer
        summary_name = "train_log"
        summary_writer = tf.summary.FileWriter(os.path.join(result_dir, summary_name), s

        log_device_placement = self.hparams.log_device_placement
        num_epochs = self.hparams.num_epochs

        config_proto = tf.ConfigProto(log_device_placement=log_device_placement, allow_s
        config_proto.gpu_options.allow_growth = True

        with tf.Session(target=target, config=config_proto, graph=self.graph) as sess:
            sess.run(tf.global_variables_initializer())
            sess.run(tf.tables_initializer())
            global_step = self.model.global_step.eval(session=sess)

            # Initialize all of the iterators
            sess.run(self.train_batch.initializer)

            # Initialize the statistic variables
            ckpt_loss, ckpt_predict_count = 0.0, 0.0
            train_perp, last_record_perp = 2000.0, 2.0
            train_epoch = 1

            print(colorama.Fore.GREEN + '\n\nTraining started ...{} '+format(time.strft:
            print(f'Total Epochs  : {num_epochs}')
            print(f'Current Epochs: {train_epoch}')

            learning_rate = self._get_learning_rate(train_perp)
            print(f'Learning rate : {learning_rate}')
            print(colorama.Fore.RESET)

            epoch_start_time = time.time()
            while train_epoch <= num_epochs:
                # Each run of this while loop is a training step, multiple time/steps will trig
                # the train_epoch to be increased.
                print(colorama.Fore.YELLOW+f'Training epoch {train_epoch}'+colorama.Fore.RESET)

                try:
                    step_result = self.model.train_step(sess, learning_rate=learning_rate)
                    (_, step_loss, step_predict_count, step_summary, global_step,
                     step_word_count, batch_size) = step_result

                    # Write step summary.
                    summary_writer.add_summary(step_summary, global_step)

                    # update statistics
                    ckpt_loss += (step_loss * batch_size)
                    ckpt_predict_count += step_predict_count
                    print(f'ckpt_predict_count : {ckpt_predict_count}')
```

trainer.py

```python
                except tf.errors.OutOfRangeError:
                    # Finished going through the training dataset. Go to next epoch.
                    train_epoch += 1
                    mean_loss = ckpt_loss / ckpt_predict_count
                    train_perp = math.exp(float(mean_loss)) if mean_loss < 300 else math.inf

                    epoch_dur = time.time() - epoch_start_time
                    print("# Finished epoch {:2d} @ step {:5d} @ {}. In the epoch, learning rate = {:.6f}, "
                          "mean loss = {:.4f}, perplexity = {:8.4f}, and {:.2f} seconds elapsed."
                          .format(train_epoch, global_step, time.strftime("%Y-%m-%d %H:%M:%S"),
                                  learning_rate, mean_loss, train_perp, round(epoch_dur, 2)))
                    epoch_start_time = time.time()  # The start time of the next epoch

                    summary = tf.Summary(value=[tf.Summary.Value(tag="train_perp", simple_value=train_perp)])
                    summary_writer.add_summary(summary, global_step)

                    # Save checkpoint
                    if train_perp < 1.6 and train_perp < last_record_perp:
                        self.model.saver.save(sess, os.path.join(result_dir, "basic"), global_step=global_step)
                        last_record_perp = train_perp

                    ckpt_loss, ckpt_predict_count = 0.0, 0.0

                    sess.run(self.model.batch_input.initializer)
                    continue

            # Done training
            print('\n\nTraining finished\n'.format(colorama.Fore.GREEN, colorama.Fore.RESET))

            # save
            self.model.saver.save(sess, os.path.join(result_dir, "basic"), global_step=global_step)
            summary_writer.close()

    @staticmethod
    def _get_learning_rate(perplexity):
        if perplexity <= 1.48:
            return 9.6e-5
        elif perplexity <= 1.64:
            return 1e-4
        elif perplexity <= 2.0:
            return 1.2e-4
        elif perplexity <= 2.4:
            return 1.6e-4
        elif perplexity <= 3.2:
            return 2e-4
        elif perplexity <= 4.8:
            return 2.4e-4
        elif perplexity <= 8.0:
            return 3.2e-4
        elif perplexity <= 16.0:
            return 4e-4
        elif perplexity <= 32.0:
            return 6e-4
        else:
            return 8e-4

if __name__ == "__main__":
    from settings import PROJECT_ROOT

    assert sys.version_info >= (3, 3), \
    "Must be run in Python 3.5 or later. You are running {}".format(sys.version)

    corp_dir = os.path.join(PROJECT_ROOT, 'Data', 'Corpus')
    res_dir = os.path.join(PROJECT_ROOT, 'Data', 'Result')
    bt = BotTrainer(corpus_dir=corp_dir)
    bt.train(res_dir)
```

botpredictor.py

```python
import nltk
import os
import string
import tensorflow as tf

from chatbot.tokenizeddata import TokenizedData
from chatbot.modelcreator import ModelCreator
from chatbot.knowledgebase import KnowledgeBase
from chatbot.sessiondata import SessionData
from chatbot.patternutils import check_patterns_and_replace
from chatbot.functiondata import call_function

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'


class BotPredictor(object):

    def __init__(self, session, corpus_dir, knbase_dir, result_dir, result_file):
        """
        Args:
            session: The TensorFlow session.
            corpus_dir: Name of the folder storing corpus files and vocab information.
            knbase_dir: Name of the folder storing data files for the knowledge base.
            result_dir: The folder containing the trained result files.
            result_file: The file name of the trained model.
        """
        self.session = session

        # Prepare data and hyper parameters
        print("# Prepare dataset placeholder and hyper parameters ...")
        tokenized_data = TokenizedData(corpus_dir=corpus_dir, training=False)

        self.knowledge_base = KnowledgeBase()
        self.knowledge_base.load_knbase(knbase_dir)

        self.session_data = SessionData()

        self.hparams = tokenized_data.hparams
        self.src_placeholder = tf.placeholder(shape=[None], dtype=tf.string)
        src_dataset = tf.data.Dataset.from_tensor_slices(self.src_placeholder)
        self.infer_batch = tokenized_data.get_inference_batch(src_dataset)

        # Create model
        print("# Creating inference model ...")
        self.model = ModelCreator(training=False, tokenized_data=tokenized_data,
                                  batch_input=self.infer_batch)
        # Restore model weights
        print("# Restoring model weights ...")
        self.model.saver.restore(session, os.path.join(result_dir, result_file))

        self.session.run(tf.tables_initializer())


    def predict(self, session_id, question, html_format=False):
        chat_session = self.session_data.get_session(session_id)
        chat_session.before_prediction()  # Reset before each prediction

        if question.strip() == '':
            answer = "Don't you want to say something to me?"
            chat_session.after_prediction(question, answer)
            return answer

        pat_matched, new_sentence, para_list = check_patterns_and_replace(question)
```

botpredictor.py

```python

        for pre_time in range(2):
            tokens = nltk.word_tokenize(new_sentence.lower())
            tmp_sentence = [' '.join(tokens[:]).strip()]

            self.session.run(self.infer_batch.initializer,
                             feed_dict={self.src_placeholder: tmp_sentence})

            outputs, _ = self.model.infer(self.session)

            if self.hparams.beam_width > 0:
                outputs = outputs[0]

            eos_token = self.hparams.eos_token.encode("utf-8")
            outputs = outputs.tolist()[0]

            if eos_token in outputs:
                outputs = outputs[:outputs.index(eos_token)]

            if pat_matched and pre_time == 0:
                out_sentence, if_func_val = self._get_final_output(outputs, chat_session,
                                                                   para_list=para_list,
                                                                   html_format=html_format)
                if if_func_val:
                    chat_session.after_prediction(question, out_sentence)
                    return out_sentence
                else:
                    new_sentence = question
            else:
                out_sentence, _ = self._get_final_output(outputs, chat_session,
                                                         html_format=html_format)
                chat_session.after_prediction(question, out_sentence)
                return out_sentence

    def _get_final_output(self, sentence, chat_session, para_list=None, html_format=False):
        sentence = b' '.join(sentence).decode('utf-8')
        if sentence == '':
            return "I don't know what to say.", False

        if_func_val = False
        last_word = None
        word_list = []
        for word in sentence.split(' '):
            word = word.strip()
            if not word:
                continue

            if word.startswith('_func_val_'):
                if_func_val = True
                word = call_function(word[10:], knowledge_base=self.knowledge_base,
                                     chat_session=chat_session, para_list=para_list,
                                     html_format=html_format)
                if word is None or word == '':
                    continue
            else:
                if word in self.knowledge_base.upper_words:
                    word = self.knowledge_base.upper_words[word]

                if (last_word is None or last_word in ['.', '!', '?']) and not word[0].isupper():
                    word = word.capitalize()

            if not word.startswith('\'') and word != 'n\'t' \
                and (word[0] not in string.punctuation or word in ['(', '[', '{', '``', '$']) \
                and last_word not in ['(', '[', '{', '``', '$']:
                word = ' ' + word

            word_list.append(word)
            last_word = word

        return ''.join(word_list).strip(), if_func_val
```

chatbot.py

```python
import configparser
import shelve
import aiml
import colorama
import wordsegment as ws
import os
import sys
import string
import json
import random
import csv
import re
import language_check

import tensorflow as tf
import nltk
from nltk.corpus import wordnet as wn
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
from nltk.tree import ParentedTree, Tree

from itertools import groupby
from stanfordcorenlp import StanfordCoreNLP
#from tool import filter
import chatbot.crawler as crawler
import chatbot.deeplearning as deep
import chatbot.kdddatabased as kb
from settings import PROJECT_ROOT
from chatbot.botpredictor import BotPredictor

class ChatBot:
    """
        Intelligent dialogue model based on-
        1. Template-based- AIML
        2. Knowledge Based- MySQL \\\
        3. Web Search
        4. Deep Learning: RNN
    """

    # initialize
    colorama.init()
    ws.load()
    #nltk.download()

    def __init__(self, config_file='config.cfg', host='http://localhost', port=9000):
        config = configparser.ConfigParser()
        config.read(config_file)
        self.filter_file = config.get('resource', 'filter_file')
        self.load_file = config.get('resource', 'load_file')
        self.save_file = config.get('resource', 'save_file')
        self.shelve_file = config.get('resource', 'shelve_file')

        corp_dir = os.path.join(PROJECT_ROOT, 'Data', 'Corpus')
        knbs_dir = os.path.join(PROJECT_ROOT, 'Data', 'KnowledgeBase')
        res_dir = os.path.join(PROJECT_ROOT, 'Data', 'Result')

        # Initialize the KERNEL
        self.mybot = aiml.Kernel()
        sess = tf.Session()
        self.predictor = BotPredictor(sess, corpus_dir=corp_dir, knbase_dir=knbs_dir, result_dir=res_dir, result_
        self.session_id = self.predictor.session_data.add_session()

        # Create AI Engine
        if os.path.isfile("model\AIChatEngine.brn"):
            self.mybot.bootstrap(brainFile = "model\AIChatEngine.brn")
        else:
            self.mybot.bootstrap(learnFiles=self.load_file, commands='load aiml b')
            self.mybot.saveBrain("model\AIChatEngine.brn")

        #Initialization learning library
        self.template = '<aiml version="1.0" encoding="UTF-8">\n{rule}\n</aiml>'
        self.category_template = '<category><pattern>{pattern}</pattern><template>{answer}</template></category>'
```

chatbot.py

```python
        # Initialize Filter sensitive words
        #self.gfw = filter.DFAFilter()
        #self.gfw.parse(self.filter_file)

        # Use an existing server: StanfordCoreNLP
        self.nlp = StanfordCoreNLP(host, port=port, timeout=30000)
        self.props = {
            'annotators': 'tokenize,ssplit,pos,lemma,ner,parse,depparse,dcoref,relation',
            'pipelineLanguage': 'en',
            'outputFormat': 'json'
        }

        # Initialize the Language Tool for GEC
        self.tool = language_check.LanguageTool('en-US')


# ##########################################################

    def response(self, user_message):
        print('# User -->: '+user_message)

        # Limit word count
        if len(user_message) > 200:
            return self.mybot.respond('MAX')
        elif len(user_message) < 2:
            return self.mybot.respond('MIN')

        # ****************************************************
        #       Filter sensitive words
        # ****************************************************
        #message = self.gfw.filter(message, "*")
        #if message.find("*") != -1:
            #return self.mybot.respond('FILTER')

        # ****************************************************
        #       Manage Short form words
        # ****************************************************
        user_string = user_message.split(" ")
        j = 0
        for _str in user_string:
            # File path which consists of Abbreviations.
            fileName = "C:\\MyPhdChatbot_Code\\Chatbot_3.6\\Data\\abbreviation_data.txt"
            # File Access mode [Read Mode]
            accessMode = "r"
            with open(fileName, accessMode) as myCSVfile:
                # Reading file as CSV with delimiter as "=", so that abbreviation are stored in row[0]
                dataFromFile = csv.reader(myCSVfile, delimiter="=")
                # Removing Special Characters.
                _str = re.sub('[^a-zA-Z0-9-_.]', '', _str)
                for row in dataFromFile:
                    # Check if selected word matches short forms[LHS] in text file.
                    if _str.upper() == row[0]:
                        # If match found replace it with its appropriate phrase in text file.
                        user_string[j] = row[1]
                myCSVfile.close()
            j = j + 1
        # Replacing commas with spaces for final output.
        user_message = ' '.join(user_string)

        # ****************************************************
        #       Grammar Error Check and Prompt to User
        # ****************************************************
        gec_message = self.checkGrammarError(user_message)
        print('# Correction -->: '+gec_message)
        matches = self.tool.check(user_message)
        if(len(matches)>0):
            return self.mybot.respond('Confirmation '+ gec_message)

        # ****************************************************
        #       Start Conversation
        # ****************************************************
        responseAnswer = ''
        botresponse = self.mybot.respond(gec_message)
        print ('# Bot1  --> ' + botresponse)
```

chatbot.py

```python
148
149         if botresponse[0]=='@':
150             botresponse = botresponse.replace('@','')
151             print('# After Confirmation--> '+botresponse)
152             if gec_message =='Yes':
153                 botresponse = self.mybot.respond(botresponse)
154             else:
155                 return self.mybot.respond('ASK NEW QUERY')
156
157         # Initialize Lemmatization
158         wordnet_lemmatizer = WordNetLemmatizer()
159
160         # User Sentence Tokenization
161         word_tokens = self.nlp.word_tokenize(botresponse)
162
163         # Removing stopwords
164         stop_words = set(stopwords.words('english'))
165         #stopwords.extend(string.punctuation)
166         filtered_sentence = [w for w in word_tokens if not w in stop_words]
167         filtered_stop_words = []
168         for w in word_tokens:
169             if w not in stop_words:
170                 filtered_stop_words.append(w)
171
172         print(colorama.Fore.RED+'\n------------------ User Input Words --> Lemma -------------------------
173         final_sentence = []
174         for word in filtered_stop_words:
175             final_sentence.append(wordnet_lemmatizer.lemmatize(word, pos="v"))
176             print ("{0:10}{1:5}{2:20}".format(word, '--> ', wordnet_lemmatizer.lemmatize(word, pos="v")))
177
178         #print(colorama.Fore.GREEN+'\n******************** Dependency Parser ******************** '+colo
179         #dependency_parser = self.nlp.dependency_parse(' '.join(final_sentence))
180         #print(dependency_parser)
181
182         # POS Tagger
183         postagger = self.nlp.pos_tag(' '.join(final_sentence))
184         print(colorama.Fore.YELLOW+'\n------------------ Identify POS Tagger -------------------------- '+c
185         print('pos tagger: ', postagger)
186
187         print("----------------------------------------------------------------------")
188         grammar = r"""Chunk: {<RB.?>*<VB.?>*<NNP>+<NN>?}"""
189         cp = nltk.RegexpParser(grammar)
190         #tree = cp.parse(postagger)
191         #print ("CP: ", cp)
192         tree = cp.parse(postagger)
193         print (tree)
193         print (tree)
194
195         for word, pos in postagger:
196             if pos=='NNP':
197                 print (word)
198
199         # Add all NOUNs into list
200         nounEntityList = []
201         for pos in postagger:
202             if pos[1] in ('NN','NNS','NNP','NNPS'):
203                 nounEntityList.append(pos[0])
204         print(nounEntityList, '\n')
205
206
207         # 1: Template-based Strategy
208         if botresponse[0] != '#':
209             print('Template-based Strategy')
210             responseAnswer = botresponse
211
212         # 2: KB Searching Strategy
213         elif botresponse.find('#NONE#') != -1:
214             nounEntityList.remove('#NONE')
215             ans = ''
216             #ans = kb.kdd_search(nounEntityList, ' '.join(final_sentence), gec_message)
217             if ans != '':
218                 print('KB Searching Strategy')
219                 responseAnswer = ans.encode('utf-8')
```

chatbot.py

```python
            # 3: Internet Retrieval Strategy
            else:
                #ans = crawler.web_search(gec_message)
                if ans != '':
                    print('Internet Retrieval Strategy')
                    responseAnswer = ans.encode('utf-8')

                # 4: Generative Strategy- RNN
                else:
                    if gec_message =='Yes':
                        confirm_mgs = botresponse.replace('#NONE#:','')
                        ans = deep.neural_network(self, confirm_mgs)
                        print('Generative Strategy with - YES')
                        print(confirm_mgs)
                    else:
                        ans = deep.neural_network(self, gec_message)
                        print('Generative Strategy')

                    responseAnswer = ans.encode('utf-8')

        # Learning Mode
        elif result.find('#LEARN#') != -1:
            question = result[8:]
            answer = message
            self.save(question, answer)
            return self.mybot.respond('Already studied')

        else:
            responseAnswer = self.mybot.respond('I don\'t know.')

        return responseAnswer


    # Grammar Error Check on Raw User Input
    def checkGrammarError(self, user_message):
        print(colorama.Fore.GREEN+'\n------------------ Grammar Error Correction --------------------------
        matches = self.tool.check(user_message)
        gec_user_message = language_check.correct(user_message, matches)
        if(len(matches)>0):
            i = 0
            for x in matches:
                print('Grammatical Error --> ', matches[i])
                print('Apply Rules--> ', matches[i].replacements)
                i=i+1
        else:
            print('No Error Found.')
        return gec_user_message

    # SAVE Model
    def save(self, question, answer):
        db = shelve.open(self.shelve_file, 'c', writeback=True)
        db[question] = answer
        db.sync()
        rules = []
        for r in db:
            rules.append(self.category_template.format(pattern=r, answer=db[r]))
        with open(self.save_file, 'w') as fp:
            fp.write(self.template.format(rule='\n'.join(rules)))


    def forget(self):
        os.remove(self.save_file) if os.path.exists(self.save_file) else None
        os.remove(self.shelve_file) if os.path.exists(self.shelve_file) else None
        self.mybot.bootstrap(learnFiles=self.load_file, commands='load aiml b')


if __name__ == '__main__':
    bot = ChatBot()
    while True:
        user_message = raw_input('User > ')
        bot.response(user_message)
```

chathistory.py

```python
import mysql.connector

# Save Chat History in Database
def chat_history(message):

    # Connect to MySQL Database
    mydb = mysql.connector.connect(
      host="localhost",
      user="root",
      passwd="soho123",
      database="aichatbot"
    )
    dbcursor = mydb.cursor()

    sql = 'INSERT INTO customers (name, address) VALUES (%s, %s)'
    val = ("John", "Highway 21")
    dbcursor.execute(sql, val)
    mydb.commit()

    return 'Record inserted.'

if __name__ == '__main__':
    print (chat_history('Chat History'))
```

crawler.py

```python
from chatbot.search import *

#####################
#     Search Web    #
#####################
def web_search(message):
    result = ''

    '''Wikipedia'''
    ### Need to Identify Entity before search from Wiki
    if message.find(message) != -1:
        result += get_wikipedia(message)
        return result

    '''Britannica'''
    ### Need to Identify Entity before search from Wiki
    if message.find(message) != -1:
        result += get_britannica(message)
        return result

    '''Joke'''
    if message.find('joke') != -1:
        result += 'Ok, here is a joke for you~~~\n'
        result += get_joke()
        return result

    return result

if __name__ == '__main__':
    message = "IntelliBot"
    print (search(message))
```

deepleaming.py

```python
import os
import sys
import json
import requests
import tensorflow as tf
import colorama
from bs4 import BeautifulSoup
from settings import PROJECT_ROOT
from chatbot.botpredictor import BotPredictor

colorama.init()

# Recurrent Neural Network for Predicting Appropriate Response
def neural_network(self, message):

    ans_response = ''
    ans_response = self.predictor.predict(self.session_id, message)
    print(ans_response)

    return ans_response


if __name__ == '__main__':
    print (rnn_generator('IntelliBot: RNN'))

```

hparams.py

```python
import codecs
import json
import os
import tensorflow as tf


class HParams:
    def __init__(self, model_dir):
        """
        Args:
            model_dir: Name of the folder storing the hparams.json file.
        """
        self.hparams = self.load_hparams(model_dir)

    @staticmethod
    def load_hparams(model_dir):
        """Load hparams from an existing directory."""
        hparams_file = os.path.join(model_dir, "hparams.json")
        if tf.gfile.Exists(hparams_file):
            print("# Loading hparams from {} ...".format(hparams_file))
            with codecs.getreader("utf-8")(tf.gfile.GFile(hparams_file, "rb")) as f:
                try:
                    hparams_values = json.load(f)
                    hparams = tf.contrib.training.HParams(**hparams_values)
                except ValueError:
                    print("Error loading hparams file.")
                    return None
            return hparams
        else:
            return None

```

functiondata.py

```python
import calendar as cal
import datetime as dt
import random
import re
import time

#This class is only used at inference time.
class FunctionData:

    def __init__(self, knowledge_base, chat_session, html_format):
        """
        Args:
            knowledge_base: The knowledge base data needed for prediction.
            chat_session: The chat session object that can be read and written.
            html_format: Whether out_sentence is in HTML format.
        """
        self.knowledge_base = knowledge_base
        self.chat_session = chat_session
        self.html_format = html_format

    """
    # Rule 2: Date and Time
    """
    @staticmethod
    def get_date_time():
        return time.strftime("%Y-%m-%d %H:%M")

    @staticmethod
    def get_time():
        return time.strftime("%I:%M %p")

    @staticmethod
    def get_today():
        return "{:%B %d, %Y}".format(dt.date.today())

    @staticmethod
    def get_weekday(day_delta):
        now = dt.datetime.now()
        if day_delta == 'd_2':
            day_time = now - dt.timedelta(days=2)
        elif day_delta == 'd_1':
            day_time = now - dt.timedelta(days=1)
        elif day_delta == 'd1':
            day_time = now + dt.timedelta(days=1)
        elif day_delta == 'd2':
            day_time = now + dt.timedelta(days=2)
        else:
            day_time = now

        weekday = cal.day_name[day_time.weekday()]
        return "{}, {:%B %d, %Y}".format(weekday, day_time)
    """
    # Rule 3: Stories and Jokes, and last topic
    """
    def get_story_any(self):
        self.chat_session.last_topic = "STORY"
        self.chat_session.keep_topic = True

        stories = self.knowledge_base.stories
        _, content = random.choice(list(stories.items()))
        if not self.html_format:
            content = re.sub(r'_np_', '', content)
        return content

    def continue_last_topic(self):
        if self.chat_session.last_topic == "STORY":
            self.chat_session.keep_topic = True
            return self.get_story_any()
        elif self.chat_session.last_topic == "JOKE":
            self.chat_session.keep_topic = True
            return self.get_joke_any()
        else:
            return "Sorry, but what topic do you prefer?"
```

functiondata.py

```python

    """
    # Rule 4: Arithmetic ops
    """
    @staticmethod
    def get_number_plus(num1, num2):
        res = num1 + num2
        desc = random.choice(FunctionData.easy_list)
        return "{}{} + {} = {}".format(desc, num1, num2, res)

    @staticmethod
    def get_number_minus(num1, num2):
        res = num1 - num2
        desc = random.choice(FunctionData.easy_list)
        return "{}{} - {} = {}".format(desc, num1, num2, res)

    @staticmethod
    def get_number_multiply(num1, num2):
        res = num1 * num2
        if num1 > 100 and num2 > 100 and num1 % 2 == 1 and num2 % 2 == 1:
            desc = random.choice(FunctionData.hard_list)
        else:
            desc = random.choice(FunctionData.easy_list)
        return "{}{} * {} = {}".format(desc, num1, num2, res)

    @staticmethod
    def get_number_divide(num1, num2):
        if num2 == 0:
            return "Sorry, but that does not make sense as the divisor cannot be zero."
        else:
            res = num1 / num2
            if isinstance(res, int):
                if 50 < num1 != num2 > 50:
                    desc = random.choice(FunctionData.hard_list)
                else:
                    desc = random.choice(FunctionData.easy_list)
                return "{}{} / {} = {}".format(desc, num1, num2, res)
            else:
                if num1 > 20 and num2 > 20:
                    desc = random.choice(FunctionData.hard_list)
                else:
                    desc = random.choice(FunctionData.easy_list)
                return "{}{} / {} = {:.2f}".format(desc, num1, num2, res)
    """
    # Rule 5: User name, call me information, and last question and answer
    """
    def ask_howru_if_not_yet(self):
        howru_asked = self.chat_session.howru_asked
        if howru_asked:
            return ""
        else:
            self.chat_session.howru_asked = True
            return random.choice(FunctionData.ask_howru_list)

    def ask_name_if_not_yet(self):
        user_name = self.chat_session.user_name
        call_me = self.chat_session.call_me
        if user_name or call_me:
            return ""
        else:
            return random.choice(FunctionData.ask_name_list)

    def get_user_name_and_reply(self):
        user_name = self.chat_session.user_name
        if user_name and user_name.strip() != '':
            return user_name
        else:
            return "Did you tell me your name? Sorry, I missed that."
```

functiondata.py

```python
    def get_callme(self, punc_type):
        call_me = self.chat_session.call_me
        user_name = self.chat_session.user_name

        if call_me and call_me.strip() != '':
            if punc_type == 'comma0':
                return ", {}".format(call_me)
            else:
                return call_me
        elif user_name and user_name.strip() != '':
            if punc_type == 'comma0':
                return ", {}".format(user_name)
            else:
                return user_name
        else:
            return ""

    def get_last_question(self):
        # Do not record this pair as the last question and answer
        self.chat_session.update_pair = False

        last_question = self.chat_session.last_question
        if last_question is None or last_question.strip() == '':
            return "You did not say anything."
        else:
            return "You have just said: {}".format(last_question)

    def get_last_answer(self):
        # Do not record this pair as the last question and answer
        self.chat_session.update_pair = False

        last_answer = self.chat_session.last_answer
        if last_answer is None or last_answer.strip() == '':
            return "I did not say anything."
        else:
            return "I have just said: {}".format(last_answer)

    def update_user_name(self, new_name):
        return self.update_user_name_and_call_me(new_name=new_name)

    def update_call_me(self, new_call):
        return self.update_user_name_and_call_me(new_call=new_call)

    def update_user_name_and_call_me(self, new_name=None, new_call=None):
        user_name = self.chat_session.user_name
        call_me = self.chat_session.call_me
        # print("{}; {}; {}; {}".format(user_name, call_me, new_name, new_call))

        if user_name and new_name and new_name.strip() != '':
            if new_name.lower() != user_name.lower():
                self.chat_session.update_pending_action('update_user_name_confirmed', None, new_name)
                return "I am confused. I have your name as {}. Did I get it correctly?".format(user_name)
            else:
                return "You told me your name already. Thank you, {}, for assuring me.".format(user_name)

        if call_me and new_call and new_call.strip() != '':
            if new_call.lower() != call_me.lower():
                self.chat_session.update_pending_action('update_call_me_confirmed', new_call, None)
                return "You wanted me to call you {}. Would you like me to call you {} now?"\
                    .format(call_me, new_call)
            else:
                return "Thank you for letting me again, {}.".format(call_me)

        if new_call and new_call.strip() != '':
            if new_name and new_name.strip() != '':
                self.chat_session.user_name = new_name

            self.chat_session.call_me = new_call
            return "Thank you, {}.".format(new_call)
        elif new_name and new_name.strip() != '':
            self.chat_session.user_name = new_name
            return "Thank you, {}.".format(new_name)

        return "Sorry, I am confused. I could not figure out what you meant."
```

functiondata.py

```python

    def update_user_name_enforced(self, new_name):
        if new_name and new_name.strip() != '':
            self.chat_session.user_name = new_name
            return "OK, thank you, {}.".format(new_name)
        else:
            self.chat_session.user_name = None  # Clear the existing user_name, if any.
            return "Sorry, I am lost."

    def update_call_me_enforced(self, new_call):
        if new_call and new_call.strip() != '':
            self.chat_session.call_me = new_call
            return "OK, got it. Thank you, {}.".format(new_call)
        else:
            self.chat_session.call_me = None  # Clear the existing call_me, if any.
            return "Sorry, I am totally lost."

    def update_user_name_and_reply_papaya(self, new_name):
        user_name = self.chat_session.user_name

        if new_name and new_name.strip() != '':
            if user_name:
                if new_name.lower() != user_name.lower():
                    self.chat_session.update_pending_action('update_user_name_confirmed', None, new_name)
                    return "I am confused. I have your name as {}. Did I get it correctly?".format(user_name)
                else:
                    return "Thank you, {}, for assuring me your name. My name is Nuruzzaman.".format(user_name)
            else:
                self.chat_session.user_name = new_name
                return "Thank you, {}. BTW, my name is Nuruzzaman.".format(new_name)
        else:
            return "My name is Nuruzzaman. Thanks."

    def correct_user_name(self, new_name):
        if new_name and new_name.strip() != '':
            self.chat_session.user_name = new_name
            return "Thank you, {}.".format(new_name)
        else:
            # Clear the existing user_name and call_me information
            self.chat_session.user_name = None
            self.chat_session.call_me = None
            return "I am totally lost."

    def clear_user_name_and_call_me(self):
        self.chat_session.user_name = None
        self.chat_session.call_me = None

    def execute_pending_action_and_reply(self, answer):
        func = self.chat_session.pending_action['func']
        if func == 'update_user_name_confirmed':
            if answer.lower() == 'yes':
                reply = "Thank you, {}, for confirming this.".format(self.chat_session.user_name)
            else:
                new_name = self.chat_session.pending_action['No']
                self.chat_session.user_name = new_name
                reply = "Thank you, {}, for correcting me.".format(new_name)
        elif func == 'update_call_me_confirmed':
            if answer.lower() == 'yes':
                new_call = self.chat_session.pending_action['Yes']
                self.chat_session.call_me = new_call
                reply = "Thank you, {}, for correcting me.".format(new_call)
            else:
                reply = "Thank you. I will continue to call you {}.".format(self.chat_session.call_me)
        else:
            reply = "OK, thanks."  # Just presents a reply that is good for most situations

        # Clear the pending action anyway
        self.chat_session.clear_pending_action()
        return reply

    """
    # Other Rules: Client Code
    """
    def client_code_show_picture_randomly(self, picture_name):
        if not self.html_format:  # Ignored in the command line interface
            return ''
        else:
            return ' _cc_start_show_picture_randomly_paral_' + picture_name + '_cc_end_'
```

functiondata.py

```python
def call_function(func_info, knowledge_base=None, chat_session=None, para_list=None,
                  html_format=False):
    func_data = FunctionData(knowledge_base, chat_session, html_format=html_format)

    func_dict = {
        'get_date_time': FunctionData.get_date_time,
        'get_time': FunctionData.get_time,
        'get_today': FunctionData.get_today,
        'get_weekday': FunctionData.get_weekday,

        'get_story_any': func_data.get_story_any,
        'get_story_name': func_data.get_story_name,
        'get_joke_any': func_data.get_joke_any,
        'continue_last_topic': func_data.continue_last_topic,

        'get_number_plus': FunctionData.get_number_plus,
        'get_number_minus': FunctionData.get_number_minus,
        'get_number_multiply': FunctionData.get_number_multiply,
        'get_number_divide': FunctionData.get_number_divide,

        'ask_howru_if_not_yet': func_data.ask_howru_if_not_yet,
        'ask_name_if_not_yet': func_data.ask_name_if_not_yet,
        'get_user_name_and_reply': func_data.get_user_name_and_reply,
        'get_callme': func_data.get_callme,
        'get_last_question': func_data.get_last_question,
        'get_last_answer': func_data.get_last_answer,

        'update_user_name': func_data.update_user_name,
        'update_call_me': func_data.update_call_me,
        'update_user_name_and_call_me': func_data.update_user_name_and_call_me,
        'update_user_name_enforced': func_data.update_user_name_enforced,
        'update_call_me_enforced': func_data.update_call_me_enforced,
        'update_user_name_and_reply_papaya': func_data.update_user_name_and_reply_papaya,

        'correct_user_name': func_data.correct_user_name,
        'clear_user_name_and_call_me': func_data.clear_user_name_and_call_me,

        'execute_pending_action_and_reply': func_data.execute_pending_action_and_reply,

        'client_code_show_picture_randomly': func_data.client_code_show_picture_randomly
    }
    para1_index = func_info.find('_para1_')
    para2_index = func_info.find('_para2_')
    if para1_index == -1:  # No parameter at all
        func_name = func_info
        if func_name in func_dict:
            return func_dict[func_name]()
    else:
        func_name = func_info[:para1_index]
        if para2_index == -1:  # Only one parameter
            func_para = func_info[para1_index+7:]
            if func_para == '_name_' and para_list is not None and len(para_list) >= 1:
                return func_dict[func_name](para_list[0])
            elif func_para == '_callme_' and para_list is not None and len(para_list) >= 2:
                return func_dict[func_name](para_list[1])
            else:  # The parameter value was embedded in the text (part of the string) of the training
                return func_dict[func_name](func_para)
        else:
            func_para1 = func_info[para1_index+7:para2_index]
            func_para2 = func_info[para2_index+7:]
            if para_list is not None and len(para_list) >= 2:
                para1_val = para_list[0]
                para2_val = para_list[1]

                if func_para1 == '_num1_' and func_para2 == '_num2_':
                    return func_dict[func_name](para1_val, para2_val)
                elif func_para1 == '_num2_' and func_para2 == '_num1_':
                    return func_dict[func_name](para2_val, para1_val)
                elif func_para1 == '_name_' and func_para2 == '_callme_':
                    return func_dict[func_name](para1_val, para2_val)

    return "You beat me to it, and I cannot tell which is which for this question."
```

kdddatabased.py

```python
import mysql.connector
import collections
import colorama
import string
import nltk
from nltk import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

colorama.init()
wordnet_lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

# Knowledge-based (KDD)Search
def kdd_search(nounEntityList, user_input_text, gec_message):

    print(gec_message)
    try:
        # Connect to MySQL Database
        connection = mysql.connector.connect(host='localhost',database='*',user='root',password='*')

        if connection.is_connected():
            db_Info = connection.get_server_info()
            dbcursor = connection.cursor()
            ans_reply = ''
            ans_id = 0
            response_count = 0
            row_count=0
            queryResultList=[]
            if (user_input_text !=''):
                mysqlstatement = 'SELECT id, response_count, tag_id, question, response_message,
                 keywords, message_type FROM knowledgebase WHERE question like\'%'+gec_message+'%\' '
                dbcursor.execute(mysqlstatement)
                qres = dbcursor.fetchall()
                for res in qres:
                    queryResultList.append(res)
                    row_count = row_count + dbcursor.rowcount
                if (row_count ==1):
                    ans_id = queryResultList[0][0]
                    response_count = queryResultList[0][1]
                    ans_reply = queryResultList[0][4]
                    return ans_reply
                else:
                    for entity in nounEntityList:
                        mysqlstatement = 'SELECT id, response_count, tag_id, question, response_message,
                         keywords, message_type FROM knowledgebase WHERE question like\'%'+entity+'%\'
                         OR keywords like\'%'+entity+'%\' '
                        dbcursor.execute(mysqlstatement)
                        qres = dbcursor.fetchall()
                        for res in qres:
                            queryResultList.append(res)
                            row_count = row_count + dbcursor.rowcount
                    i=0
                    for result in queryResultList:
                        i = i+1

                    if (row_count ==0):
                        # retrun empty for Web Search and DRNN functions
                        ans_reply = ''
                    elif (row_count ==1):
                        ans_id = queryResultList[0][0]
                        response_count = queryResultList[0][1]
                        ans_reply = queryResultList[0][4]
                    else:
                        # Remove duplicate records
                        no_duplicate = set(queryResultList)

                        if(len(no_duplicate) > 1):
                            response = sqlforMinimizeRecords(nounEntityList, no_duplicate, dbcursor)
                            ans_id = response[0][0]
                            response_count = response[0][1]
                            ans_reply = response[0][4]

                        elif(len(no_duplicate) == 1):
                            # After removing duplicate, if only one record
                            ans_id = queryResultList[0][0]
                            response_count = queryResultList[0][1]
                            ans_reply = queryResultList[0][4]
```

kdddatabased.py

```python
                    else:
                        ans_reply = ''

                # Get Jaccard similarity
                similarity_ratio = token_match(user_input_text, ans_reply)
                print(ans_reply,"\t", similarity_ratio)
        return ans_reply

    except Error as e:
        print ("Error while connecting to MySQL", e)
    finally:
        # Closing database connection
        if(connection.is_connected()):
            dbcursor.close()
            connection.close()

def token_match(a, b):
    # Question-> tokens_a --> target_sentence
    tokens_a = [token.lower().strip(string.punctuation) for token in nltk.tokenize.word_tokenize(a) \
                if token.lower().strip(string.punctuation) ]
    # Answers -> tokens_b -> ans_sentence
    word_token_b = [token.lower().strip(string.punctuation) for token in nltk.tokenize.word_tokenize(b) \
                if token.lower().strip(string.punctuation) ]

    # Tokenization, Lemmatization and Removing Words
    filtered_sentence = [w for w in word_token_b if not w in stop_words]
    filtered_stop_words = []
    for w in word_token_b:
        if w not in stop_words:
            filtered_stop_words.append(w)
    tokens_b = []
    for word in filtered_stop_words:
        tokens_b.append(wordnet_lemmatizer.lemmatize(word, pos="v"))

    print('a--> ', tokens_a)
    print('b-->', tokens_b)
    # Calculate Jaccard similarity
    ratio = len(set(tokens_a).intersection(tokens_b)) / float(len(set(tokens_a).union(tokens_b)))

    return ratio

def sqlforMinimizeRecords(nounEntityList, no_duplicate, dbcursor):
    ans_reply= []
    row_count = 0

    if(len(nounEntityList) ==1):
        nounEntityList.append(nounEntityList[0])

    # Loop all entities through SQL
    for entity in nounEntityList:
        mysqlstatement = 'SELECT id, response_count, tag_id, question, response_message, keywords,
        message_type FROM knowledgebase WHERE question like\'%'+entity+'%\' OR keywords like\'%'
        +entity+'%\' HAVING question like\'%'+nounEntityList[0]+'%\' AND question like\'%'
        +nounEntityList[1]+'%\'  '
        dbcursor.execute(mysqlstatement)
        qres = dbcursor.fetchall()
        row_count = dbcursor.rowcount

    if (row_count >0):
        print(qres)
        ans_reply = qres

    return ans_reply
```

knowledgebase.py

```python
import os
UPPER_FILE = "upper_words.txt"
STORIES_FILE = "stories.txt"
JOKES_FILE = "jokes.txt"

#This class is only used at inference time
class KnowledgeBase:
    def __init__(self):
        self.upper_words = {}
        self.stories = {}
        self.jokes = []

    def load_knbase(self, knbase_dir):
        """
        Args:
            knbase_dir: Name of the KnowledgeBase folder. The file names inside are fixed.
        """
        upper_file_name = os.path.join(knbase_dir, UPPER_FILE)
        stories_file_name = os.path.join(knbase_dir, STORIES_FILE)
        jokes_file_name = os.path.join(knbase_dir, JOKES_FILE)

        with open(upper_file_name, 'r') as upper_f:
            for line in upper_f:
                ln = line.strip()
                if not ln or ln.startswith('#'):
                    continue
                cap_words = ln.split(',')
                for cpw in cap_words:
                    tmp = cpw.strip()
                    self.upper_words[tmp.lower()] = tmp

        with open(stories_file_name, 'r') as stories_f:
            s_name, s_content = '', ''
            for line in stories_f:
                ln = line.strip()
                if not ln or ln.startswith('#'):
                    continue
                if ln.startswith('_NAME:'):
                    if s_name != '' and s_content != '':
                        self.stories[s_name] = s_content
                        s_name, s_content = '', ''
                    s_name = ln[6:].strip().lower()
                elif ln.startswith('_CONTENT:'):
                    s_content = ln[9:].strip()
                else:
                    s_content += ' ' + ln.strip()

            if s_name != '' and s_content != '':  # The last one
                self.stories[s_name] = s_content

        with open(jokes_file_name, 'r') as jokes_f:
            for line in jokes_f:
                ln = line.strip()
                if not ln or ln.startswith('#'):
                    continue
                self.jokes.append(ln)
```

modelcreator.py

```python
import tensorflow as tf
import chatbot.modelhelper as model_helper
import colorama
from tensorflow.python.layers import core as layers_core

colorama.init()

class ModelCreator(object):
    """Sequence-to-sequence model creator to create models for training or inference"""
    def __init__(self, training, tokenized_data, batch_input, scope=None):
        """
        Create the model.

        Args:
            training: A boolean value to indicate whether this model will be used for training.
            tokenized_data: The data object containing all information required for the model.
            scope: scope of the model.
        """
        self.training = training
        self.batch_input = batch_input
        self.vocab_table = tokenized_data.vocab_table
        self.vocab_size = tokenized_data.vocab_size
        self.reverse_vocab_table = tokenized_data.reverse_vocab_table

        hparams = tokenized_data.hparams
        self.hparams = hparams

        self.num_layers = hparams.num_layers
        self.time_major = hparams.time_major

        # Initializer
        initializer = model_helper.get_initializer(
            hparams.init_op, hparams.random_seed, hparams.init_weight)
        tf.get_variable_scope().set_initializer(initializer)

        # Embeddings
        self.embedding = (model_helper.create_embbeding(vocab_size=self.vocab_size,
                                                        embed_size=hparams.num_units,
                                                        scope=scope))
        # This batch_size might vary among each batch instance due to the bucketing and/or reach
        # the end of the training set. Treat it as size_of_the_batch.
        self.batch_size = tf.size(self.batch_input.source_sequence_length)

        # Projection
        with tf.variable_scope(scope or "build_network"):
            with tf.variable_scope("decoder/output_projection"):
                self.output_layer = layers_core.Dense(
                    self.vocab_size, use_bias=False, name="output_projection")

        # Training or inference graph
        print('\n\n{} Building graph for the model ...{}\n'.format(colorama.Fore.GREEN, colorama.Fore.RESET))
        print("****************************************")
        res = self.build_graph(hparams, scope=scope)

        if training:
            self.train_loss = res[1]
            self.word_count = tf.reduce_sum(self.batch_input.source_sequence_length) + \
                              tf.reduce_sum(self.batch_input.target_sequence_length)
            # Count the number of predicted words for compute perplexity.
            self.predict_count = tf.reduce_sum(self.batch_input.target_sequence_length)
        else:
            self.infer_logits, _, self.final_context_state, self.sample_id = res
            self.sample_words = self.reverse_vocab_table.lookup(tf.to_int64(self.sample_id))

        self.global_step = tf.Variable(0, trainable=False)
        params = tf.trainable_variables()

        # Gradients update operation for training the model.
        if training:
            self.learning_rate = tf.placeholder(tf.float32, shape=[], name='learning_rate')
            opt = tf.train.AdamOptimizer(self.learning_rate)
            gradients = tf.gradients(self.train_loss, params)
            colocate_gradients_with_ops=True

            clipped_gradients, gradient_norm_summary = model_helper.gradient_clip(
                gradients, max_gradient_norm=hparams.max_gradient_norm)
```

modelcreator.py

```python
            self.update = opt.apply_gradients(
                zip(clipped_gradients, params), global_step=self.global_step)

            # Summary
            self.train_summary = tf.summary.merge([
                tf.summary.scalar("learning_rate", self.learning_rate),
                tf.summary.scalar("train_loss", self.train_loss),
            ] + gradient_norm_summary)
        else:
            self.infer_summary = tf.no_op()

        # Saver
        self.saver = tf.train.Saver(tf.global_variables())

    def train_step(self, sess, learning_rate):
        """Run one step of training."""
        assert self.training

        return sess.run([self.update,
                         self.train_loss,
                         self.predict_count,
                         self.train_summary,
                         self.global_step,
                         self.word_count,
                         self.batch_size],
                        feed_dict={self.learning_rate: learning_rate})

    def build_graph(self, hparams, scope=None):
        """Creates a sequence-to-sequence model with dynamic RNN decoder API."""
        dtype = tf.float32

        with tf.variable_scope(scope or "dynamic_seq2seq", dtype=dtype):
            # Encoder
            encoder_outputs, encoder_state = self._build_encoder(hparams)

            # Decoder
            logits, sample_id, final_context_state = self._build_decoder(
                encoder_outputs, encoder_state, hparams)

            # Loss
            if self.training:
                loss = self._compute_loss(logits)
            else:
                loss = None

            return logits, loss, final_context_state, sample_id

    def _build_encoder(self, hparams):
        """Build an encoder."""
        source = self.batch_input.source
        if self.time_major:
            source = tf.transpose(source)

        with tf.variable_scope("encoder") as scope:
            dtype = scope.dtype
            # Look up embedding, emp_inp: [max_time, batch_size, num_units]
            encoder_emb_inp = tf.nn.embedding_lookup(self.embedding, source)

            # Encoder_outpus: [max_time, batch_size, num_units]
            cell = self._build_encoder_cell(hparams)

            encoder_outputs, encoder_state = tf.nn.dynamic_rnn(
                cell,
                encoder_emb_inp,
                dtype=dtype,
                sequence_length=self.batch_input.source_sequence_length,
                time_major=self.time_major)

        return encoder_outputs, encoder_state
```

modelcreator.py

```python

    def _build_encoder_cell(self, hparams):
        """Build a multi-layer RNN cell that can be used by encoder."""
        return model_helper.create_rnn_cell(
            num_units=hparams.num_units,
            num_layers=hparams.num_layers,
            keep_prob=hparams.keep_prob)

    def _build_decoder(self, encoder_outputs, encoder_state, hparams):
        """Build and run a RNN decoder with a final projection layer."""
        bos_id = tf.cast(self.vocab_table.lookup(tf.constant(hparams.bos_token)), tf.int32)
        eos_id = tf.cast(self.vocab_table.lookup(tf.constant(hparams.eos_token)), tf.int32)

        # maximum_iteration: The maximum decoding steps.
        if hparams.tgt_max_len_infer:
            maximum_iterations = hparams.tgt_max_len_infer
        else:
            decoding_length_factor = 2.0
            max_encoder_length = tf.reduce_max(self.batch_input.source_sequence_length)
            maximum_iterations = tf.to_int32(tf.round(
                tf.to_float(max_encoder_length) * decoding_length_factor))

        # Decoder.
        with tf.variable_scope("decoder") as decoder_scope:
            cell, decoder_initial_state = self._build_decoder_cell(
                hparams, encoder_outputs, encoder_state,
                self.batch_input.source_sequence_length)

            # Training
            if self.training:
                # decoder_emp_inp: [max_time, batch_size, num_units]
                target_input = self.batch_input.target_input
                if self.time_major:
                    target_input = tf.transpose(target_input)
                decoder_emb_inp = tf.nn.embedding_lookup(self.embedding, target_input)

                # Helper
                helper = tf.contrib.seq2seq.TrainingHelper(
                    decoder_emb_inp, self.batch_input.target_sequence_length,
                    time_major=self.time_major)

                # Decoder
                my_decoder = tf.contrib.seq2seq.BasicDecoder(
                    cell,
                    helper,
                    decoder_initial_state,)

                # Dynamic decoding
                outputs, final_context_state, _ = tf.contrib.seq2seq.dynamic_decode(
                    my_decoder,
                    output_time_major=self.time_major,
                    swap_memory=True,
                    scope=decoder_scope)

                sample_id = outputs.sample_id
                logits = self.output_layer(outputs.rnn_output)
            # Inference
            else:
                beam_width = hparams.beam_width
                length_penalty_weight = hparams.length_penalty_weight
                start_tokens = tf.fill([self.batch_size], bos_id)
                end_token = eos_id

                if beam_width > 0:
                    my_decoder = tf.contrib.seq2seq.BeamSearchDecoder(
                        cell=cell,
                        embedding=self.embedding,
                        start_tokens=start_tokens,
                        end_token=end_token,
                        initial_state=decoder_initial_state,
                        beam_width=beam_width,
                        output_layer=self.output_layer,
                        length_penalty_weight=length_penalty_weight)
```

modelcreator.py

```python
            else:
                # Helper
                helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
                    self.embedding, start_tokens, end_token)

                # Decoder
                my_decoder = tf.contrib.seq2seq.BasicDecoder(
                    cell,
                    helper,
                    decoder_initial_state,
                    output_layer=self.output_layer  # applied per timestep
                )

            # Dynamic decoding
            outputs, final_context_state, _ = tf.contrib.seq2seq.dynamic_decode(
                my_decoder,
                maximum_iterations=maximum_iterations,
                output_time_major=self.time_major,
                swap_memory=True,
                scope=decoder_scope)

            if beam_width > 0:
                logits = tf.no_op()
                sample_id = outputs.predicted_ids
            else:
                logits = outputs.rnn_output
                sample_id = outputs.sample_id

        return logits, sample_id, final_context_state

    def _build_decoder_cell(self, hparams, encoder_outputs, encoder_state,
                            source_sequence_length):
        """Build a RNN cell with attention mechanism that can be used by decoder."""
        num_units = hparams.num_units
        num_layers = hparams.num_layers
        beam_width = hparams.beam_width

        dtype = tf.float32

        if self.time_major:
            memory = tf.transpose(encoder_outputs, [1, 0, 2])
        else:
            memory = encoder_outputs

        if not self.training and beam_width > 0:
            memory = tf.contrib.seq2seq.tile_batch(memory, multiplier=beam_width)
            source_sequence_length = tf.contrib.seq2seq.tile_batch(source_sequence_length,
                                                                   multiplier=beam_width)
            encoder_state = tf.contrib.seq2seq.tile_batch(encoder_state,
                                                          multiplier=beam_width)
            batch_size = self.batch_size * beam_width
        else:
            batch_size = self.batch_size

        attention_mechanism = tf.contrib.seq2seq.LuongAttention(
            num_units, memory, memory_sequence_length=source_sequence_length)

        cell = model_helper.create_rnn_cell(
            num_units=num_units,
            num_layers=num_layers,
            keep_prob=hparams.keep_prob)

        # Only generate alignment in greedy INFER mode.
        alignment_history = (not self.training and beam_width == 0)
        cell = tf.contrib.seq2seq.AttentionWrapper(
            cell,
            attention_mechanism,
            attention_layer_size=num_units,
            alignment_history=alignment_history,
            name="attention")

        if hparams.pass_hidden_state:
            decoder_initial_state = cell.zero_state(batch_size, dtype).clone(cell_state=encoder_state)
        else:
            decoder_initial_state = cell.zero_state(batch_size, dtype)

        return cell, decoder_initial_state
```

modelcreator.py

```python

    def _compute_loss(self, logits):
        """Compute optimization loss."""
        target_output = self.batch_input.target_output
        if self.time_major:
            target_output = tf.transpose(target_output)
        max_time = self.get_max_time(target_output)
        crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(
            labels=target_output, logits=logits)
        target_weights = tf.sequence_mask(
            self.batch_input.target_sequence_length, max_time, dtype=logits.dtype)
        if self.time_major:
            target_weights = tf.transpose(target_weights)

        loss = tf.reduce_sum(crossent * target_weights) / tf.to_float(self.batch_size)
        return loss

    def get_max_time(self, tensor):
        time_axis = 0 if self.time_major else 1
        return tensor.shape[time_axis].value or tf.shape(tensor)[time_axis]

    def infer(self, sess):
        assert not self.training
        _, infer_summary, _, sample_words = sess.run([
            self.infer_logits, self.infer_summary, self.sample_id, self.sample_words
        ])

        # make sure outputs is of shape [batch_size, time]
        if self.time_major:
            sample_words = sample_words.transpose()

        return sample_words, infer_summary
```

modelhelper.py

```python
import tensorflow as tf

def get_initializer(init_op, seed=None, init_weight=None):
    """Create an initializer. init_weight is only for uniform."""
    if init_op == "uniform":
        assert init_weight
        return tf.random_uniform_initializer(-init_weight, init_weight, seed=seed)
    elif init_op == "glorot_normal":
        return tf.contrib.keras.initializers.glorot_normal(seed=seed)
    elif init_op == "glorot_uniform":
        return tf.contrib.keras.initializers.glorot_uniform(seed=seed)
    else:
        raise ValueError("Unknown init_op %s" % init_op)

def create_embbeding(vocab_size, embed_size, dtype=tf.float32, scope=None):
    """Create embedding matrix for both encoder and decoder."""
    with tf.variable_scope(scope or "embeddings", dtype=dtype):
        embedding = tf.get_variable("embedding", [vocab_size, embed_size], dtype)

    return embedding

def _single_cell(num_units, keep_prob, device_str=None):
    """Create an instance of a single RNN cell."""
    single_cell = tf.contrib.rnn.GRUCell(num_units)

    if keep_prob < 1.0:
        single_cell = tf.contrib.rnn.DropoutWrapper(cell=single_cell, input_keep_prob=keep_prob)

    # Device Wrapper
    if device_str:
        single_cell = tf.contrib.rnn.DeviceWrapper(single_cell, device_str)

    return single_cell
```

modelhelper.py

```python

def create_rnn_cell(num_units, num_layers, keep_prob):
    """Create multi-layer RNN cell."""
    cell_list = []
    for i in range(num_layers):
        single_cell = _single_cell(num_units=num_units, keep_prob=keep_prob)
        cell_list.append(single_cell)

    if len(cell_list) == 1:  # Single layer.
        return cell_list[0]
    else:  # Multi layers
        return tf.contrib.rnn.MultiRNNCell(cell_list)

def gradient_clip(gradients, max_gradient_norm):
    """Clipping gradients of a model."""
    clipped_gradients, gradient_norm = tf.clip_by_global_norm(gradients, max_gradient_norm)
    gradient_norm_summary = [tf.summary.scalar("grad_norm", gradient_norm)]
    gradient_norm_summary.append(
        tf.summary.scalar("clipped_gradient", tf.global_norm(clipped_gradients)))

    return clipped_gradients, gradient_norm_summary
```

pattemutils.py

```python
import re

#This file is only used at inference time.
def check_patterns_and_replace(question):
    pat_matched, new_sentence, para_list = _check_arithmetic_pattern_and_replace(question)

    if not pat_matched:
        pat_matched, new_sentence, para_list = _check_not_username_pattern_and_replace(new_sentence)

    if not pat_matched:
        pat_matched, new_sentence, para_list = _check_username_callme_pattern_and_replace(new_sentence)

    return pat_matched, new_sentence, para_list


def _check_arithmetic_pattern_and_replace(sentence):
    pat_matched, ind_list, num_list = _contains_arithmetic_pattern(sentence)
    if pat_matched:
        s1, e1 = ind_list[0]
        s2, e2 = ind_list[1]
        # Leave spaces around the special tokens so that NLTK knows they are separate tokens
        new_sentence = sentence[:s1] + ' _num1_ ' + sentence[e1:s2] + ' _num2_ ' + sentence[e2:]
        return True, new_sentence, num_list
    else:
        return False, sentence, num_list

def _contains_arithmetic_pattern(sentence):
    numbers = [
        "zero", "one", "two", "three", "four", "five", "six", "seven",
        "eight", "nine", "ten", "eleven", "twelve", "thirteen", "fourteen",
        "fifteen", "sixteen", "seventeen", "eighteen", "nineteen",
        "twenty", "thirty", "forty", "fifty", "sixty", "seventy", "eighty", "ninety",
        "hundred", "thousand", "million", "billion", "trillion"]

    pat_op1 = re.compile(
        r'\s(plus|add|added|\+|minus|subtract|subtracted|-|times|multiply|multiplied|\*|divide|(divided\
        re.IGNORECASE)
    pat_op2 = re.compile(r'\s((sum\s+of)|(product\s+of))\s', re.IGNORECASE)
    pat_as = re.compile(r'((\bis\b)|=|(\bequals\b)|(\bget\b))', re.IGNORECASE)

    mat_op1 = re.search(pat_op1, sentence)
    mat_op2 = re.search(pat_op2, sentence)
    mat_as = re.search(pat_as, sentence)
```

pattemutils.py

```python
    if (mat_op1 or mat_op2) and mat_as:  # contains an arithmetic operator and an assign operator
        # Replace all occurrences of word "and" with 3 whitespaces before feeding to
        # the pattern matcher.
        pat_and = re.compile(r'\band\b', re.IGNORECASE)
        if mat_op1:
            tmp_sentence = pat_and.sub('   ', sentence)
        else:  # Do not support word 'and' in the English numbers any more as that can be ambiguous.
            tmp_sentence = pat_and.sub('_T_', sentence)

        number_rx = r'(?:{})'.format('|'.join(numbers))
        pat_num = re.compile(r'\b{0}(?:(?:\s+(?:and\s+)?|-){0})*\b|\d+'.format(number_rx),
                             re.IGNORECASE)
        ind_list = [(m.start(0), m.end(0)) for m in re.finditer(pat_num, tmp_sentence)]
        num_list = []
        if len(ind_list) == 2:  # contains exactly two numbers
            for start, end in ind_list:
                text = sentence[start:end]
                text_int = _text2int(text)
                if text_int == -1:
                    return False, [], []
                num_list.append(text_int)
            return True, ind_list, num_list

    return False, [], []

def _check_not_username_pattern_and_replace(sentence):
    import nltk

    tokens = nltk.word_tokenize(sentence)
    tmp_sentence = ' '.join(tokens[:]).strip()
    pat_not_but = re.compile(r'(\s|^)my\s+name\s+is\s+(not|n\'t)\s+(.+?)(\s\.|\s,|\s!)\s*but\s+(.+?)(\s\.|\s,
                             re.IGNORECASE)
    mat_not_but = re.search(pat_not_but, tmp_sentence)
    pat_not = re.compile(r'(\s|^)my\s+name\s+is\s+(not|n\'t)\s+(.+?)(\s\.|\s,|\s!|$)', re.IGNORECASE)
    mat_not = re.search(pat_not, tmp_sentence)
    para_list = []
    found = 0
    if mat_not_but:
        wrong_name = mat_not_but.group(3).strip()
        correct_name = mat_not_but.group(5).strip()
        para_list.append(correct_name)
        new_sentence = sentence.replace(wrong_name, ' _ignored_ ', 1).replace(correct_name, ' _name_ ', 1)
        # print("User name is not: {}, but {}.".format(wrong_name, correct_name))
        found += 1
    elif mat_not:
        wrong_name = mat_not.group(3).strip()
        new_sentence = sentence.replace(wrong_name, ' _ignored_ ', 1)
        # print("User name is not: {}.".format(wrong_name))
        found += 1
    else:
        new_sentence = sentence
        # print("Wrong name not found.")

    if found >= 1:
        return True, new_sentence, para_list
    else:
        return False, sentence, para_list

def _check_username_callme_pattern_and_replace(sentence):
    tokens = nltk.word_tokenize(sentence)
    tmp_sentence = ' '.join(tokens[:]).strip()
    pat_name = re.compile(r'(\s|^)my\s+name\s+is\s+(.+?)(\s\.|\s,|\s!|$)', re.IGNORECASE)
    pat_call = re.compile(r'(\s|^)call\s+me\s+(.+?)(\s(please|pls))?(\s\.|\s,|\s!|$)', re.IGNORECASE)
    mat_name = re.search(pat_name, tmp_sentence)
    mat_call = re.search(pat_call, tmp_sentence)

    para_list = []
    found = 0
    if mat_name:
        user_name = mat_name.group(2).strip()
        para_list.append(user_name)
        new_sentence = sentence.replace(user_name, ' _name_ ', 1)
        found += 1
    else:
        para_list.append('')  # reserve the slot
        new_sentence = sentence
```

search.py

```python
import urllib
import requests
from bs4 import BeautifulSoup

def get_html(url):
    headers = {'User-Agent':
                    'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
    soup_html = BeautifulSoup(requests.get(url=url, headers=headers).content, "lxml")
    return soup_html

# Wikipedia
def get_wikipedia(message):
    url = 'https://www.britannica.com/biography/Albert-Einstein'
    soup_html = get_html(url)

    title = soup_html.find('h1').get_text().strip()
    topic = soup_html.find(class_='topic-identifier').get_text().strip()
    return title + ' is a ' + topic

# Britannica
def get_britannica(message):
    url = 'https://www.britannica.com/biography/Albert-Einstein'
    soup_html = get_html(url)

    title = soup_html.find('h1').get_text().strip()
    topic = soup_html.find(class_='topic-identifier').get_text().strip()
    return title + ' is a ' + topic

# Get a joke
def get_joke():
    url = 'https://bestlifeonline.com/text-jokes/'
    soup_html = get_html(url)

    contents = soup_html.select('.content')
    from random import choice
    content = choice(contents)
    result = content.get_text().strip()
    return result
```

sessiondata.py

```python
"""
This class is only used at inference time.
In the case of a production system, the SessionData has to be maintained so that ChatSessior
can expire and then be cleaned from the memory.
"""
class SessionData:
    def __init__(self):
        self.session_dict = {}

    def add_session(self):
        items = self.session_dict.items()
        if items:
            last_id = max(k for k, v in items)
        else:
            last_id = 0
        new_id = last_id + 1

        self.session_dict[new_id] = ChatSession(new_id)
        return new_id

    def get_session(self, session_id):
        return self.session_dict[session_id]
```

sessiondata.py

```python

class ChatSession:
    def __init__(self, session_id):
        self.session_id = session_id
        self.howru_asked = False
        self.user_name = None
        self.call_me = None
        self.last_question = None
        self.last_answer = None
        self.update_pair = True
        self.last_topic = None
        self.keep_topic = False

        # Will be storing the information of the pending action:
        # The action function name, the parameter for answer yes, and the parameter for answer no.
        self.pending_action = {'func': None, 'Yes': None, 'No': None}

    def before_prediction(self):
        self.update_pair = True
        self.keep_topic = False

    def after_prediction(self, new_question, new_answer):
        self._update_last_pair(new_question, new_answer)
        self._clear_last_topic()

    def _update_last_pair(self, new_question, new_answer):
        """
        Last pair is updated after each prediction except in a few cases.
        """
        if self.update_pair:
            self.last_question = new_question
            self.last_answer = new_answer

    def _clear_last_topic(self):
        """
        Last topic is cleared after each prediction except in a few cases.
        """
        if not self.keep_topic:
            self.last_topic = None

    def update_pending_action(self, func_name, yes_para, no_para):
        self.pending_action['func'] = func_name
        self.pending_action['Yes'] = yes_para
        self.pending_action['No'] = no_para

    def clear_pending_action(self):
        """
        Pending action is, and only is, cleared at the end of function: execute_pending_action_and_reply.
        """
        self.pending_action['func'] = None
        self.pending_action['Yes'] = None
        self.pending_action['No'] = None

```

tokenizeddata.py

```python
import codecs
import os
import tensorflow as tf
from collections import namedtuple
from tensorflow.python.ops import lookup_ops
from chatbot.hparams import HParams

COMMENT_LINE_STT = "#=="
CONVERSATION_SEP = "==="
AUG0_FOLDER = "Augment0"
AUG1_FOLDER = "Augment1"
AUG2_FOLDER = "Augment2"
MAX_LEN = 1000  # Assume no line in the training data is having more than this number of characters
VOCAB_FILE = "vocab.txt"

```

tokenizeddata.py

```python
class TokenizedData:
    def __init__(self, corpus_dir, hparams=None, training=True, buffer_size=8192):
        """
        Args:
            corpus_dir: Name of the folder storing corpus files for training.
            hparams: The object containing the loaded hyper parameters. If None, it will be
                    initialized here.
            training: Whether to use this object for training.
            buffer_size: The buffer size used for mapping process during data processing.
        """
        if hparams is None:
            self.hparams = HParams(corpus_dir).hparams
        else:
            self.hparams = hparams

        self.src_max_len = self.hparams.src_max_len
        self.tgt_max_len = self.hparams.tgt_max_len
        self.training = training
        self.text_set = None
        self.id_set = None
        vocab_file = os.path.join(corpus_dir, VOCAB_FILE)
        self.vocab_size, _ = check_vocab(vocab_file)
        self.vocab_table = lookup_ops.index_table_from_file(vocab_file,
                                                            default_value=self.hparams.unk_id)
        if training:
            self.case_table = prepare_case_table()
            self.reverse_vocab_table = None
            self._load_corpus(corpus_dir)
            self._convert_to_tokens(buffer_size)
        else:
            self.case_table = None
            self.reverse_vocab_table = \
                lookup_ops.index_to_string_table_from_file(vocab_file,
                                                           default_value=self.hparams.unk_token)

    def get_training_batch(self, num_threads=4):
        assert self.training
        buffer_size = self.hparams.batch_size * 400
        # Comment this line for debugging.
        train_set = self.id_set.shuffle(buffer_size=buffer_size)

        # Create a target input prefixed with BOS and a target output suffixed with EOS.
        # After this mapping, each element in the train_set contains 3 columns/items.
        train_set = train_set.map(lambda src, tgt:
                                  (src, tf.concat(([self.hparams.bos_id], tgt), 0),
                                   tf.concat((tgt, [self.hparams.eos_id]), 0)),
                                  num_parallel_calls=num_threads).prefetch(buffer_size)

        # Add in sequence lengths.
        train_set = train_set.map(lambda src, tgt_in, tgt_out:
                                  (src, tgt_in, tgt_out, tf.size(src), tf.size(tgt_in)),
                                  num_parallel_calls=num_threads).prefetch(buffer_size)

        def batching_func(x):
            return x.padded_batch(
                self.hparams.batch_size,
                # The first three entries are the source and target line rows, these have unknown-length
                # vectors. The last two entries are the source and target row sizes, which are scalars.
                padded_shapes=(tf.TensorShape([None]),  # src
                               tf.TensorShape([None]),  # tgt_input
                               tf.TensorShape([None]),  # tgt_output
                               tf.TensorShape([]),      # src_len
                               tf.TensorShape([])),     # tgt_len
                # Pad the source and target sequences with eos tokens. Though we don't generally need to
                # do this since later on we will be masking out calculations past the true sequence.
                padding_values=(self.hparams.eos_id,  # src
                                self.hparams.eos_id,  # tgt_input
                                self.hparams.eos_id,  # tgt_output
                                0,       # src_len -- unused
                                0))      # tgt_len -- unused

        if self.hparams.num_buckets > 1:
            bucket_width = (self.src_max_len + self.hparams.num_buckets - 1) // self.hparams.num_buckets

            # Parameters match the columns in each element of the dataset.
            def key_func(unused_1, unused_2, unused_3, src_len, tgt_len):
                # Calculate bucket_width by maximum source sequence length. Pairs with length [0, bucket_width)
                # go to bucket 0, length [bucket_width, 2 * bucket_width) go to bucket 1, etc. Pairs with
                # length over ((num_bucket-1) * bucket_width) words all go into the last bucket.
                # Bucket sentence pairs by the length of their source sentence and target sentence.
                bucket_id = tf.maximum(src_len // bucket_width, tgt_len // bucket_width)
                return tf.to_int64(tf.minimum(self.hparams.num_buckets, bucket_id))
```

tokenizeddata.py

```python
            # No key to filter the dataset. Therefore the key is unused.
            def reduce_func(unused_key, windowed_data):
                return batching_func(windowed_data)

            batched_dataset = train_set.apply(
                tf.contrib.data.group_by_window(key_func=key_func,
                                                reduce_func=reduce_func,
                                                window_size=self.hparams.batch_size))
        else:
            batched_dataset = batching_func(train_set)

        batched_iter = batched_dataset.make_initializable_iterator()
        (src_ids, tgt_input_ids, tgt_output_ids, src_seq_len, tgt_seq_len) = (batched_iter.get_next())

        return BatchedInput(initializer=batched_iter.initializer,
                            source=src_ids,
                            target_input=tgt_input_ids,
                            target_output=tgt_output_ids,
                            source_sequence_length=src_seq_len,
                            target_sequence_length=tgt_seq_len)

    def get_inference_batch(self, src_dataset):
        text_dataset = src_dataset.map(lambda src: tf.string_split([src]).values)

        if self.hparams.src_max_len_infer:
            text_dataset = text_dataset.map(lambda src: src[:self.hparams.src_max_len_infer])
        # Convert the word strings to ids
        id_dataset = text_dataset.map(lambda src: tf.cast(self.vocab_table.lookup(src),
                                                          tf.int32))
        if self.hparams.source_reverse:
            id_dataset = id_dataset.map(lambda src: tf.reverse(src, axis=[0]))
        # Add in the word counts.
        id_dataset = id_dataset.map(lambda src: (src, tf.size(src)))

        def batching_func(x):
            return x.padded_batch(
                self.hparams.batch_size_infer,
                # The entry is the source line rows; this has unknown-length vectors.
                # The last entry is the source row size; this is a scalar.
                padded_shapes=(tf.TensorShape([None]), tf.TensorShape([])), # src_len
                # Pad the source sequences with eos tokens. Though notice we don't generally need to
                # do this since later on we will be masking out calculations past the true sequence.
                padding_values=(self.hparams.eos_id,  0))                   # src_len -- unused

        id_dataset = batching_func(id_dataset)
        infer_iter = id_dataset.make_initializable_iterator()
        (src_ids, src_seq_len) = infer_iter.get_next()

        return BatchedInput(initializer=infer_iter.initializer,
                            source=src_ids,
                            target_input=None,
                            target_output=None,
                            source_sequence_length=src_seq_len,
                            target_sequence_length=None)

    def _load_corpus(self, corpus_dir):
        for fd in range(2, -1, -1):
            file_list = []
            if fd == 0:
                file_dir = os.path.join(corpus_dir, AUG0_FOLDER)
            elif fd == 1:
                file_dir = os.path.join(corpus_dir, AUG1_FOLDER)
            else:
                file_dir = os.path.join(corpus_dir, AUG2_FOLDER)

            for data_file in sorted(os.listdir(file_dir)):
                full_path_name = os.path.join(file_dir, data_file)
                if os.path.isfile(full_path_name) and data_file.lower().endswith('.txt'):
                    file_list.append(full_path_name)

            assert len(file_list) > 0
            dataset = tf.data.TextLineDataset(file_list)
            src_dataset = dataset.filter(lambda line:
                                         tf.logical_and(tf.size(line) > 0,
                                                        tf.equal(tf.substr(line, 0, 2), tf.constant('Q:'))))
```

tokenizeddata.py

```python
            src_dataset = src_dataset.map(lambda line:
                                          tf.substr(line, 2, MAX_LEN)).prefetch(4096)
            tgt_dataset = dataset.filter(lambda line:
                                         tf.logical_and(tf.size(line) > 0,
                                                        tf.equal(tf.substr(line, 0, 2), tf.constant('A:')))
            tgt_dataset = tgt_dataset.map(lambda line:
                                          tf.substr(line, 2, MAX_LEN)).prefetch(4096)
            src_tgt_dataset = tf.data.Dataset.zip((src_dataset, tgt_dataset))
            if fd == 1:
                src_tgt_dataset = src_tgt_dataset.repeat(self.hparams.aug1_repeat_times)
            elif fd == 2:
                src_tgt_dataset = src_tgt_dataset.repeat(self.hparams.aug2_repeat_times)

            if self.text_set is None:
                self.text_set = src_tgt_dataset
            else:
                self.text_set = self.text_set.concatenate(src_tgt_dataset)

        def _convert_to_tokens(self, buffer_size):
            # The following 3 steps act as a python String lower() function
            # Split to characters
            self.text_set = self.text_set.map(lambda src, tgt:
                                              (tf.string_split([src], delimiter='').values,
                                               tf.string_split([tgt], delimiter='').values)
                                              ).prefetch(buffer_size)
            # Convert all upper case characters to lower case characters
            self.text_set = self.text_set.map(lambda src, tgt:
                                              (self.case_table.lookup(src), self.case_table.lookup(tgt))
                                              ).prefetch(buffer_size)
            # Join characters back to strings
            self.text_set = self.text_set.map(lambda src, tgt:
                                              (tf.reduce_join([src]), tf.reduce_join([tgt]))
                                              ).prefetch(buffer_size)

            # Split to word tokens
            self.text_set = self.text_set.map(lambda src, tgt:
                                              (tf.string_split([src]).values, tf.string_split([tgt]).values)
                                              ).prefetch(buffer_size)
            # Remove sentences longer than the model allows
            self.text_set = self.text_set.map(lambda src, tgt:
                                              (src[:self.src_max_len], tgt[:self.tgt_max_len])
                                              ).prefetch(buffer_size)

            # Reverse the source sentence if applicable
            if self.hparams.source_reverse:
                self.text_set = self.text_set.map(lambda src, tgt:
                                                  (tf.reverse(src, axis=[0]), tgt)).prefetch(buffer_size)

            self.id_set = self.text_set.map(lambda src, tgt:
                                            (tf.cast(self.vocab_table.lookup(src), tf.int32),
                                             tf.cast(self.vocab_table.lookup(tgt), tf.int32))
                                            ).prefetch(buffer_size)
def check_vocab(vocab_file):
    """Check to make sure vocab_file exists"""
    if tf.gfile.Exists(vocab_file):
        vocab_list = []
        with codecs.getreader("utf-8")(tf.gfile.GFile(vocab_file, "rb")) as f:
            for word in f:
                vocab_list.append(word.strip())
    else:
        raise ValueError("The vocab_file does not exist. Please run the script to create it.")
    return len(vocab_list), vocab_list

def prepare_case_table():
    keys = tf.constant([chr(i) for i in range(32, 127)])
    l1 = [chr(i) for i in range(32, 65)]
    l2 = [chr(i) for i in range(97, 123)]
    l3 = [chr(i) for i in range(91, 127)]
    values = tf.constant(l1 + l2 + l3)
    return tf.contrib.lookup.HashTable(
        tf.contrib.lookup.KeyValueTensorInitializer(keys, values), ' ')

class BatchedInput(namedtuple("BatchedInput",
                              ["initializer","source","target_input","target_output",
                               "source_sequence_length",
                               "target_sequence_length"])):
    pass
```

load.aiml

```
<aiml version="2.0" encoding="UTF-8">
    <category>
        <pattern>LOAD AIML B</pattern>
        <template>
            <learn>resources/startup.aiml</learn>
            <learn>resources/common.aiml</learn>
            <learn>resources/learn.aiml</learn>
            <learn>resources/save.aiml</learn>
        </template>
    </category>
</aiml>
```

common.aiml

```
<aiml version="2.0" encoding="UTF-8">
    <category>
        <pattern>HI</pattern>
        <template>
                Hi, May I know your name please?
        </template>
    </category>
    <category>
        <pattern>*</pattern>
        <that>HI MAY I KNOW YOUR NAME PLEASE</that>
        <template>
            <think><set name="username"><star /></set></think>
            Nice to meet you, <get name="username"/>. How can I help you?
        </template>
    </category>
    <category>
        <pattern>MY NAME IS *</pattern>
        <template>
            <think><set name="username"><star /></set></think>
            Nice to meet you, <get name="username"/>. How can I help you?
        </template>
    </category>
    <category>
        <pattern>WHAT IS YOUR NAME</pattern>
        <template>
            <random>
                <li>I am a IntelliBot for your assistance. </li>
                <li>I am a robot, called IntelliBot.</li>
                <li>I am your assistance! You can call me: IntelliBot</li>
            </random>
        </template>
    </category>
     <category>
        <pattern>THANK YOU</pattern>
        <template>
            <random>
                <li>You're welcome, <get name="username"/>.</li>
            </random>
        </template>
    </category>
    <category>
        <pattern>BYE</pattern>
        <template>
            <random>
                <li>Bye bye, <get name="username"/></li>
                <li>Goodbye, <get name="username"/></li>
                <li>See you again~ <get name="username"/></li>
            </random>
        </template>
    </category>
</aiml>
```

learn.aiml

```
<aiml version="2.0" encoding="UTF-8">

    <!-- Learning function -->
    <category>
        <pattern>LEARN</pattern>
        <template>Please teach me.</template>
    </category>
</aiml>
```

server.py

```python
import os
import sys
import colorama
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
import logging
from logging.handlers import TimedRotatingFileHandler
from chatbot.chatbot import ChatBot
from flask import Flask, render_template, request
from settings import PROJECT_ROOT

colorama.init()

def init_log(log_file='log/info.log'):

    handler = TimedRotatingFileHandler(log_file, when="D", interval=1, backupCount=7)
    formatter = logging.Formatter('%(asctime)s : %(levelname)s : %(message)s')
    handler.setFormatter(formatter)
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    logger.addHandler(handler)

    return logger

logger = init_log()
bot = ChatBot()
app = Flask(__name__, static_url_path='')

@app.route('/', methods=['GET', 'POST'])
def view():
    return render_template('index.html')

@app.route('/chat', methods=['GET'])
def response():
    data = request.args.to_dict()
    message = data['message']

    if message != '':
        if message.strip() == 'exit' or message.strip() == 'quit':
            answer = 'Thank you for using IntelliBot. GoodBye'
        else:
            answer = bot.response(message)
    return answer

@app.route('/forget', methods=['GET'])
def forget():
    bot.forget()
    return 'success'

if __name__ == '__main__':
    print (bot.response("Server started..."))
    app.run('127.0.0.1', debug=True)
```

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-COMPATIBLE" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>IntelliBot</title>

    <link type="text/css" rel="stylesheet" href="vendor/bootstrap/css/bootstrap.css" >
    <link type="text/css" rel="stylesheet" href="medius/css/medius-style.css">
    <link type="text/css" rel="stylesheet" href="vendor/fontawesome-free/css/all.min.css">
    <link type="text/css" rel="stylesheet" href="vendor/foundation-icons/foundation-icons.css">

    <script type="text/javascript" src="vendor/jquery/jquery-3.3.1.min.js"></script>
    <script type="text/javascript" src="medius/js/chat.js"></script>

    <style>
        body, html {
            height: 100%;
        }
        .bg  {
            background-image: url("medius/images/bg.png");
            height: 100%;
            background-repeat: no-repeat;
            background-size: cover;
        }
        .bgChat  {
            background-image: url("medius/images/chat-bg.png");
            height: 100%;
            background-repeat: no-repeat;
        }
    </style>
</head>

<body>
<nav class="navbar navbar-expand nav-header">
    <a class="navbar-brand mr-1" href="index.html">IntelliBot : A Dialogue-based Insurance Chatbot</a>
    <div class="d-none d-md-inline-block form-inline ml-auto mr-0 mr-md-3 my-2 my-md-0">
        <ul class="navbar-nav ml-auto ml-md-0">
            <a class="nav-link" href="#">Assessments</a>
            <a class="nav-link" href="#">Profile</a>
            <a class="nav-link" href="#">Account</a>
        </ul>
    </div>
</nav>

<div class="bg">
    <div class="col-md-4 col-md-offset-4">
        <div class="card card-chatbot mx-auto">
            <div class="card-header text-center"><strong>Ask me anything you like</strong></div>
            <div class="card-body bgChat">
                <div class="speak_box">
                    <div class="answer">
                        <div class="answer_text">
                            <p>Welcome to IntelliBot. I am here to answer your queries.</p>
                        </div>
                    </div>
                </div>
                <div class="text-left">
                    <a class="d-block mt-3">Type your message</a>
                </div>
                <div class="right-inner-addon chat_box">
                    <i class="fi-magnifying-glass"></i>
                    <input class="form-control" type="text" onKeyUp="key_up()"/>
                </div>
            </div>
        </div>
    </div>
</div>
```

```
70
71  <script type="text/javascript">
72      $(function () {
73          $(document).keydown(function (event) {
74              if (event.keyCode == 13) {
75                  send_message();
76              }
77          });
78
79          $('.chat_box input').focus();
80      });
81  </script>
82
83  </body>
84  </html>
```