TF-IDF and RNN Movie Chat Bot

by:
Odin Songe Thorsen
Rohan Subramanian

Motivation:

We wanted to make a chatbot that would respond to you using language it learned from movies, we imagined this would lead to some dramatic and interesting dialog.
The first idea we had to accomplish this is using something called tf_idf, or term frequency divided by inverse document frequency. Term frequency being how often a term shows up in that particular document, and inverse document frequency being the number of documents total, divided by the number of documents containing the term.

Method:

Our first approach was to make a TF_IDF collection, one for each "document" in our corpus (I say document but in our case it is just one line from a movie per document). Then we convert user input into a TF_IDF based on our compiled data and use cosine similarity to determine which sentence in our corpus was most similar to the input of the user..
Once we know what the most similar sentence is, we simply use the very next line from that movie as our response.
This turned out to be a pretty cool idea, and we had a lot of fun talking to it, but it was easy to get responses that made no sense simply by using sentences that are very much unlike the ones in our training data. In the end it was basically a

dictionary of movie quotes that could handle spelling errors or word switches and could fall back on a random number generator when it fails completely. In order to improve on this we thought we could make a pipeline of sorts, what if we used the movie quote response if the similarity score met a certain threshold, and used something else otherwise?
One thing we considered for that something else was a n-gram model that would predict the next word based on nearest neighbours and a hidden markov model, like we did previously in our homework assignment. But since we already did it before and it had its own issues with not knowing when to terminate a sentence and only making somewhat coherent sentences, we thought we might be better off reaching for a neural network solution. After doing some research we decided to go with LSTM or long short term memory, which is a kind of recurrent neural network.

Neural networks are a kind of deep learning algorithm that comes with an arbitrary number of nodes and layers (known as hidden layers) and they each have weights that transform features as they pass through the nodes. A connected neural network is one where each node in one layer is connected to every other node in the next layer, which seems to be the norm.
The way a neural network "learns" is through backwards propagation, the neural network transmits outputs all the way to the end of the net, and only once every neuron has fired does it start to update it's weight, beginning at the end and propagating backwards. This works because each neuron can know it's contribution to the cost function by calculating the partial derivative with regards to itself, just like in gradient descent we adjust the weight in the opposite direction of the derivative slope, and the result is that every weight in the network gets nudged in the right direction, proportional to how much they contributed to the loss function.

A recurrent neural network is a kind of neural network that uses one hidden layer but sends it's past outputs into itself as input on the next time series step. In the case of NLP applications one time step is one token, which can be a character, a word or a sentence. It is a time series because in a sentence, some words come **before** other words (left to right usually). And in a word some characters come before others. This means we can treat NLP problems much like time series problems and that's why recurrent neural networks work so well for tasks like this.

The training starts on time step 0 with the first token, that token enters the hidden layer which then outputs some values. These values are then propagated into the future, as inputs for the second layer. The second layer is actually the same as the first layer, that is why it is called recurrent, but now it gets the second token, and the previous output, as input. This allows recurrent neural networks to learn from how a sentence began and interpret it more accurately than a neural net with no temporal awareness.
Even though I just said recurrent neural nets can remember the start of a sentence, that is not really true. The first token influences the second token greatly, the third token moderately, and by the fourth or fifth token it is barely noticeable. So while it might be good for learning bigrams and trigrams, it will not be able to remember there was a negation at the start of the sentence by the time it gets to the end of it.
This is where LSTM comes in, by adding a memory unit that interacts with the neural net, it has a mechanism for remembering key words that drastically changes the meaning of a sentence, and the memory unit itself is also trained with the model, getting better at knowing which words to keep in memory and which ones to forget. Turns out forgetting is the most important part of remembering when it comes to neural nets!

TF_IDF_chatbot.py

```
from utils.word_vector_math import *


def parse_text():
   f =
open("C:/Users/Administrator/PycharmProjects/pythonProject/chatbot/movie_lines.txt",
"r", encoding="iso-8859-1")
   sentences = []
   line_sentences = []
   while True:
       line = f.readline()
       if not line:
           break
       parts = line.split(" +++$+++ ")
       line_num = int(parts[0][1:])
       line_text = parts[4][:-1]  # trims off \n character
       line_sentences.append((line_num, line_text))
```

```
    for sent in sorted(line_sentences):
        sentences.append(sent[1])
    return sentences

tokenizer = TreebankWordTokenizer()
docs = parse_text()
d_l = len(docs) / 20
docs = docs[0:int(d_l)]
[tf_idfs, lexicon, idfs] = tf_idf(docs)
s = ""
print("Hello there, what's on your mind?")
while s != "I don't want to talk to you anymore":
    s = input()
    user_input_tf_idf_vec = td_idf_erizer(s, lexicon, idfs)
    max_similarity = -99999
    response = "I'm sorry, I don't understand you"

    for i, tf_idf_vec in enumerate(tf_idfs):
        similarity = cosine_similarity(user_input_tf_idf_vec, tf_idf_vec)
        if similarity > max_similarity:
            max_similarity = similarity
            response = docs[i+1]
    print(response)
```

Vector_math.py:

```
import math
from collections import defaultdict, OrderedDict
from nltk.tokenize import TreebankWordTokenizer
tokenizer = TreebankWordTokenizer()

def cosine_similarity(vec1, vec2):
    # we assume these come in as dictionaries because that's how dist freq is
usually kept
    vec1 = [v for v in vec1.values()]
    vec2 = [v for v in vec2.values()]
    dot_product = 0
    for i, num in enumerate(vec1):
        dot_product += vec2[i] * num
    # len is for euclidean distance
    len_vec1 = math.sqrt(sum([n**2 for n in vec1]))
    len_vec2 = math.sqrt(sum([n**2 for n in vec2]))

    return dot_product / (1 + (len_vec1 * len_vec2))


def inverse_doc_freq(docs, word):
```

```
    frequency = 0
    for document in docs:
        if word in document:
            frequency += 1
    return len(docs) / (1 + frequency)


def term_freq(doc, word):
    count = 0
    for w in doc:
        if w.lower() == word.lower():
            count += 1
    return count / len(doc)


def get_freq_dist(doc):
    dist = defaultdict(lambda: 0)
    for word in doc:
        dist[word] += 1
    size = len(doc)
    for (word, count) in dist:
        dist[word] = count / size
    return dist


def get_word_counts(doc):
    dist = defaultdict(lambda: 0)
    for word in doc:
        dist[word] += 1
    return dist


def td_idf_erizer(sentence, lexicon, idfs):
    vec = OrderedDict((token, 0) for token in lexicon)
    word_counts = get_word_counts(tokenizer.tokenize(sentence))
    for word, count in word_counts.items():
        if word not in lexicon:
            continue
        tf = count / len(word_counts)
        vec[word] = (math.log2(tf) * math.log2(idfs[word]))
    return vec
```

```
def tf_idf(docs):
    lexicon = set()
    for doc in docs:
```

```
        lexicon.update(tokenizer.tokenize(doc))
    idfs = {}
    i = 0
    for word in lexicon:

        idfs[word] = inverse_doc_freq(docs, word)
        i += 1

    result = []
    tokenized_docs = [tokenizer.tokenize(doc) for doc in docs]

    for doc in tokenized_docs:
        vec = OrderedDict((token, 0) for token in lexicon)
        word_counts = get_word_counts(doc)
        for word, count in word_counts.items():
            tf = count / len(word_counts)
            vec[word] = (math.log2(tf) * math.log2(idfs[word]))
        result.append(vec)

    return [result, lexicon, idfs]
```

Results and Analysis:

The tf_idf method was incredibly space and time inefficient at large scale. With 300 000 sentences from the movie database we had over 90 000 unique tokens which meant a 90 000 dimensional array of vectors that needed to be compared for cosine similarity. Training the model is just a matter of preprocessing the tf_idf vectors before launching our chatbot, if this was the only thing slowing us down it would have been no problem because we could just calculate it one time, save that to a json file and reload it on subsequent boot ups. But unfortunately when an input is written by the user it does take considerable time to make the new tf_idf with 90 000 dimensions and then getting the cosine similarity compared to all the 300 000 other ones in our dataset. For our demo we decided to only use 10% of our data, which made the bot run at moderate speed and show the proof of concept. The good news is that this runtime could likely be improved, either by using Latent Semantic Analysis or by finding clever ways to find cosine similarity matches without having to do every single one, there were still many things we would have liked to try if time permitted.

The RNN model on the first iteration took almost an hour to train over only two epochs. We quickly learned that the corpus of 300,000 sentences was also inefficient when it comes to testing out hyperparameters in the keras model. After cutting down the training set significantly to 10% of the original and tweaking the epochs, batch size, and max sentence length (which we found would exponentially increase training time), we managed to train an acceptable model

within 15 minutes. However since that is still unacceptable on startup for a conversation bot, the model was saved and loaded later by the text generator portion. After at least 30 successfully compiled models and countless failed compilations, we settled on a training set that wasn't too big to be super slow, but not too small to be uninteresting. The end result of the RNN generation is weirdly structured sentences that sometimes make sense, but more often do not. Moving forward, training the model in different ways (perhaps with a stateful LSTM layer that we couldn't get working this time).

Conclusion:

After countless hours researching various chat bot architectures and trial and error over and over again, we produced a chatbot that when fed inputs somewhat similar to the movie lines it's trained on, will produce the proper result. There is a lot of room to improve however: we could train the RNN better, optimize the TF-IDF functions so we can use a larger portion of the corpus, or a million other things. The chat bot that was produced however is good at its niche job of connecting you with what movie characters would say.

References:

We learned a lot from the book:
Natural Language Processing in Action the first chapter was shared in modules of this course but we also bought the rest of the book.
Corpus - Cornell movie dialogues corpus