Machine Learning Engineer Nanodegree

Capstone Project: Depression Detector

Mr. Pongsasit Thongpramoon

April 2020

Index

0. Abstract

Nowadays many places locked down, people can't go outside their rooms or houses, loneliness cause people stressful and depress. And most of them using social network like twitters to express their feeling. So I used their posts for training my machine learning model. Using TWINT for data scraping, pandas for data wrangling and feature engineering. Then use GloVe or Fasttext for embedding the words for PyTorch simple RNN and LSTM models (4 experiments), and use pretrained model like ULMFiT (1 experiment), and AWS Blazing Text (1 experiment) for finding the best method (evaluated by precision, recall, F-1 Score, Accuracy). Then pick only 1 model from 6 experiments for making Ib application as product.

1. Background

People write to communicate with others. In addition to describing simple factual information, people also use writing to express their activities, and convey their feelings, mental states, hopes and desires. Recipients then use this written information from emails and other forms of social media texts to make inferences, such as what someone else is feeling, which in turn influences interpersonal communication. Even when writing is shaped by the way someone wants to be perceived, this text still provides important cues to help friends and family recognize important life events for them to respond to with support and encmyagement. When people write digitally (e.g. on email or social media), their texts are processed automatically. Natural language processing (NLP) techniques make inferences about what people say and feel, and these inferences can trigger messages or other actions. One of the most common uses of NLP is in the

marketing sector where companies analyze emails and social media to generate targeted advertising and other forms of 'interventions' (generally aimed at changing my behavior toward buying something or following a link). HoIver, potential applications of NLP techniques extend beyond marketing. For example, NLP techniques have been identified as an important area of growth within the artificial intelligence (AI) in medicine community (Peek et al. 2015). In this paper, I discuss the potential for NLP techniques to be utilized in the mental health sector. Mental health applications are designed to support mental health and Illbeing in an online environment. These applications are reliant on interdisciplinary collaboration betIen researchers and practitioners from areas such as computational linguistics, human-computer interaction and mental health (and mental health service delivery). Interdisciplinary collaborations benefit from the development of a common language and body of research evidence (Calvo et al. 2016). Regrettably, there is a paucity of organized literature at this intersection betIen NLP, human-computer interaction and mental health research.

2. Related work

2.1 From my reference [2], the researchers use this method for making a depression detector.

A. Data

They gathered information from the Shared Task organizers of the CLPsych 2015 conference. This dataset was developed from an amalgamation of users with public Twitter accounts who posted a status update in the form of a statement of diagnosis, such as "I was diagnosed with X today", where X would represent either depression or PTSD. For each user, up to 3000 of their most recent public tIets Ire included in the dataset, and each user was isolated from the others. It should be noted that this 3000 tIets limit derives from Twitter's archival polices, and that most

tIets concentrated long after a two-month timespan may possibly loIr the effectiveness of a classifier, as shown by Tsugawa et al. Before releasing the dataset to participants, Coppersmith et al. matched age and gender to the demographics of the population, ultimately designing a dataset which consisted of 574 individuals (~63% of the dataset) with no mental health condition, and 326 users (~36% of the dataset) with a mental health condition of depression. For the purposes of their research, this resulted in 1,253,594 documents (tIets) as control variables, and 742,560 documents with a mental health condition of depressed. Lastly, each user, and the users they interacted with, had been anonymized the dataset to ensure their privacy would be protected. In addition, Shared Task participants Ire required to sign a privacy agreement, institute security measures on the data, and obtain the approval of an ethics review board in order to secure the dataset. Data had been distributed in compliance with Twitter company policy and terms of service.

B. Features

- Bag of Words approach

C. Classifiers

- Decision trees

- Support Vector Machine Classifiers

- Logistic regression

- Naïve Bayes

D. Results

Classification Algorithm	Precision	Recall	F1-Score	Accuracy	Samples
Decision Trees	0.67	0.68	0.75	0.67	332421
Linear Support Vector Classifier	0.83	***0.83***	0.83	0.82	332421
Naïve Bayes w/ 2-grams	0.82	0.82	0.82	0.82	332421
Logistic Regression	***0.86***	0.82	***0.84***	0.82	332421
Naïve Bayes w / 1-gram	0.81	0.82	0.81	***0.86***	332421
Ridge Classifier	0.81	0.79	0.78	0.79	332421

So, from this research's results. Precision: 0.86, Recall: 0.83, F1-Score 0.84, Accuracy 0.86. Because I will use new dataset, this research's result will be only reference not my goal.

3. Methodology

3.1 Data Collecting

Because the data set used by [2] was not opened to me, I have to collect it by myself using similar concept using TWINT. TWINT is an advanced Twitter scraping tool written in Python that doesn't use Twitter's API, allowing you to scrape a user's folloIrs, following, TIets and more while evading most API limitations. I will use TWINT for scraping depressive tIets. And positive tIets will use tIets from Sentiment140 dataset.

3.2 Text Processing

3.2.1 Decapitalize the alphabets.

3.2.2 Remove punctuation mark.

3.2.3 Remove common word.

3.2.4 Reducing words to their stem (e.g. "play" from "playing") using stemming algorithms.

3.2.5 Ignoring frequent words that don't contain much information, called stop words, like "a," "of," etc.

3.3 Feature Extraction

3.3.1 Bag of words

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity.

The bag-of-words model is commonly used in methods of document classification where the (frequency of) occurrence of each word is used as a feature for training a classifier.

The bag-of-words model is simple to understand and implement and has seen great success in problems such as language modeling and document classification.

It involves two things:

1. A vocabulary of known words: This step revolves around constructing a document corpus which consists of all the unique words in the whole of the text present in the data provided. It is sort of like a dictionary where each index will correspond to one word and each word is a different dimension.

   Example: If we are given 4 reviews for an Italian pasta dish.

   Review 1 : This pasta is very tasty and affordable.

   Review 2: This pasta is not tasty and is affordable.

   Review 3 : This pasta is delicious and cheap.

   Review 4: Pasta is tasty and pasta tastes good.

   Now if I count the number of unique words in all the fmy reviews, I will be getting a total of 12 unique words. Below are the 12 unique words:

1. 'This'
2. 'pasta'
3. 'is'
4. 'very'
5. 'tasty'
6. 'and'
7. 'affordable'
8. 'not'
9. 'delicious'
10. 'cheap'
11. 'tastes'
12. 'good'

2. A measure of the presence of known words: Now if I take the first review and plot count of each word in the below table I will have where row 1 corresponds to the index of the unique words and row 2 corresponds to the number of times a word occurs in a review. (Here review 1)

1	2	3	4	5	6	7	8	9	10	11	12
1	1	1	1	1	1	1	0	0	0	0	0

I will make a sparse vector of d-unique words and for each document (review) I will fill it will number of times the corresponding word occurs in a document.

Review 4: Pasta is tasty and pasta tastes good.

For example, in review 4 "pasta" has count 2 whereas in review 1 it is 1.

After converting the reviews into such vectors, I can compare different sentences and calculate the Euclidean distance betIen them so as to check if two sentences are similar or not. If there would be no common words distance would be much larger and vice-versa. BOW doesn't work very Ill when there are small changes in the terminology I are using as here I have sentences with similar meaning but with just different words. This results in a vector with lots of zero scores called a sparse vector or sparse representation. Sparse vectors require more memory and computational resmyces when modeling and the vast number of positions or dimensions can make the modeling process very challenging for traditional algorithms.

3.3.2 TF-IDF

tf–idf or TFIDF, short for term frequency-inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general. tf–idf is one of the most popular term-Iighting schemes today; 83% of text-based recommender systems in digital libraries use tf–idf.

This concept includes:

· ***Counts***. *Count the number of times each word appears in a document.*

· ***Frequencies***. *Calculate the frequency that each word appears in a document out of all the words in the document.*

Term frequency:

Term frequency (TF) is used in connection with information retrieval and shows how frequently an expression (term, word) occurs in a document. Term frequency indicates the significance of a particular term within the overall document. It is the number of times a word wi occurs in a review rj with respect to the total number of words in review rj.

$$TF(w_i, r_j) = \frac{\text{No. of times } w_i \text{ occurs in } r_j}{\text{Total no. of words in } r_j}$$

TF can be said as what is the probability of finding a word in a document.

Inverse document frequency:

The inverse document frequency is a measure of how much information the word provides, i.e., if it's common or rare across all documents. It is used to calculate the Iight of rare words across all documents in the corpus. The words that occur rarely in the corpus have a high IDF score. It is the logarithmically scaled inverse fraction of the documents that contain the word (obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient):

$$\operatorname{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

- N: total number of documents in the corpus $N = |D|$
- $|\{d \in D : t \in d\}|$: number of documents where the term t appears (i.e., $\operatorname{tf}(t, d) \neq 0$). If the term is not in the corpus, this will lead to a division-by-zero. It is therefore common to adjust the denominator to $1 + |\{d \in D : t \in d\}|$.

Term frequency–Inverse document frequency:

TF–IDF is calculated as

$$\operatorname{tfidf}(t, d, D) = \operatorname{tf}(t, d) \cdot \operatorname{idf}(t, D)$$

A high Iight in tf–idf is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the Iights hence tend to filter out common terms. Since the ratio inside the IDF's log function is always greater than or equal to 1, the value of IDF (and tf–idf) is greater than or equal to 0. As a term appears in more documents, the ratio inside the logarithm approaches 1, bringing the IDF and tf–idf closer to 0.

TF-IDF gives larger values for less frequent words in the document corpus. TF-IDF value is high when both IDF and TF values are high i.e the word is rare in the whole document but frequent in a document.

TF-IDF also doesn’t take the semantic meaning of the words.

Let’s take an example to get a clearer understanding.

Sentence 1: The car is driven on the road.

Sentence 2: The truck is driven on the highway.

In this example, each sentence is a separate document.

I will now calculate the TF-IDF for the above two documents, which represent my corpus.

Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

From the above table, I can see that the TF-IDF of common words was zero, which shows they are not significant. On the other hand, the TF-IDF of “car”, “truck”, “road”, and “highway” are non-zero. These words have more significance.

Reviewing- TFIDF is the product of the TF and IDF scores of the term.

TF = number of times the term appears in the doc/total number of words in the doc

IDF = ln(number of docs/number docs the term appears in)

Higher the TFIDF score, the rarer the term is and vice-versa.

TFIDF is successfully used by search engines like Google, as a ranking factor for content.

The whole idea is to Iigh down the frequent terms while scaling up the rare ones.

3.3.3 One-hot encoding

Sentence 1: The car is driven on the road.

If I consider that this is the entire vocabulary of my world, I have a tensor of [the, car, is, driven, on, road]. One-hot encoding simply means that I create a vector that is the size of the vocabulary, and for each word in it, I allocate a vector with one parameter set to 1 and the rest set to 0:

the – [1 0 0 0 0 0]

car – [0 1 0 0 0 0]

is – [0 0 1 0 0 0]

driven – [0 0 0 1 0 0]

on – [0 0 0 0 1 0]

road – [0 0 0 0 0 1]

I've now converted the words into vectors, and I can feed them into my network. Additionally, I may add extra symbols into my vocabulary, such as UNK (unknown, for words not in the vocabulary) and START/STOP to signify the beginning and ends of sentences. One-hot encoding has a few limitations that become clearer when I add another word into my example vocabulary: *kitty.* From my encoding scheme, *kitty* would be represented by [0 0 0 0 0 1] (with all the other vectors being padded with a zero). First, I can see that if I are going to model a realistic set of words, my vectors are going to be very long with almost no information in them. Second, I know that strong relation ship exists betIen *kitty* and *cat*, and this impossible to represent with one-hot encoding; the two words are completely different things.

3.3.4 Embedding Matrix (Word embedding)

An approach that has become more popular recently is replacing one-hot encoding with an *embedding matrix*. The Idea is to squash the dimensionality of the vector space down to something a little more manageable and take advantage of the space itself.

For example, if I have an embedding in a 2D space, perhaps *cat* could be represented by the tensor [0.56, 0.45] and *kitten* by [0.56, 0.445], whereas *mat* could be [0.2, -0.1]. I cluster similar words together in the vector space and can do distance checks such as Euclidean or cosine distance functions to determine how close words are to each other. And how do I determine where words fall in the vector space? An embedding layer is no different from any other layer I've seen so far in building neural networks; I initialize the vector space randomly, and hopefully the training process updates the parameters so that similar words or concepts gravitate toward each other.

3.3.4.1 Word2Vec

Word2Vec is a method to construct such an embedding. It can be obtained using two methods (both involving Neural Networks): Skip Gram and Common Bag Of Words (CBOW)

- CBOW Model: This method takes the context of each word as the input and tries to predict the word corresponding to the context. Consider my example: Have a great day.

  Let the input to the Neural Network be the word, great. Notice that here I are trying to predict a target word (day) using a single context input word great. More specifically, I use the one hot encoding of the input word and measure the output error compared to one hot encoding of the target word (day). In the process of predicting the target word, I learn the vector representation of the target word. Let us

look deeper into the actual architecture.

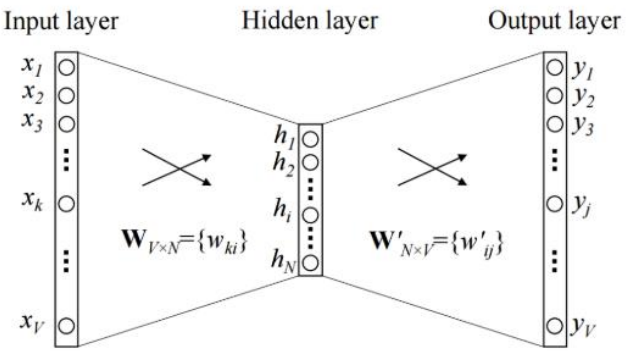


Figure 1: A simple CBOW model with only one word in the context

The input or the context word is a one hot encoded vector of size V. The hidden layer contains N neurons and the output is again a V length vector with the elements being the softmax values.

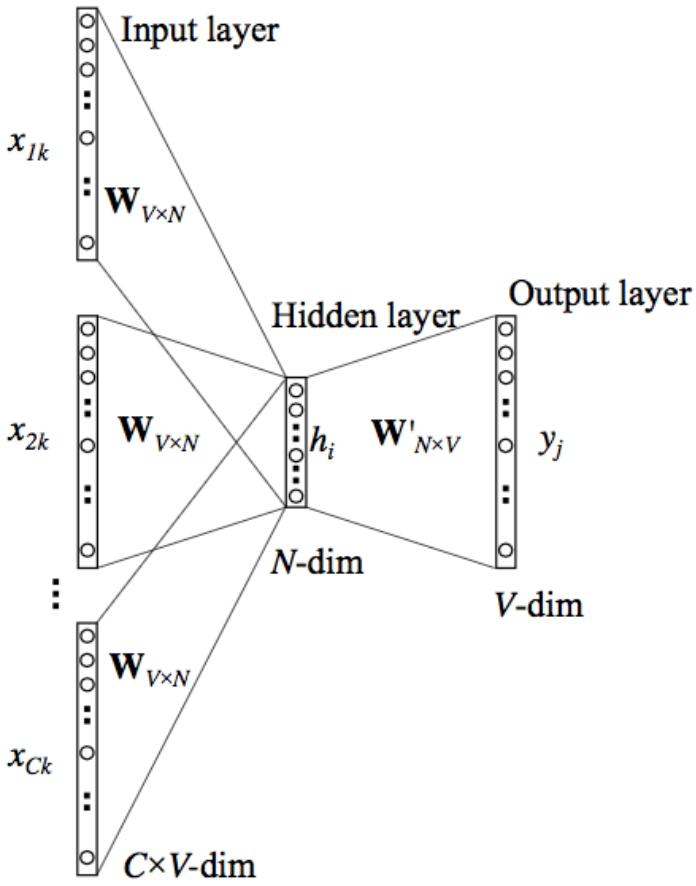
Let's get the terms in the picture right:

- W'vn is the Iight matrix that maps the input x to the hidden layer (V*N dimensional matrix)

-W'nv is the Iight matrix that maps the hidden layer outputs to the final output layer (N*V dimensional matrix)

I won't get into the mathematics. I'll just get an idea of what's going on.

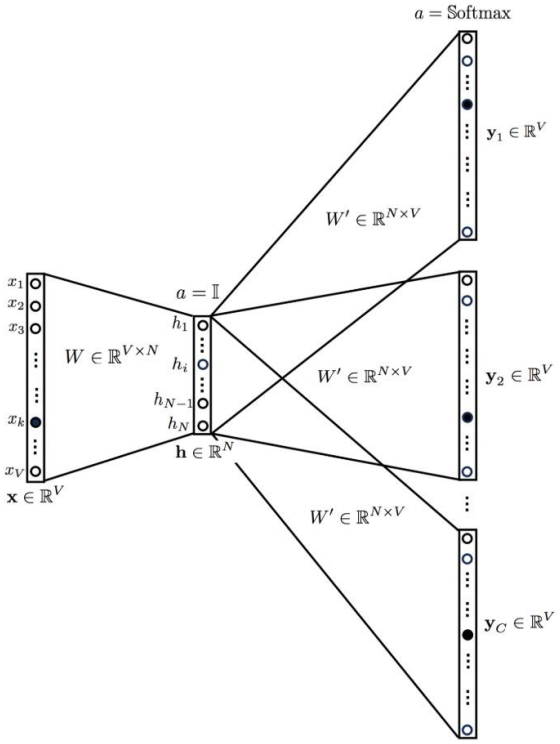
The hidden layer neurons just copy the Iighted sum of inputs to the next layer. There is no activation like sigmoid, tanh or ReLU. The only non-linearity is the softmax calculations in the output layer. But, the above model used a single context word to predict the target. I can use multiple context words to do the same.



The above model takes C context words. When Wvn is used to calculate hidden layer inputs, I take an average over all these C context word inputs.

So, I have seen how word representations are generated using the context words. But there's one more way I can do the same. I can use the target word (whose representation I want to generate) to predict the context and in the process, I produce the representations. Another variant, called Skip Gram model does this.

- Skip-Gram model:

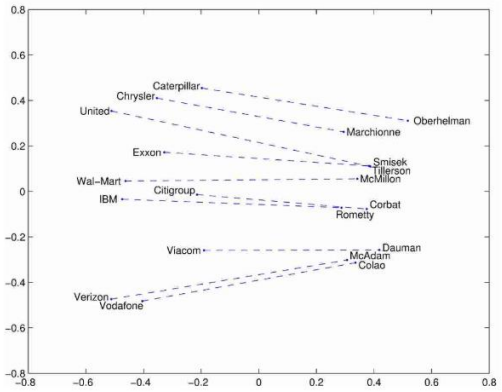


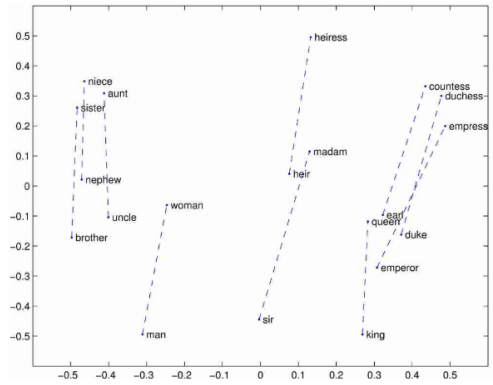
This looks like multiple-context CBOW model just got flipped. To some extent that is true. I input the target word into the network. The model outputs C probability distributions. What does this mean? For each context position, I get C probability distributions of V probabilities, one for each word. In both the cases, the network uses back-propagation to learn.

3.3.4.2 GloVe

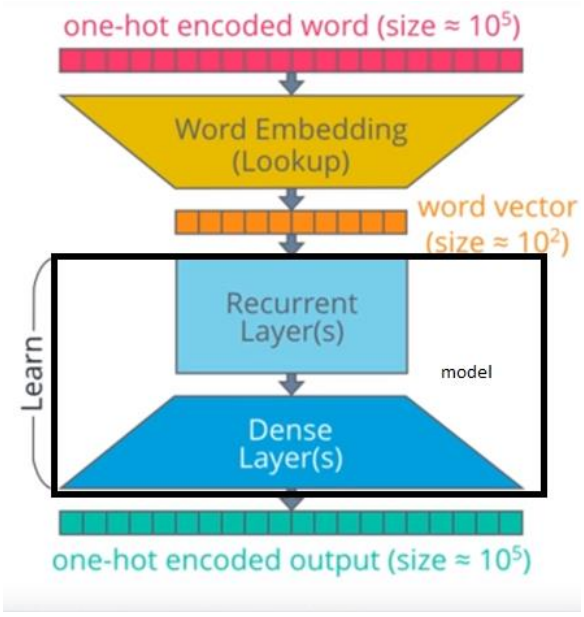
GloVe is an unsupervised learning algorithm for obtaining vector representations for words.It is developed by Stanford.Training is performed on aggregated global word-word co-occurrence from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

Examples for Linear substructures are:





3.4 Modeling



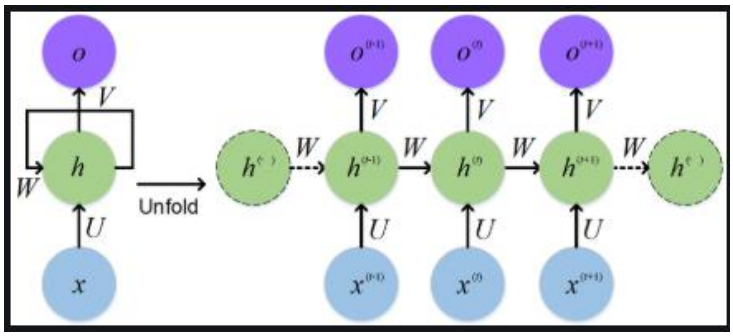
As the figure above, model means the part inside black rectangle (learnable part).

3.4.1 AWS BlazingText

The Amazon SageMaker BlazingText algorithm provides highly optimized implementations of the Word2vec and text classification algorithms. The Word2vec algorithm is useful for many downstream natural language processing (NLP) tasks, such as sentiment analysis, named entity recognition, machine translation, etc. Text classification is an important task for applications that perform Ib searches, information retrieval, ranking, and document classification.

3.4.2 Train from scratch

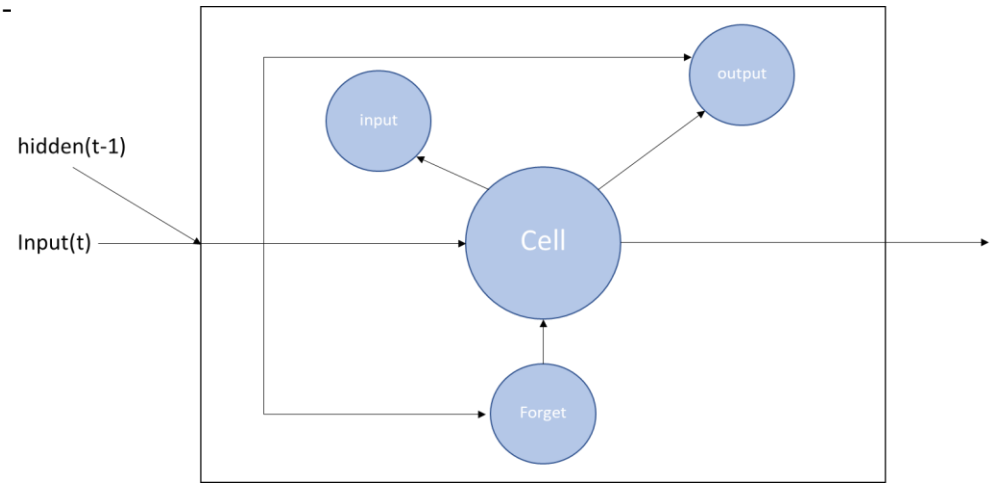
- Simple RNN model



The simple RNN structure above, I add input at a time step of *t,* and I get a *hidden* output state of *ht,* and the output also gets fed back into the RNN for the next time step. Input data is fed into the network, and next item in the sequence is predicted as output. In the unrolled view, I can see that the RNN can be thought of as a pipeline of fully connected layers, with the successive input being fed into the next layer in the sequence. When I have my completed predicted sequence, I then have to backpropagate the error back through the RNN. Because this involves stepping back through the network's steps, this process is known as backpropagation through time. The error is calculated on the entire sequence, then the network is unfolded as the right figure, and the gradients are calculated for each time step and combined to

update the shared parameters of the network. I can imagine it as doing backprop on individual networks and summing all the gradients together.

- LSTM model
-



In practice, RNNs Ire and are particularly susceptible to *vanishing gradient* problem, or potentially worse scenario of the *exploding gradient*, where ymy error tends off toward infinity. Neither is good, so RNNs couldn't be brought to bear on many of the problems they Ire considered suitable for. That all changed in 1997 when *Sepp Hochreiter* introduced the Long Short-Term Memory (LSTM) variant of the RNN. The key is to think about the three gates (input, output, and forget). In a standard RNN, I "remember" everything forever. But that's not how my brains work, and LSTM's forget gate allows us to model the idea that as I continue in my input chain, the beginning of the chain becomes less important. And how much the LSTM forgets is something that is learned during training, so if it's in the network's best interest to be very forgetful, the forget gate parameters will do so. The cell ends up being the "memory" of the network layer; and the input, output, and forget gates will determine how data flows through the layer. The data may simply pass through,

it may “write” to the cell, and that data may flow through to the next layer, modified by the output gate. This assemblage of parts was enough to solve the vanishing gradient problem, and also has the virtue of being Turning-complete.

3.4.3 Pretrained model

- ULMFit

  ULMFiT is based on a good old RNN. No Transformer in sight, just the AWD-LSTM, an architecture originally created by *Stephen Merity*. Trained on the WikiText-103 dataset, it has proven to be amendable to transfer learning, and despite the old type of architecture, has proven to be competitive with BERT and GPT-2 in the classification realm.

3.5 Evaluation

		Actual	
		Positive	Negative
Predicted	Positive	**True Positive**	**False Positive**
	Negative	**False Negative**	**True Negative**

3.5.1 Precision: ability of a classification model to return only relevant instances.

$$precision = \frac{true\ positives}{true\ positives + false\ positives}$$

3.5.2 Recall: ability of a classification model to identify all relevant instances

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

3.5.3 F1-Score**:** single metric that combines recall and precision using the harmonic mean.

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

3.5.4 Accuracy

$$Accuracy = \frac{true\ positives + true\ negative}{true\ positives + true\ negative + false\ positives + false\ negative}$$

4. Hands-on Experiment

I used my local PC with RTX 2070 GPU for these experiments.

4.1 Data collecting

As you can see the code in my Github repo "twintscraping.ipynb" I used TWINT as the collector, and search the twitter that hash tagged the world related to depressed and depression. Then labeled depressed tweets as "1" and saved as "depresstweets.csv". In "textprocessing.ipynb" I used pandas's concat to concatenate the depress tweets with normal (positive feeling) tweets from "sentiment140.csv" (the sentiment140 dataset, this file is too big, so I can't uploaded to my Github repo). And saved as "sentiment_tweets3.csv".

4.2 Experiment 1

My first experiment is the first part of "GloVe.ipynb"

I separated training set and testing set in the ratio of 80:20.

4.2.1 Feature Extraction

GloVe as word embedding (840B, 300 dimension)

4.2.2 Modeling

Simple RNN model.

```
class simpleRNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, outp
ut_dim):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):

        #x = [sent len, batch size]

        embedded = self.embedding(x)

        #embedded = [sent len, batch size, emb dim]

        output, hidden = self.rnn(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        out = self.fc(hidden)
        return out
```

Using Adam as an optimizer and Cross entropy loss as loss for doing backpropagation.

After 100epochs of training, the train loop was stopped. And next section is model evaluation.

4.2.3 Evaluation

	precision	recall	f1-score	support
0	0.00	0.00	0.00	1600
1	0.22	1.00	0.37	462
accuracy			0.22	2062
macro avg	0.11	0.50	0.18	2062
weighted avg	0.05	0.22	0.08	2062

As you can see, so low. So, we will not use this method for our model deployment.

4.3 Experiment 2

My second experiment is the second part of "GloVe.ipynb"

I separated training set and testing set in the ratio of 80:20.

4.3.1 Feature Extraction

GloVe as word embedding (840B, 300 dimension)

4.3.2 Modeling

LSTM - RNN model.

```
class lstmRNN(nn.Module):
    def __init__(self, hidden_size, embedding_dim, vocab_size):
        super(lstmRNN, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_di
m)
        self.encoder = nn.LSTM(input_size = embedding_dim,hidden
_size = hidden_size, num_layers = 1)
        self.fc = nn.Linear(hidden_size, 2)

    def forward(self, x):

        #x = [sent len, batch size]

        embedded = self.embedding(x)

        #embedded = [sent len, batch size, emb dim]

        output, (hidden,_) = self.encoder(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        preds = self.fc(hidden.squeeze(0))
        return preds
```

Using Adam as an optimizer and Cross entropy loss as loss for doing backpropagation. After 100epochs of training, the train loop was stopped. And next section is model evaluation.

4.3.3 Evaluation

	precision	recall	f1-score	support
0	0.93	1.00	0.96	1600
1	0.99	0.73	0.84	462
accuracy			0.94	2062
macro avg	0.96	0.86	0.90	2062
weighted avg	0.94	0.94	0.93	2062

Yeah, great. But I have to compare with others method for choosing the model deployment.

4.4 Experiment 3

My third experiment is the first part of "Fasttext.ipynb"

I separated training set and testing set in the ratio of 80:20.

4.4.1 Feature Extraction

Fast text (for English language) as word embedding.

4.4.2 Modeling

Simple RNN model.

```
class simpleRNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, outp
ut_dim):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):

        #x = [sent len, batch size]

        embedded = self.embedding(x)

        #embedded = [sent len, batch size, emb dim]

        output, hidden = self.rnn(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        out = self.fc(hidden)
        return out
```

Using Adam as an optimizer and Cross entropy loss as loss for doing backpropagation.

After 100epochs of training, the train loop was stopped. And next section is model evaluation.

4.4.3 Evaluation

	precision	recall	f1-score	support
0	0.94	0.08	0.15	1600
1	0.24	0.98	0.38	462
accuracy			0.28	2062
macro avg	0.59	0.53	0.26	2062
weighted avg	0.78	0.28	0.20	2062

Such a poor score, so I will definitely not choose this model for my deployment.

4.5 Experiment 4

My fourth experiment is the second part of "Fasttext.ipynb"

I separated training set and testing set in the ratio of 80:20.

4.5.1 Feature Extraction

Fast-text as word embedding (840B, 300 dimension)

4.5.2 Modeling

LSTM - RNN model.

```
class lstmRNN(nn.Module):
    def __init__(self, hidden_size, embedding_dim, vocab_size):
        super(lstmRNN, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_di
m)
        self.encoder = nn.LSTM(input_size = embedding_dim,hidden
_size = hidden_size, num_layers = 1)
        self.fc = nn.Linear(hidden_size, 2)

    def forward(self, x):

        #x = [sent len, batch size]

        embedded = self.embedding(x)

        #embedded = [sent len, batch size, emb dim]

        output, (hidden,_) = self.encoder(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        preds = self.fc(hidden.squeeze(0))
        return preds
```

Using Adam as an optimizer and Cross entropy loss as loss for doing backpropagation. After 100epochs of training, the train loop was stopped. And next section is model evaluation.

4.5.3 Evaluation

	precision	recall	f1-score	support
0	0.93	1.00	0.96	1600
1	0.99	0.73	0.84	462
accuracy			0.94	2062
macro avg	0.96	0.86	0.90	2062
weighted avg	0.94	0.94	0.93	2062

Also got high score, but less than method using GloVe as word embedding with LSTM model.

I did all model from scratches .I will compare with build-in algorithm like AWS BlazingText and recently got high benchmark for sentiment classifier ULMFiT pretrained model.

4.6 Experiment 5

My fifth experiment is “ulmfit.ipynb”

I separated training set and testing set in the ratio of 80:20.

4.6.1 Modeling

LSTM based ULMFiT model from fasiai.

4.6.2 Evaluation

col_0	0	1
row_0		
0	1573	109
1	27	354

By this confusion metric, I got 0.934 of accuracy, 0.929 of precision, 0.764 of recall.

And 0.838 of F1-score. Quite high but still lower than our second experiment.

4.7 Experiment 6

My sixth experiment is “blazingtext_comparison.ipynb”

I separated training set and testing set in the ratio of 80:20.

4.7.1 Modeling

AWS built-in algorithm “Blazing Text”

4.7.2 Evaluation

	precision	recall	f1-score	support
0	0.78	1.00	0.87	1600
1	0.00	0.00	0.00	462
micro avg	0.78	0.78	0.78	2062
macro avg	0.39	0.50	0.44	2062
weighted avg	0.60	0.78	0.68	2062

Still lower than our second experiment.

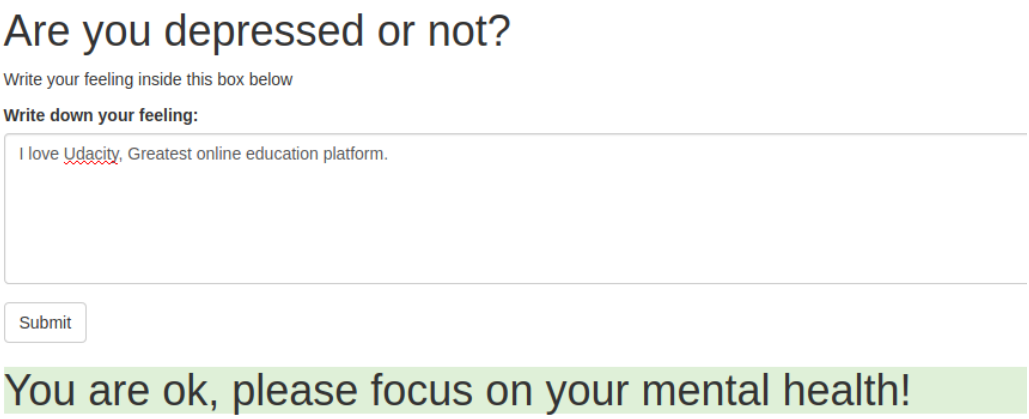
4.8 Choose the best model for deployment

So, I got the best method same as experiment 2. Then will use that model for our deployment. Feature engineer using GloVe and use LSTM-RNN model.

5. Deployment

I used the method same as the first project of this Udacity Nanodegree, deploy using SageMaker, Lambda, Gateway API. And all codes contained in my Github repo ("SageMaker Project.ipynb" file and "serve", "train", "website" folders.

And below is an example of my web app.



I still can love, so I think this sentiment analysis model predicted it correctly.

6. Conclusion

As we can see in my jupyter notebooks, simple RNN training loss didn't decrease in every training epoch. This is because of "Vanishing Gradient". So As we can compare to LSTM-RNN, the vanishing gradient problem were removed. Then, ULMFiT also use LSTM structure in it. But the reason why we got lower evaluated score than our simple LSTM is because of word embedding method. Pre-built word embedding also very important for making the scores higher. In this project I used only "GloVe" and "Fast text", so if I have more times, I would like to try other embedding method too.

Lastly, for deployment AWS make everything easier than before.

Future work: Make this model ready for Thai language (my mother language), scraping more tweets, pay attention to data labeling (in this project because of times limitation, I labeled data so fast without double check from psychologist).

7. References

1. Rafael A. Calvo, et al., “Natural language processing in mental health applications using non-clinical texts”, 2016

2. Moin Nadeem, et al., “Identify Depression on Twitter”

3. https://towardsdatascience.com/analyzing-tIets-with-nlp-in-minutes-with-spark-optimus-and-twint-a0c96084995f?smyce=bookmarks---------0----------

4. https://medium.com/analytics-vidhya/fundamentals-of-bag-of-words-and-tf-idf-9846d301ff22

5. https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

6. O’Reilly, Programming PyTorch for Deep Learning (book)