
Deep Reinforcement Learning for Dialogue Generation

Release 0.1

Kapil Chiravarambath

Mar 27, 2019

CONTENTS

1	Getting started	3
1.1	Download Source	3
1.2	Change directory	3
1.3	Create Environment	3
1.4	Test Environment	3
1.5	Download Movie Dialog Dataset	3
1.6	Chat Demo	3
2	Deep Reinforcement Learning for Dialogue Generation	5
2.1	Motivation	5
2.2	Problem Description	5
2.3	Methodology	6
2.4	Dataset Description	6
2.5	References	7
3	Data Wrangling	9
3.1	Data Description	9
3.2	Load Data	10
3.3	Data Cleaning	12
4	Data Insights	17
4.1	Response Length	17
4.2	Conversation Length	19
4.3	Most common Responses	20
4.4	Dull responses	21

Contents:

GETTING STARTED

1.1 Download Source

> *git clone https://github.com/kc3/Springboard.git*

1.2 Change directory

> *cd capstone_2*

1.3 Create Environment

> *conda env create -n 'capstone_2' -f environment.yml*

1.4 Test Environment

> *tox*

1.5 Download Movie Dialog Dataset

> Download data using *python ./src/datamake_dataset.py ./src/data/raw ./src/data/interim ./src/data/processed*

1.6 Chat Demo

> Launch Chatbot using command *python -m src.models.predict*

DEEP REINFORCEMENT LEARNING FOR DIALOGUE GENERATION

2.1 Motivation

Conversational bots are one of the most exciting applications of NLP. Services such as Google Assistant, Apple Siri and Amazon Alexa are becoming ubiquitous and improving in capabilities rapidly. Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition. Some chatbots use sophisticated natural language processing systems, but many simpler ones scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database.

2.2 Problem Description

Efforts to build statistical dialog systems fall into two major categories, statistical machine translation models and task-oriented dialog systems.

The first treats dialogue generation as a source to target transduction problem and learns mapping rules between input messages and responses from a massive amount of training data. Seq2Seq models [2] or their variants are used for generating responses.

Seq2Seq model works by generating an internal representation of the (sentence, response) pairs in the dataset using a Recursive Neural Network using auto-encoders and then building a decoder to decode the internal representation into a sequence of words that would be the most likely response. The problem faced with this model is a high generation of “I don’t know” responses, which is a direct consequence of such occurrences in training text.

The second method focus on solving domain-specific tasks and often rely on carefully limited dialogue parameters, or hand-built templates with state, action and reward signals designed by humans for each new domain, making the paradigm difficult to extend to open-domain scenarios.

There are three problems that conversational dialog systems typically have:

- **Dull and generic responses:** Example,

A: *How is life?*

B: *I don't know what you are talking about.*

Another example of this is:

A: *How old are you?*

B: *I don't know.*

This can be explained as most systems try to maximize the probability of the next response, and *I don't know.* is an adequate response for most cases. This is further aggravated by the fact that this response shows up a lot in the training data set.

- **Repetitive Responses:** Example,

A: See you later!

B: See you later.

A: See you later.

and so on. Another example is,

A: Good bye!

B: Good bye!

A: Good bye!

and so on.

- **Short-sighted responses:** These systems tend to pick responses that do not increase the length of the conversation or make it interesting or even coherent. An example,

A: How old are you?

B: I am 16.

A: 16?

B: I don't know what you are talking about?

A: You don't know what you are saying?

and so on. The first response by agent B is a very close ended response and it is very hard for the agent A to come up with a good response for this. Unfortunately, the results of the response '*I am 16*' does not show up a few turns later. This is the main motivation for using Reinforcement Learning to solve this problem.

2.3 Methodology

The goal of this project would be to build a bot that can have an open dialogue with the user, having three important conversational properties: informativity, coherence and ease of answering.

This project solves the problem by creating a learning system of two agents already trained with supervised seq2seq model and each talking with one another with a goal of maximizing long term rewards, such as increasing the length of conversation and metrics mentioned above.

The main components of the project would be:

- **Training Engine:** An implementation of a LSTM based attention model for this project.
- **Reinforcement Engine:** An implementation of policy gradient method for training the dataset.
- **Chat App:** A web application to have conversations with the chatbot.

2.4 Dataset Description

The original paper uses the OpenSubtitles dataset. For this project we will be using the Cornell Movie Dialogs Corpus instead, partly because it has meta-data and it is of smaller size.

This corpus contains a large metadata-rich collection of fictional conversations extracted from raw movie scripts:

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies
- in total 304,713 utterances
- movie metadata included:

- genres
- release year
- IMDB rating
- number of IMDB votes
- IMDB rating
- character metadata included:
 - gender (for 3,774 characters)
 - position on movie credits (3,321 characters)

Dataset Location: https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

2.5 References

- [1] Deep Reinforcement Learning for Dialogue Generation, Li et al
- [2] Sequence to Sequence Learning with Neural Networks - Sutskever et al.

DATA WRANGLING

3.1 Data Description

The Cornell Movie Dialog corpus contains a metadata-rich collection of fictional conversations extracted from raw movie scripts:

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies
- in total 304,713 utterances
- movie metadata included:
 - genres
 - release year
 - IMDB rating
 - number of IMDB votes
 - IMDB rating
- character metadata included:
 - gender (for 3,774 characters)
 - position on movie credits (3,321 characters)

B) Files description:

In all files the field separator is ” +++\$+++ “

- movie_titles_metadata.txt
 - contains information about each movie title
 - fields:
 - * movieID,
 - * movie title,
 - * movie year,
 - * IMDB rating,
 - * no. IMDB votes,
 - * genres in the format [‘genre1’,‘genre2’,,‘genreN’]
- movie_characters_metadata.txt

- contains information about each movie character
- fields:
 - * characterID
 - * character name
 - * movieID
 - * movie title
 - * gender (“?” for unlabeled cases)
 - * position in credits (“?” for unlabeled cases)
- movie_lines.txt
 - contains the actual text of each utterance
 - fields:
 - * lineID
 - * characterID (who uttered this phrase)
 - * movieID
 - * character name
 - * text of the utterance
- movie_conversations.txt
 - the structure of the conversations
 - fields
 - * characterID of the first character involved in the conversation
 - * characterID of the second character involved in the conversation
 - * movieID of the movie in which the conversation occurred
 - * list of the utterances that make the conversation, in chronological order: [‘lineID1’, ‘lineID2’, ..., ‘lineIDN’] has to be matched with movie_lines.txt to reconstruct the actual content
- raw_script_urls.txt
 - the urls from which the raw sources were retrieved

3.2 Load Data

We are primarily going to use movie_conversations.txt and movie_lines.txt. First load the movie_lines.txt and extract the lines.

```
# Python imports
import numpy as np
import pandas as pd
import re
```

```
# Load the data
lines = open('../src/data/interim/movie_lines.txt', encoding='utf-8', errors='ignore
↳').read().split('\n')
conv_lines = open('../src/data/interim/movie_conversations.txt', encoding='utf-8',
↳errors='ignore').read().split('\n')
```

```
# The sentences that we will be using to train our model.
lines[:10]
```

```
['L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!',
 'L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!',
 'L985 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I hope so.',
 'L984 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ She okay?',
 'L925 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Let's go.',
 'L924 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ Wow',
 'L872 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Okay -- you're gonna need
↳to learn how to lie.',
 'L871 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ No',
 'L870 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I'm kidding. You know
↳how sometimes you just become this "persona"? And you don't know how to
↳quit?',
 'L869 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Like my fear of wearing
↳pastels?']
```

```
# The sentences' ids, which will be processed to become our input and target data.
conv_lines[:10]
```

```
["u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L194', 'L195', 'L196', 'L197']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L198', 'L199']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L200', 'L201', 'L202', 'L203']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L204', 'L205', 'L206']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L207', 'L208']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L271', 'L272', 'L273', 'L274', 'L275']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L276', 'L277']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L280', 'L281']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L363', 'L364']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L365', 'L366']"]
```

```
# Create a dictionary to map each line's id with its text
id2line = {}
for line in lines:
    _line = line.split(' +++$+++ ')
    if len(_line) == 5:
        id2line[_line[0]] = _line[4]
```

```
# Add the sentence end marker
id2line['L0'] = '<EOC>'
```

```
# Create a list of all of the conversations' lines' ids.
convs = [ ]
for line in conv_lines[:-1]:
    _line = line.split(' +++$+++ ')[-1][1:-1].replace('"', '').replace(" ", "")
    convs.append(_line.split(','))
```

```
convs[:10]
```

```
[['L194', 'L195', 'L196', 'L197'],  
 ['L198', 'L199'],  
 ['L200', 'L201', 'L202', 'L203'],  
 ['L204', 'L205', 'L206'],  
 ['L207', 'L208'],  
 ['L271', 'L272', 'L273', 'L274', 'L275'],  
 ['L276', 'L277'],  
 ['L280', 'L281'],  
 ['L363', 'L364'],  
 ['L365', 'L366']]
```

```
# Sort the sentences into questions (inputs) and answers (targets)  
questions = []  
answers = []  
  
for conv in convs:  
    for i in range(len(conv)-1):  
        questions.append(id2line[conv[i]])  
        answers.append(id2line[conv[i+1]])  
  
    # Add a conversation end marker  
    questions.append(id2line[conv[len(conv)-1]])  
    answers.append(id2line['L0'])
```

```
# Check if we have loaded the data correctly  
limit = 0  
for i in range(limit, limit+5):  
    print(questions[i])  
    print(answers[i])  
    print()
```

```
Can we make this quick?  Roxanne Korrine and Andrew Barrett are having an incredibly_  
↪horrendous public break- up on the quad.  Again.  
Well, I thought we'd start with pronunciation, if that's okay with you.  
  
Well, I thought we'd start with pronunciation, if that's okay with you.  
Not the hacking and gagging and spitting part.  Please.  
  
Not the hacking and gagging and spitting part.  Please.  
Okay... then how 'bout we try out some French cuisine.  Saturday?  Night?  
  
Okay... then how 'bout we try out some French cuisine.  Saturday?  Night?  
<EOC>  
  
You're asking me out.  That's so cute. What's your name again?  
Forget it.
```

3.3 Data Cleaning

Next we work on cleaning data by expanding English contractions such as “don’t” for “do not”.


```

contractions_dict = {
    "ain't": "am not ",
    "aren't": "are not",
    "'bout": "about",
    "can't": "cannot",
    "can't've": "cannot have",
    "'cause": "because",
    "could've": "could have",
    "couldn't": "could not",
    "couldn't've": "could not have",
    "didn't": "did not",
    "doesn't": "does not",
    "don't": "do not",
    "hadn't": "had not",
    "hadn't've": "had not have",
    "hasn't": "has not",
    "haven't": "have not",
    "he'd": "he would",
    "he'd've": "he would have",
    "he'll": "he will",
    "he'll've": "he will have",
    "he's": "he is",
    "how'd": "how did",
    "how'd'y": "how do you",
    "how'll": "how will",
    "how's": "how is",
    "i'd": "I had",
    "i'd've": "I would have",
    "i'll": "I will",
    "i'll've": "I will have",
    "i'm": "I am",
    "i've": "I have",
    "isn't": "is not",
    "it'd": "it had",
    "ot'd've": "it would have",
    "it'll": "it will",
    "it'll've": "it will have",
    "it's": "it is",
    "let's": "let us",
    "ma'am": "madam",
    "mayn't": "may not",
    "might've": "might have",
    "mightn't": "might not",
    "mightn't've": "might not have",
    "must've": "must have",
    "mustn't": "must not",
    "mustn't've": "must not have",
    "needn't": "need not",
    "needn't've": "need not have",
    "o'clock": "of the clock",
    "oughtn't": "ought not",
    "oughtn't've": "ought not have",
    "shan't": "shall not",
    "sha'n't": "shall not",
    "shan't've": "shall not have",
    "she'd": "she would",
    "she'd've": "she would have",

```

(continues on next page)

(continued from previous page)

```
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that had",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you had",
```

(continues on next page)

(continued from previous page)

```

    "you'd've": "you would have",
    "you'll": "you shall",
    "you'll've": "you shall have",
    "you're": "you are",
    "you've": "you have"
}

contractions_re = re.compile('(' + '%s)' % '|'.join(contractions_dict.keys()), re.
↳ IGNORECASE)

def expand_contractions(s, contractions_dict=contractions_dict):
    def replace(match):
        return contractions_dict[match.group(0).lower()]
    text = contractions_re.sub(replace, s)
    return re.sub(r"[-()\"#/@;:<>{}`+=~|.!?],", "", text)

expand_contractions('you\'re great!')
```

```
'you are great'
```

```

# Clean the data
clean_questions = []
for question in questions:
    clean_questions.append(expand_contractions(question))

clean_answers = []
for answer in answers:
    if answer != '<EOC>':
        clean_answers.append(expand_contractions(answer))
    else:
        clean_answers.append(answer)
```

```

# Take a look at some of the data to ensure that it has been cleaned well.
limit = 0
for i in range(limit, limit+5):
    print(clean_questions[i])
    print(clean_answers[i])
    print()
```

```

Can we make this quick  Roxanne Korrine and Andrew Barrett are having an incredibly_
↳ horrendous public break up on the quad  Again
Well I thought we would start with pronunciation if that is okay with you

Well I thought we would start with pronunciation if that is okay with you
Not the hacking and gagging and spitting part  Please

Not the hacking and gagging and spitting part  Please
Okay then how about we try out some French cuisine  Saturday  Night

Okay then how about we try out some French cuisine  Saturday  Night
<EOC>

you are asking me out  that is so cute what is your name again
Forget it
```

We now have a clean Question and Answer datasets that are used for further processing and gathering further insights.

```
# Save the files
conv_final = pd.DataFrame({'question': clean_questions, 'answer': clean_answers})
conv_final.to_csv('../src/data/processed/movie_qa.txt')
```

```
conv_final.head()
```

DATA INSIGHTS

Load the already cleaned question answer pairs generated.

```
# Python Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Load the QA dataset
qa = pd.read_csv('../src/data/processed/movie_qa.txt', index_col=0)
qa.head()
```

4.1 Response Length

We first look at the how long a typical response is. We start by filtering out End of conversation markers ‘EOC’

```
# Filter out EOC
qa_no_eoc = qa[qa.answer != '<EOC>']['answer']

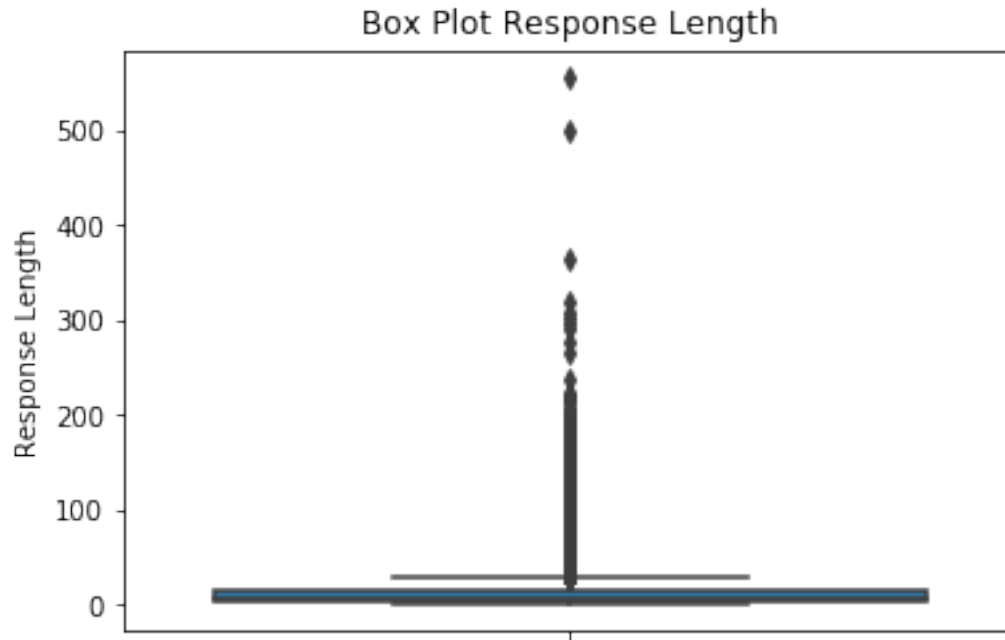
# Get lengths for each value
resp_len = qa_no_eoc.apply(lambda x: len(str(x).split()))

resp_len.describe()
```

```
count    221616.000000
mean         11.083054
std          12.630780
min           0.000000
25%           4.000000
50%           7.000000
75%          14.000000
max          556.000000
Name: answer, dtype: float64
```

We see that the median response length is 7 words, but the distribution is skewed by presence of extreme outliers. Lets look at the box plot for verification.

```
# Box Plot
_ = sns.boxplot(resp_len, orient='v')
_ = plt.ylabel('Response Length')
_ = plt.xlabel('')
_ = plt.title('Box Plot Response Length')
```



We remove the outliers by using the 1.5 times IQR thumb rule, so we filter out values outside range (2, 30)

```

filt = np.logical_and(resp_len > 2, resp_len < 30)
resp_len_filt = resp_len[filt]

resp_len_filt.describe()

```

```

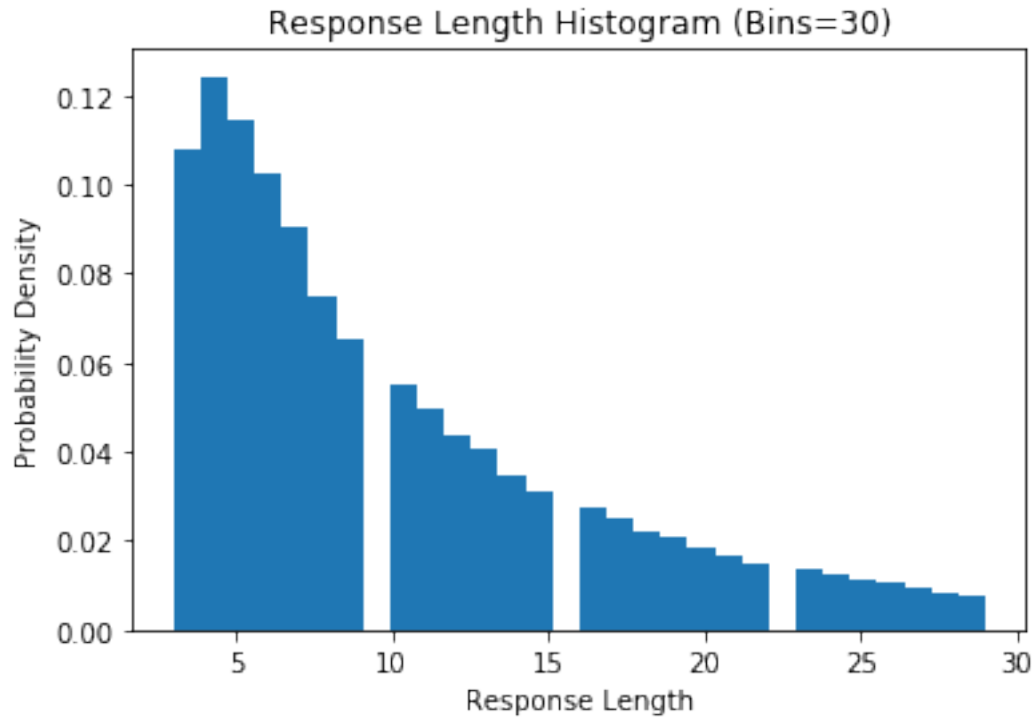
count      173926.000000
mean         9.990921
std          6.379758
min          3.000000
25%          5.000000
50%          8.000000
75%         13.000000
max         29.000000
Name: answer, dtype: float64

```

```

_ = plt.hist(resp_len_filt, density=True, bins=30)
_ = plt.xlabel('Response Length')
_ = plt.ylabel('Probability Density')
_ = plt.title('Response Length Histogram (Bins=30)')

```



We see that response length goes down exponentially. People prefer brevity in responses over longer sentences. This is an important insight for training batches, as the longer input sequences lead to padding in shorter sequences increasing cpu/memory usage.

4.2 Conversation Length

Next we look at how long do conversations last. This can be easily inferred by looking at the eoc_markers.

```
eoc_marks = sorted(list(set(qa.answer.index.values) - set(qa_no_eoc.index.values)))

conv_len_arr = [eoc_marks[0]]
for i in range(len(eoc_marks)-1):
    conv_len_arr.append(eoc_marks[i+1] - eoc_marks[i] - 1)

conv_len = pd.Series(conv_len_arr)
conv_len.describe()
```

```
count      83097.000000
mean         2.666955
std          2.891798
min          1.000000
25%          1.000000
50%          2.000000
75%          3.000000
max          88.000000
dtype: float64
```

We again see extreme values in this distribution. 75% of the conversations are less than 3 turns. We again filter for extreme values in range (2,30) and check how many conversations are dropped.

```

filt = np.logical_and(conv_len > 0, conv_len < 30)
conv_len_filt = conv_len[filt]

conv_len_filt.describe()
print('% kept: {0:.2%}'.format(conv_len_filt.count() / conv_len.count()))

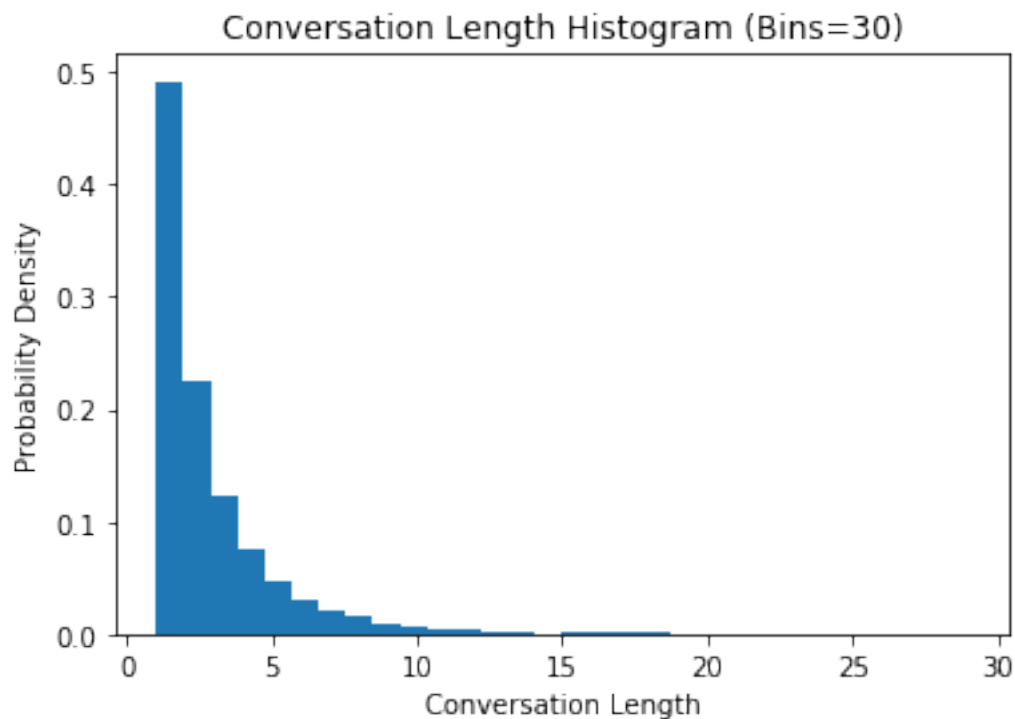
```

```
% kept: 99.92%
```

```

_ = plt.hist(conv_len_filt, density=True, bins=30)
_ = plt.xlabel('Conversation Length')
_ = plt.ylabel('Probability Density')
_ = plt.title('Conversation Length Histogram (Bins=30)')

```



We see that in naturally occurring conversations, most conversations end in 2-3 turns. This gives us an insight that we can limit the number of turns to some small number, such as 5, for evaluation of the model.

4.3 Most common Responses

Next we look at the most common responses generated in the data.

```

from collections import Counter

resp_cnt = Counter(qa_no_eoc)

resp_cnt.most_common(10)

```

```

[('What', 1532),
 ('Yes', 1484),
 ('No', 1423),

```

(continues on next page)

(continued from previous page)

```
( 'Yeah', 1117),
( 'Why', 502),
( 'I do not know', 397),
( 'Okay', 357),
( 'Sure', 277),
( 'Oh', 266),
( 'Thank you', 246)]
```

We see that most common responses are responses to questions and most act as conversation stoppers (with the exception of ‘Why?’). We expect them to see again in the list of dull responses next.

```
resp_dist = pd.Series([i for i in resp_cnt.values()])
resp_dist.describe()
```

```
count      189270.000000
mean         1.170899
std          7.122922
min          1.000000
25%          1.000000
50%          1.000000
75%          1.000000
max         1532.000000
dtype: float64
```

We see that most responses occur in the corpus only once. This should also result in model predicting unique responses.

4.4 Dull responses

Identifying dull responses is an important functionality for deciding on rewards in Reinforcement learning module. We look at all the questions that result in end of conversation.

```
# Questions generating a end of conversation response
qa_dull = qa[qa.answer == '<EOC>']['question']

dull_cnt = Counter(qa_dull)

dull_cnt.most_common(10)
```

```
[ ('No', 360),
  ('Yes', 356),
  ('Yeah', 307),
  ('What', 234),
  ('Okay', 211),
  ('Sure', 154),
  ('I do not know', 145),
  ('Thank you', 136),
  ('Oh', 120),
  ('Right', 104)]
```

This is again expected as these responses are stop responses that typically end a conversation.