Chris Wilson, Jeff Levesque

cwilson@syr.edu, jlevesqu@syr.edu | ist-664

IST-664: Chatbot

Professor Mccracken

**Introduction**

Conversational agents often collect a combination of input, then return some predetermined set of response. The most basic strategy for this approach is defining a set of rules. As input enters the system, the application attempts to bin the response based on a hierarchy of definitions. The simplest rules-based approach can often be expanded by training a classification model based on similar sentences or parts of speech. However, without a large dataset to train many different combinations of models, the responses can often be found incorrect, predictable or repetitive.

To arrive at a more robust model, a combination of different techniques can be implemented. First, rather than applying on a rules-based response, a recurrent neural network (RNN) can be employed for a sequence to sequence training. Specifically, a sentence sequence would be matched with a response sequence. With enough sentence pairs, a neural network can learn to generate varying types of responses. This differs from the classifier approach, since the combining layers of a neural network would predict the best sequence of words, rather than having a large inventory of predictable responses.

A rules-based, coupled with a neural network could greatly enhance the conversational agent. For example, a proxy could be implemented before the chatbot initiates. Specifically, the classifier could predict the genre for a given input, then upon prediction engage the respective neural network model. More sophisticated models could further expand by applying generative learning. Rather than training the model once, the application could iteratively update the model, by training the additional new data.

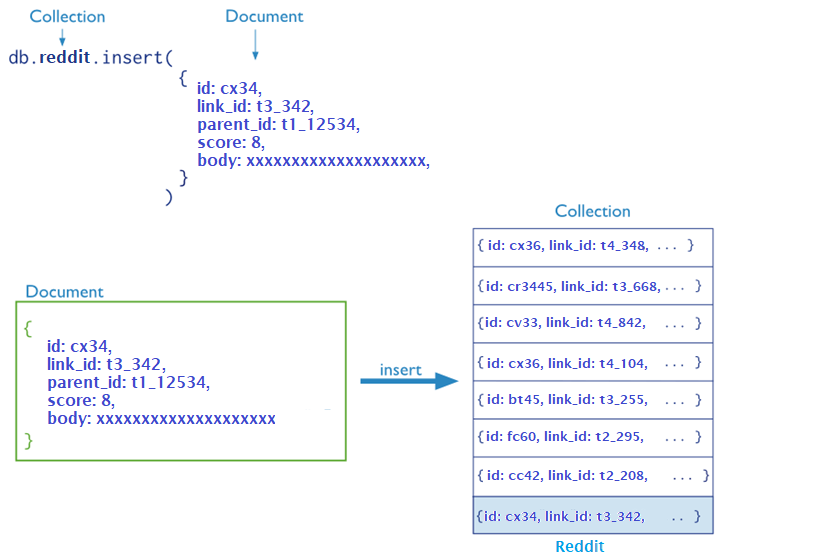
In this project, an attempt is made to create two rules-based classifiers around the main RNN conversational agent. Specifically, a classifier is created as a proxy to predict whether an input sentence is a question. If the classifier predicts a given sentence is a question, the RNN model is generated, then returns a response. Each response includes a score. If the returned sentence does not return an acceptable score, a secondary classifier attempts to predict what StackOverflow channel is best suited for the original question.

**Database**

This project implements mongodb as database framework based on three criteria’s:

* Less application code
* High Availability
* Distributed datastore

By implementing mongodb, the amount of coding syntax is greatly reduced. Specifically, a predefined schema is not needed, and the application does not need to segregate queries on structural requirements.



**Figure 1**. Reddit insert, and overall document-collection representation. The exact code adaptation can be reviewed in Appendix A below.

Additionally, mongodb allows high availability without additional packages. This allows the replicated instance to contain a primary database, with secondary nodes replicating identical changes. The benefit of this layout allows fault tolerance where the primary node may become unavailable, thereby electing a secondary node to be the new primary database:

A close up of text on a black background

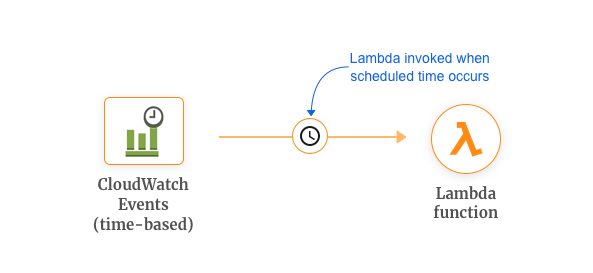
Description automatically generated

**Figure 2**. Replicated mongodb setup containing a single primary with two secondary nodes. The corresponding implementation can be reviewed in Appendix B below.

Next, each replicated instance can be distributed across multiple shards, commonly known as clustering. This strategy allows for large data storage, since data can be distributed across multiple databases on various machines. Additionally, each database instance may incorporate its storage to be entirely memory based[[1]](#footnote-1).

**Development**

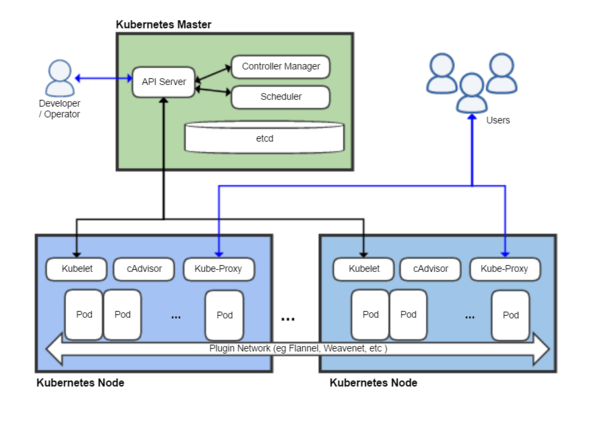
The project initially required eight virtual machines for each mongodb component. Additionally, a puppetserver was implemented as an additional layer of configuration management, along with an amazon web services (aws) p2.xlarge[[2]](#footnote-2) compute resource. Utility scripts were created for the purpose of the latter infrastructure[[3]](#footnote-3). However, a single VPC on aws only provided 5 elastic ip addresses[[4]](#footnote-4). This introduced complexity, and a solution was needed to reduce the cost associated with machine runtime. Therefore, lambda scripts[[5]](#footnote-5) were created, to ensure virtual machines were powered off at specific times of the week.



**Figure 3**. Simple lamda functions could decrease run time, by running scheduled jobs. Associated code can be reviewed in Appendix C below.

However, since five virtual machines did not have elastic ip addresses, their public ip addresses constantly changed upon machine startup. This posed challenges, and manual intervention were required to ensure network security groups were properly defined. Specifically, each mongodb blocked all incoming ports except other mongodb instances within it’s sharded system. Due to the complexity, this infrastructure was abandoned, but corresponding scripts to deploy this system is still available and version controlled.

To reduce the environment, a local docker-compose.yml[[6]](#footnote-6) successfully deployed single replicated shard as a proof of concept. Furthermore, the dockerized concept can easily be scaled out using kubernetes to deploy across networks containing kubelets:



**Figure 3**. Kubernetes architecture diagram[[7]](#footnote-7).

Using kubernetes could additionally solve the earlier problem, if multiple kubernete pods were deployed within each ec2 instance. However, since a cluster was successfully deployed, the remaining efforts of this study scaled down to a single virtualbox machine configured using vagrant[[8]](#footnote-8). Specifically, large amounts of data were not required to satisfy a proof of concept. Therefore, only a single mongodb database was utilized in the same machine containing the compute logic.

**Data Preparation: chatbot**

The first 6 months of reddit data[[9]](#footnote-9) was collected and used. Each monthly json file consists of numerous json strings. However, not all key-values were important. This was filtered using the mapreduce feature. Specifically, a map function emitted values to be aggregated by the reduce function. Out of all key attributes from the json string, the highlighted values were emitted from the map to the reduce function, if the body attribute was not a repost, deleted, or removed:

{

"author\_flair\_text": null,

"author": "[deleted]",

"id": "c57tk",

"edited": false,

"parent\_id": "t1\_c56p8",

"gilded": 0,

"retrieved\_on": 1473823533,

"distinguished": null,

"body": "Oooh. That was great too.",

"controversiality": 0,

"stickied": false,

"link\_id": "t3\_56lg",

"subreddit\_id": "t5\_6",

"subreddit": "reddit.com",

"created\_utc": 1146460515,

"author\_flair\_css\_class": null,

"score": 3,

"ups": 3

}

**Figure 4**. One of many json strings found in each dataset. The corresponding map function, which emits a subset of the above keys can be reviewed in Appendix D below.

After the map emits specific keys based on an aggregated link\_id, the reduce function further removes any reddit instance, where the body matches the following regular expressions:

const singleton = ['.', '-', '$', ',', ':', '%', "'"]

const regexParts = [

/\s+/,

/\]|\[|\(|\)/,

/-&gt;|&gt;|&lt;/,

/\$?[0-9]{6,}/,

/https?:\/\/|https?;\/\//,

/((?:www\.|(?!www)|[a-zA-Z]+\.)[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,}|www\.[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,})/,

/((?:www\.|(?!www))[a-zA-Z0-9]\.[^\s]{2,}|www\.[a-zA-Z0-9]\.[^\s]{2,})/,

/[^\x01-\x7F]+/,

/--|\\*|\.\.\.|"|:-|:|!!!|\?\?\?|\+|=|\/\/|;/,

/\s(\.-\$,:%')\s/

],

regexString = regexParts.map(function(x){return x.source}).join('|'),

tokenRegex = new RegExp(regexString, 'g');

**Figure 5**. Regular expression used by the reduce function. The full reduce implementation can be reviewed in Appendix E below.

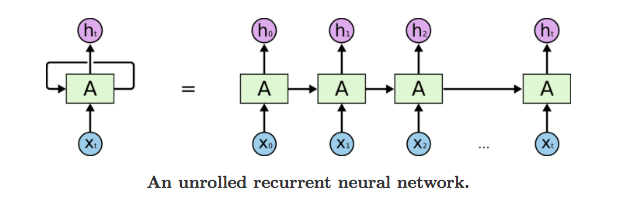
This removes special, and non-ascii characters, as well as urls from the body attribute. Once completed, the map function takes each body value from each json string, aggregated by a common link\_id, and checks whether the given parent\_id matches another reddit json string id. This requires a nested loop to iterate the same array of json string aggregated by link\_id. Once completed, the map function returns all instances of matching reddit json strings. Specifically, if a json string contains a parent\_id that matches another json string’s id, then the former string body attribute is appended to the comments array, while the latter is appended to the posts array.

**Data Preparation: question classifier**

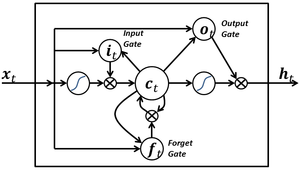
A second type of dataset was collected containing question-answer pairs[[10]](#footnote-10). This was obtained from an NSF Graduate Research Fellowship at Carnegie Mellon University[[11]](#footnote-11) (CMU). Each dataset file composed of multiple tab delimited values. However, in this study only the Question, and Answer columns were aggregated into a common dataframe.

**Results**

A subset of the originally provided reddit data was inserted into mongodb. However, since the proof of concept was greatly reduced, only a portion of the first month[[12]](#footnote-12) was used to train an NMT recurrent neural network. Specifically, using an NMT tensorflow based framework[[13]](#footnote-13), a long short-term memory (LSTM) was trained:



**Figure 6**. Recurrent neural network visualizes how each step the network takes an input xi and hi−1, then generates a hidden state hi. Additionally, at each step i, the hidden state hi is updated[[14]](#footnote-14).



**Figure 7**. LSTM minimizes vanishing gradient error, by product of weights shrinking in value for each backward propagation iteration.

The earlier mapreduce results needed to output the corresponding comments, and posts to be individual files defined by NMT[[15]](#footnote-15). Specifically, each sentence is split as a newline in it’s corresponding [to|from] file. Then, a supplied prepare\_data.py, converts the newline separated sentence file into required bpe files used by the train.py[[16]](#footnote-16). Though, the resulting chatbot was executable, it was largely disappointing. Furthermore, costs associated with AWS compute, along with project timeline eliminated the option of reinvestigating the use of p2.xlarge on aws. Since a very simplified model was achieved, a generalized model (i.e. chatbot demo) was downloaded and implemented for the remaining project[[17]](#footnote-17).

Next, classifiers were trained and implemented as augmented components to the conversational agents. First, using the earlier CMU dataset, three dataset files were merged into a single dataframe. Each question and corresponding answer were broken down to parts of speech (pos). However, implementing a nested set() in the list comprehension ensures that only unique pos combination are retained in the resulting array:

*## (1) pos: combine questions + answers*

questions\_pos = questions.apply(tokenizer)

questions\_unique = [replace(list(x), penn\_scale()) **for** x **in** set(tuple(x) **for** x **in** questions\_pos)]

answers\_pos = answers.apply(tokenizer)

answers\_unique = [replace(list(x), penn\_scale()) **for** x **in** set(tuple(x) **for** x **in** answers\_pos)]

**Figure 6**. List comprehension tokenizes a series of sentences, then returns the associated codified penn tree values. The codification can be reviewed in Appendix G below.

After successfully codifying the parts of speech with associated numeric values, the questions, and answers array are merged into a dataframe. Specifically, the first column represents an array of pos, and a second array corresponds to the sentiment type.

1409 [37, 29, 4, 8, 15, 15, 29]

2284 [35, 29, 22, 8, 7, 30, 14, 7, 14]

245 [5, 32, 8, 8, 14, 7, 15, 2, 15, 15]

2245 [15, 15, 29, 7, 4, 8, 13, 7, 15, 7, 15, 7, 3]

2227 [35, 33, 4, 13, 26, 28, 14]

824 [10]

1649 [37, 29, 15, 13, 13, 7, 15]

646 [4, 10, 13, 7, 4, 13, 7, 13]

559 [4, 15, 13, 33, 31, 7, 15, 15]

517 [4, 32, 14, 30, 4, 14, 2, 14, 8, 14, 32, 21, 3...

Name: pos, dtype: object

**Figure 7**. An examination of the merged dataframe with a 0.20 test split. The associated code to generate the codified train, can be reviewed in Appendix H below.

Since matrices and corresponding label vectors need match in dimension, the number of columns for the input matrix was fixed to 40. Therefore, cells with no values were padded a trivial value of 1:

0 1 2 3 4 5 6 7 8 9 ... 30 31 32 33 34 35 36 37 38 39

0 7 4 15 15 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

1 15 15 13 15 15 15 7 4 3 13 ... 1 1 1 1 1 1 1 1 1 1

2 15 7 4 13 2 15 7 4 13 1 ... 1 1 1 1 1 1 1 1 1 1

3 15 15 15 28 21 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

4 19 12 28 13 8 13 2 13 13 1 ... 1 1 1 1 1 1 1 1 1 1

5 33 15 4 13 13 7 15 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

6 4 13 33 31 7 4 13 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

7 33 4 13 13 8 26 4 8 13 1 ... 1 1 1 1 1 1 1 1 1 1

8 33 15 31 7 15 15 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

9 37 8 14 21 32 7 4 8 13 1 ... 1 1 1 1 1 1 1 1 1 1

[10 rows x 40 columns]

**Figure 8**. Normalized train dataset fixed at 40 columns. The associated code to generate the codified train, can be reviewed in Appendix I below.

0 1 2 3 4 5 6 7 8 9 ... 30 31 32 33 34 35 36 37 38 39

0 4 13 7 14 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

1 15 14 32 21 3 26 3 14 8 1 ... 1 1 1 1 1 1 1 1 1 1

2 15 8 14 21 8 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

3 35 33 4 13 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

4 8 14 7 13 33 31 7 4 8 13 ... 1 1 1 1 1 1 1 1 1 1

5 37 29 15 28 15 14 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

6 4 10 13 7 4 13 7 13 1 1 ... 1 1 1 1 1 1 1 1 1 1

7 33 4 13 18 13 13 8 14 1 1 ... 1 1 1 1 1 1 1 1 1 1

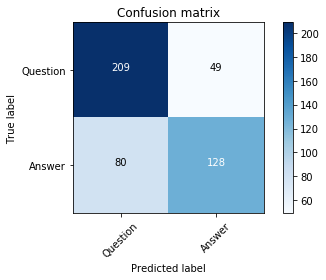
8 8 26 28 4 8 14 7 8 13 1 ... 1 1 1 1 1 1 1 1 1 1

9 4 21 4 14 32 14 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1

[10 rows x 40 columns]

**Figure 8**. Normalized test dataset fixed at 40 columns. The associated code to generate the codified train, can be reviewed in Appendix I below.

A random forest classifier is created using 1500 estimators, with predictions roughly 72% accurate. The corresponding train time was 5.05s, and a prediction time of 0.288s.



**Figure 9**. The confusion matrix for corresponding random forest classifier. The associated code to generate the model, can be reviewed in Appendix J below.

Finally, a StackOverflow (SO) classifer was created[[18]](#footnote-18). Results from this research indicates that a classifier can generally predict the best suited SO channel for a sentence with a 60-73.5% accuracy:

A screenshot of a social media post

Description automatically generated

**Figure 10**. Confusion matrix predicting stackoverflow channel given a sentence.

Given the above results, if the provided sentence is predicted a question, the corresponding classifier starts the conversational agent. The LSTM agent then attempts to predict a corresponding sentence given its input question. However, multiple responses can be returned, each containing its own confidence score. To be more practical, the secondary classifier is implemented by the conversational agent, if the highest scoring response does not meet a predefined threshold. Therefore, if the question proposed to the conversational agent does not have an adequate response, the SO classifier will attempt to predict what stackoverflow channel best suites the original inquiry.

The following is an example session of the conversational agent within the local virtualbox environment:

root@development:/vagrant# python3 run.py

Starting interactive mode (first response will take a while):

> hey, where is billy?

/usr/local/lib/python3.5/dist-packages/sklearn/base.py:251: UserWarning: Trying to unpickle estimator DecisionTreeClassifier from version 0.20.1 when using version 0.20.0. This might lead to breaking code or invalid results. Use at your own risk.

  UserWarning)

/usr/local/lib/python3.5/dist-packages/sklearn/base.py:251: UserWarning: Trying to unpickle estimator RandomForestClassifier from version 0.20.1 when using version 0.20.0. This might lead to breaking code or invalid results. Use at your own risk.

  UserWarning)

hey jimmy, maybe checkout ['math']

>

**Figure 11**. Example session with the conversational agent. The run.py entrypoint script can be further reviewed in Appendix K below.

**Conclusions**

Overall this study has touched multiple areas including general infrastructure, brief passing into recurrent neural network using LSTM, along with concepts of ensembling multiple classifiers around the primary RNN based conversational agent. Given more time, an interesting extension for this project would be to better encompass the neural network model. Specifically, reverse engineering the NMT framework to better tailor the needs for this application. This includes the implementation of generative learning[[19]](#footnote-19).

Despite the earlier neural network not producing better quality responses, it is interesting to note the adaptation of the NMT framework within the same language. Another aspect of improvements could potentially involve additional proxy classifiers. This could also imply multiple trained neural networks which could behave as subject matter experts for a given conversational category. Specifically, a series of classifiers could determine the best suited LSTM conversational agent to deploy. Corrections, and error minimization could potentially involve the integration of sentiment analysis, along with other classifiers which attempts to predict whether a question is resolved.

The area of natural language processing is a rich field. Sometimes models for a specific task can be integrated for a more general ensembled solution. Though findings in this study are not novel, it does provide some basic insight for combining different approaches. As data science becomes a defacto necessity, one wonders whether reusing multiple trained models becomes a standard for solving what today seems as an involved, or complicated problem set.

**Appendix A**

Scan given data directory, and append each json line as an array for bulk insert:

def insert(client, database, collection, data\_directory='data'):

# database + collection

db = client[database]

col = db[collection]

# insert data

data = []

for f in listdir(data\_directory):

file = join(data\_directory, f)

if isfile(file):

with open(file) as fp:

for line in fp.readlines():

# verify valid json

try:

valid = json.loads(line)

data.append(valid)

print('insert: {}'.format(line))

except ValueError as e:

valid = False

print('Not valid json: {}'.format(e))

# insert to mongodb

if data:

post\_id = col.insert\_many(data).inserted\_ids

print('post\_id: {}'.format(post\_id))

**Appendix B**

A subset of docker-compose.yml containing a single replicaset:

version: '3'

services:

mongorsn1:

container\_name: mongors1n1

image: mongo

command: mongod --shardsvr --replSet mongors1 --dbpath /data/db --port 27017

ports:

- 27017:27017

expose:

- "27017"

environment:

TERM: xterm

volumes:

- /etc/localtime:/etc/localtime:ro

- /mongo\_cluster/data1:/data/db

restart: always

mongors1n2:

container\_name: mongors1n2

image: mongo

command: mongod --shardsvr --replSet mongors1 --dbpath /data/db --port 27017

ports:

- 27027:27017

expose:

- "27017"

environment:

TERM: xterm

volumes:

- /etc/localtime:/etc/localtime:ro

- /mongo\_cluster/data2:/data/db

restart: always

mongors1n3:

container\_name: mongors1n3

image: mongo

command: mongod --shardsvr --replSet mongors1 --dbpath /data/db --port 27017

ports:

- 27037:27017

expose:

- "27017"

environment:

TERM: xterm

volumes:

- /etc/localtime:/etc/localtime:ro

- /mongo\_cluster/data3:/data/db

restart: always

**Appendix C**

AWS lamda script to ensure ec2 machine have schedule poweroff.

|  |
| --- |
| import boto3 |
| # Region your instances are in, e.g. 'us-east-1' |
| region = 'us-east-1' |
|  |
| # instances ID: ex. ['X-XXXXXXXX', 'X-XXXXXXXX'] |
| instances = [''X-XXXXXXXX'] |
|  |
| def lambda\_handler(event, context): |
| ec2 = boto3.client('ec2', region\_name=region) |
| ec2.start\_instances(InstanceIds=instances) |
| print('started your instances: {}'.format(str(instances))) |

Corresponding lambda configurations:

|  |
| --- |
| AWSTemplateFormatVersion: '2010-09-09'  Transform: 'AWS::Serverless-2016-10-31' |
| Description: An AWS Serverless Specification template describing your function. |
| Resources: |
| lambdastopec2: |
| Type: 'AWS::Serverless::Function' |
| Properties: |
| Handler: lambda\_function.lambda\_handler |
| Runtime: python3.6 |
| CodeUri: . |
| Description: '' |
| MemorySize: 128 |
| Timeout: 3 |
| Role: 'arn:aws:iam::<<<OWNER-ID>>>:role/lambda\_basic\_execution' |
| Events: |
| Schedule1: |
| Type: Schedule |
| Properties: |
| Schedule: cron(0 6 \* \* ? \*) |

**Appendix D**

Map function implemented in mapreduce:

# emit the 'values' to the reduce

map = Code('''

function() {

if (

this.body != 'repost' &&

this.body != '[deleted]' &&

this.body != '[removed]'

) {

emit(

this.link\_id,

{

'id': [this.id],

'score': [this.score],

'parent\_id': [this.parent\_id],

'body': [this.body],

'comment': [this.body]

}

);

}

}

''')

**Appendix E**

Reduce function implemented in mapreduce:

reduce = Code('''

function (key, values) {

const singleton = ['.', '-', '$', ',', ':', '%', "'"]

const regexParts = [

/\s+/,

/\]|\[|\(|\)/,

/-&gt;|&gt;|&lt;/,

/\$?[0-9]{6,}/,

/https?:\/\/|https?;\/\//,

/((?:www\.|(?!www)|[a-zA-Z]+\.)[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,}|www\.[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,})/,

/((?:www\.|(?!www))[a-zA-Z0-9]\.[^\s]{2,}|www\.[a-zA-Z0-9]\.[^\s]{2,})/,

/[^\x01-\x7F]+/,

/--|\\*|\.\.\.|"|:-|:|!!!|\?\?\?|\+|=|\/\/|;/,

/\s(\.-\$,:%')\s/

],

regexString = regexParts.map(function(x){return x.source}).join('|'),

tokenRegex = new RegExp(regexString, 'g');

var results = { posts: [], comments: [], match\_id: [], scores: [] };

for (var i = 0; i < values.length; i++) {

if (

values[i] &&

values[i].body &&

values[i].parent\_id &&

! singleton.includes(values[i].body)

) {

var comment = values[i].body;

var score = values[i].score;

var wantedParent = values[i].parent\_id[0].split('\_')[1];

for (var j = 0; j < values.length; j++) {

if (

values[j] &&

values[j].body &&

values[j].body != values[i].body &&

wantedParent == values[j].id &&

! singleton.includes(values[j].body)

) {

results.posts = results.posts.concat([

values[j].body[0].replace(tokenRegex, ' ').trim()

]);

results.comments = results.comments.concat([

comment[0].replace(tokenRegex, ' ').trim()

]);

results.match\_id = results.match\_id.concat(wantedParent);

results.scores = results.scores.concat(score);

}

}

}

}

if (

results.posts.length > 0 &&

results.comments.length > 0 &&

results.match\_id.length > 0 &&

results.scores.length > 0

) {

return { results };

}

}

''')

**Appendix F**

Implemented mapreduce (using Appendix D and Appendix E) with pymongo finalize:

# finalize: remove single case (reducer skipped)

finalize = Code('''

function finalize(key, values) {

if (

values &&

values.results &&

values.results.posts &&

values.results.posts[0] &&

values.results.comments &&

values.results.comments[0] &&

values.results.posts[0].length > 0 &&

values.results.comments[0].length > 0 &&

values.results.posts[0] != values.results.comments[0]

) {

return values.results;

}

}

''')

# select data

return col.map\_reduce(

map=map,

reduce=reduce,

finalize=finalize,

out='to\_from'

)

**Appendix G**

Codified penn tree:

return {

'CC': 2,

'CD': 3,

'DT': 4,

'EX': 5,

'FW': 6,

'IN': 7,

'JJ': 8,

'JJR': 9,

'JJS': 10,

'LS': 11,

'MD': 12,

'NN': 13,

'NNS': 14,

'NNP': 15,

'NNPS': 16,

'PDT': 17,

'POS': 18,

'PRP': 19,

'PRP$': 20,

'RB': 21,

'RBR': 22,

'RBS': 23,

'RP': 24,

'SYM': 25,

'TO': 26,

'UH': 27,

'VB': 28,

'VBD': 29,

'VBG': 30,

'VBN': 31,

'VBP': 32,

'VBZ': 33,

'WDT': 34,

'WP': 35,

'WP$': 36,

'WRB': 37

}

**Appendix H**

Create part of speech and sentiment type dataframe, then split into train and test:

pos = answers\_unique + questions\_unique

sent\_type = []

for i in range(len(questions\_unique)):

sent\_type.append('0')

for j in range(len(answers\_unique)):

sent\_type.append('1')

df\_adjusted = pd.DataFrame({

'pos': pos,

'type': sent\_type

})

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

df\_adjusted['pos'],

df\_adjusted['type'],

test\_size=0.2

)

**Appendix I**

Normalize data, by assigning stop\_value for a given row cell, if len(column) < stop\_gap, otherwise remove columns after stop\_gap - 1:

def normalize\_data(X\_data, stop\_gap=40, stop\_value=1, train=False):

if train:

length = len(sorted(X\_data, key=len, reverse=True)[0])

X\_data=np.array([xi+[stop\_value]\*(length-len(xi)) for xi in X\_data])

X\_data=pd.DataFrame(X\_data)

rows\_train, columns\_train = X\_data.shape

delta = stop\_gap - columns\_train

if delta > 0 and delta <= stop\_gap:

for i in range(delta):

X\_data['filler-{}'.format(i)] = stop\_value

if delta < 0:

rem\_list = [x for x in range(abs(columns\_train))][:delta]

X\_data = X\_data.iloc[:,rem\_list]

else:

columns\_train = len(X\_data)

delta = stop\_gap - columns\_train

if delta > 0 and delta <= stop\_gap:

for i in range(delta):

X\_data.append(stop\_value)

if delta < 0:

rem\_list = [x for x in range(abs(columns\_train))][:delta]

X\_data = X\_data[:len(X\_data) - rem\_list]

return(X\_data)

**Appendix J**

Create random forest classifier:

clf=RandomForestClassifier(n\_estimators=1500)

tr0 = time()

clf.fit(X\_train\_final, np.asarray(y\_train))

tr1 = time()

y\_pred=clf.predict(X\_test\_final)

tr2 = time()

**Appendix K**

Chatbot entrypoint script:

import os

cwd = os.getcwd()

from pymongo import MongoClient

from nltk import tag, word\_tokenize, tokenize

from nltk.tokenize import TreebankWordTokenizer, RegexpTokenizer

import sys

import joblib

from chatbot.nmt\_chatbot.inference import interactive

from chatbot.app.train import train

from chatbot.app.insert import insert

from chatbot.app.select import select

from chatbot.config import (

mongos\_endpoint,

database,

collection,

data\_directory

)

from QuestionAnswerCMU.utility import (

tokenizer,

normalize\_data,

replace,

penn\_scale,

qa\_model

)

from StackOverflow.utility import tokenize, so\_model

username='jimmy'

def main(op='generic'):

if op == 'insert':

client = MongoClient(mongos\_endpoint)

insert(

client,

database,

collection,

'{base}/chatbot/{subdir}'.format(base=cwd, subdir=data\_directory)

)

elif op == 'local':

client = MongoClient(mongos\_endpoint)

results = select(client, database, collection)

# combine sequence pairs

combined = {}

for doc in results.find():

if doc['value']:

for k, v in doc['value'].items():

if k in combined.keys():

combined[k] += v

else:

combined[k] = v

posts = combined['posts']

comments = combined['comments']

scores = combined['scores']

model = train(posts, comments, cwd=cwd)

elif op == 'generic':

# interative session

print('\n\nStarting interactive mode (first response will take a while):')

while True:

# prompt input

sentence = input('\n> ')

# tokenize + parts of speech

pos = tokenizer(sentence)

# convert pos to numeric

sentence\_pos = replace(pos, penn\_scale())

# normalize question

X\_sentence = normalize\_data(sentence\_pos, stop\_gap=40)

# check if question

prediction = qa\_model(cwd).predict([X\_sentence])

# generate response

if prediction == '0':

inference\_internal = interactive(sentence)

answers = inference\_internal(sentence)[0]

# display response

if answers is None:

print(colorama.Fore.RED + "Answer can't be empty!" + colorama.Fore.RESET)

elif answers['best\_score'] < 12:

print('hey {name}, maybe checkout {url}'.format(

name=username,

url=so\_model(cwd).predict([sentence])

))

else:

print('{response}'.format(response=answers['answers'][answers['best\_index']]))

if \_\_name\_\_ == '\_\_main\_\_':

if len(sys.argv) > 1 and sys.argv[1] == '--insert':

main(op='insert')

elif len(sys.argv) > 1 and sys.argv[1] == '--local':

main(op='local')

else:

main(op='generic')

1. https://docs.mongodb.com/manual/core/inmemory/ [↑](#footnote-ref-1)
2. https://aws.amazon.com/ec2/instance-types/p2/ [↑](#footnote-ref-2)
3. https://github.com/jeff1evesque/ist-664/tree/master/chatbot/utility [↑](#footnote-ref-3)
4. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html [↑](#footnote-ref-4)
5. https://aws.amazon.com/lambda/ [↑](#footnote-ref-5)
6. https://github.com/jeff1evesque/ist-664/blob/master/chatbot/utility/docker-compose.yml [↑](#footnote-ref-6)
7. https://en.wikipedia.org/wiki/Kubernetes [↑](#footnote-ref-7)
8. https://github.com/jeff1evesque/ist-664/blob/master/Vagrantfile [↑](#footnote-ref-8)
9. https://github.com/jeff1evesque/ist-664/tree/master/chatbot/data [↑](#footnote-ref-9)
10. https://github.com/jeff1evesque/ist-664/tree/master/QuestionAnswerCMU/data [↑](#footnote-ref-10)
11. http://www.cs.cmu.edu/~ark/QA-data/ [↑](#footnote-ref-11)
12. https://github.com/jeff1evesque/ist-664/blob/master/chatbot/data/reddit-2005-12 [↑](#footnote-ref-12)
13. https://github.com/daniel-kukiela/nmt-chatbot [↑](#footnote-ref-13)
14. https://medium.com/explore-artificial-intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912 [↑](#footnote-ref-14)
15. https://github.com/daniel-kukiela/nmt-chatbot/blob/59551eed4f868ba2030f933f92842177720ca121/README.md#setup [↑](#footnote-ref-15)
16. https://github.com/daniel-kukiela/nmt-chatbot/blob/master/setup/prepare\_data.py [↑](#footnote-ref-16)
17. https://github.com/daniel-kukiela/nmt-chatbot#demo-chatbot [↑](#footnote-ref-17)
18. https://github.com/jeff1evesque/ist-664/blob/master/ Wilson\_Levesque\_Final\_Project.docx [↑](#footnote-ref-18)
19. L. Lin, R. Zhang and X. Duan, "Adaptive Scene Category Discovery With Generative Learning and Compositional Sampling," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 25, no. 2, pp. 251-260, Feb. 2015. [↑](#footnote-ref-19)