1 Purpose - Background

The original purpose of this project was to try and create a chat bot that could respond to any given question, in the way that we might respond to a question.
A brief example:

Query: How are you? Reply: Good, you?

There are two parts of the application, the preprocessing of data, and then the active chatbot after that. So first let's break down how the workflow for both should look like:

Preprocessing :

1. Obtain prior data, creating a corpus of messages
2. Break down data into message pairs, so we can map a question to a potential query to it's response
   - I hope we get a prize → I wish
3. Train Markov Chain on a corpus of just our responses
4. Create matrix for LSI input

Active Bot (cycle until quit statement) :

1. Accept user input
2. Try to fit user input to a set of previous queries via LSI
3. Take the first word from the matching response pair, and then set off the Markov chain of decisions using that first word.
4. Return output

2 Preprocessing

2.1 Creating a corpus

To create a corpus of messages, we used Facebook's data download to obtain all message data from a set period of time. Originally we tried to download a longer range, but the download glitched out, so we settled for a shorter range (June 2015- December 2015). This is also in the JSON format - so we can easily manipulate it with python.
Since we am not sending all of the text files, this version of the bot has the code for this Preprocessing and message pair creation in the INTRO function of the python file.

```
ff = open("./messages.txt","w")
file = open(os.path.join(root, name) , 'r')
data = json.load(file)
for piece in data['messages']:
   if(piece["sender_name"]=='Pranav Chaudhari'):
      if(data['messages'][length-1]["content"] != piece['content']):
         cplusone=data['messages'][count+1]
         if(cplusone["sender_name"]!='Pranav Chaudhari'):
            cplusone=data['messages'][count+1]["content"]
            if('photo' not in cplusone and 'sticker' not in cplusone ):
               ff.write(cplusone.lower()+"\n")
            if(piece["content"]!='You sent a photo.' and piece["content"]!='You sent
                a sticker.'):
               ff.write(piece["content"].lower()+"\n")
   count = count+1
```

Listing 1: The data collection from JSON file

2.2 Message pair creation & Sending corpus

With this information, now we can create a set of message pairs, as well as we have a corpus of sentences for the Markov chain to generate off of. We use a dictionary to map a query and it's response, and then create an array of queries to send to the LSI(Latent Semantic Indexing) input. In LSI, words are then compared by taking the cosine of the angle between the two vectors (or the dot product between the normalizations of the two vectors) formed by any two rows. Values close to 1 represent very similar words while values close to 0 represent very dissimilar words.

2.3 Train Markov Chain on a corpus of just our responses

At it's core we create a dictionary, and we have an array of all possible words that come after a single word. The actual generation occurs during the active bot querying.

```
def make_pairs(corpus):
   for i in range(len(corpus)-1):
      yield (corpus[i], corpus[i+1])

def wwdictionary(pairs):
   for word_1, word_2 in pairs:
      if word_1 in word_dict.keys():
         word_dict[word_1].append(word_2)
      else:
         word_dict[word_1] = [word_2]
```

2.4 Create matrix for LSI input

Thanks to a package called gensim, a lot of the heavy lifting is removed from our end. To convert documents to vectors, we'll use a document representation called bag-of-words. In this representation, each document is represented by one vector where each vector element represents a question-answer pair. We repeat this process for all of the sentences in our query corpus, saving it to a larger structure referred to as a Matrix Market.

```
for text in texts:
      for token in text:
         frequency[token] += 1
texts = [[token for token in text if frequency[token] > 1] for text in texts]
dictionary = corpora.Dictionary(texts)
dictionary.save('/tmp/large.dict')
corpus = [dictionary.doc2bow(text) for text in texts]
corpora.MmCorpus.serialize('/tmp/corpus.mm', corpus)
```

3 Active Bot

The simple loop keeping everything in track:

```
while not re.search(r"(?i)quit",user_response):
   user_response=user_response.lower()
   query(user_response)
   user_response = input()
```

3.1 Send user query to LSI

We take the user input, and using gensim, construct an LSI model to test the query against. Since we will obtain a list of somewhat similar queries, we make a random choice from the top ten to send to the Markov chain creation process.

```
dictionary = corpora.Dictionary.load('/tmp/large.dict')
corpus = corpora.MmCorpus('/tmp/corpus.mm')
lsi = models.LsiModel(corpus, id2word=dictionary, num_topics=30)
index = similarities.MatrixSimilarity(lsi[corpus])
vec_bow = dictionary.doc2bow(onetwo.lower().split())
vec_lsi = lsi[vec_bow]
sims = index[vec_lsi]
sims = sorted(enumerate(sims), key=lambda item: -item[1])
catchthis(random.choice(sims[0:10]))
```

3.2 Markov Chain generation

We get the suggested question in CATCHTHIS, and then we send that into CHECKME to look for a matching answer pair. Then we take the first word of a random answer pair, and proceed to just run through the list. If we run into a KeyError before we are done computing a string of length 1 to 15, then we just print what we have generated up to that point.

```
def catchthis(onetwo):
    chain=[]
    sss=checkme(documents[onetwo[0]])
    chain.append(sss.split()[0])
    length=random.randint(1, 15)
    for i in range(length):
        try:
            chain.append(random.choice(word_dict[chain[-1]]))
        except(KeyError):
            return print(' '.join(chain))
    print(' '.join(chain))

def checkme(check):
    sett=[]
    for i in pairs:
        if(i["question"]==check):
            sett.append(i["answer"])
    return random.choice(sett)
```

4 Results

The Markov chain, while great at using a corpus to generate more text, fails to recognize some latent structures. we prefer this over a bigram model, as we get more of an opportunity to create new text instead of something that has been seen before. The result is definitely not we expected from the start, and has yet to pass a Turing test when I'm generating messages to send to our friends. We think this is in part due to the somewhat small size of corpus, and how our speech patterns change over time, so old data is not going to work to simulate how we might speak now.

Query: How are you? Reply: who is good adc sion top and at 8 Reply: when ranked game damn words Reply: last goodbye so they stopped talking about an updated