Twitter data sentiment analysis

Chen Xing cxing6@wisc.edu
Jitian Zhao jzhao326@wisc.edu
Lize Du ldu43@wisc.edu

Abstract

In this project, we focused on tweets sentiment analysis problem. We implemented several models using pytorch package and improve the models step by step. First, we chose two basic models to serve as our baseline: Naive Bayes and multi-layer perceptron. These two models achieved acceptable accuracy (70% for naive Bayes and 80.5% for mlp) considering its speed (1 min/epoch).

Since mlp doesn't take the order of words into consideration, we considered using CNN to correct this drawback and reached 84% with 2 min/epoch.

We also tried RNN and LSTM as they are popular methods dealing with text problems. However, our RNN model took us more than 4 hours to train an epoch and we couldn't wait for its convergence. The gradient vanishing problem became extremely severe when doing back-propagation of RNN. Thus, we tackled gradient vanishing problem with LSTM. We first tried a simple one-direction LSTM and reached an accuracy of 81% and speed around 1 hour an epoch. Then we improved our model into a bidirectional LSTM and achieved 86.77% accuracy while taking more than 3 hours an epoch.

GRU came into the picture when the speed became an important concern since it contains fewer parameters. This model gave us an accuracy around 86.33% and speed (1.5 h/epoch) much faster than bidirectional LSTM.

1. Introduction

With the development of smart phones and all kinds of social apps, people are generating more and more data on the social media platforms in recent years. Doing sentiment analysis with these data can provide us with countless benefits. First of all, this procedure can help us transform unstructured information into structured data of public opinions towards all kinds of products, services, brands and so on. For example, companies can see the attitudes of people towards a certain product and adjust their product according to the results of sentiment analysis.

And what's even better is that this procedure is totally automatic. We'll no longer need to read all the texts by ourselves if we want to do research on a certain product, all we need to do is simply download the text data and send them into this 'sentiment analysis' system and wait for the result. In this project we constructed different models to analyze the sentiment of tweets: input a tweet text and give us a binary classification of negative or positive. To achieve this, we simply split our projects into 4 procedures: data-cleaning, word embedding, model implementation and result comparison.

Firstly, since we used the tweet data that contains certain features(such as @user and emojis) and contain a lot of misspelling, the first thing we did is data-cleaning.

After cleaning the data into a more 'standard' form, next problem we need to figure out is how to do word embedding: transform words into numeric vectors. This is not as trivial as what we did in image recognition where images can automatically be transformed into gray-scale matrices. Also, the input sentences might be of different lengths, we need to force them to be of the same size at a certain stage no matter which deep learning model we decide to use.

With the numeric word-embedding matrix, we can now apply this matrix to different models. And finally, after doing experiments with different kinds of models, we are able to compare their performances.

The following content will show why we used certain method to do the word embedding and how did we implement all the models step by step, from simple machine learning model to complex neural network.

2. Related Work

The dataset we used for modeling is sentiment 140, which contains 1.6 billion of tweets. There are many literature using this dataset to do sentiment analysis. Some of them implemented traditional machine learning model. Goel, Ankur used Naive Bayes to do sentiment analysis on Sentiment 140.[4]. Fang, Xing also tried a combination of rule classifier and SVM.[3]. Some use deep learning method to deal with this problem. Cunanan, Kevin use deep learning models sunch as RNN and LSTM and get an accuracy of 74.55%. [2].However, we didn't find a literature that focus on sentiment 140 dataset and compare the machine learning and deep learning models and that is what

our project does.

While implementing the deep learning models, the code material we referred to most is the tutorial written by Ben Trevett (`https://github.com/bentrevett/pytorch-sentiment-analysis`), which provides us a comprehensive understanding of doing sentiment analysis with pytorch. He implemented several models including RNN, CNN, fastext and trained the models using IMDb dataset in torchtext package. For the simplest RNN model, his accuracy reached 47%. After changing ways of doing word embedding (use pre-trained glove vocabulary), and trying some other architectures, he finally reached an accuracy of 85% using CNN. Our project used a different dataset and spent quite a lot time reading the torchtext file about 'datafield' since his dataset already comes from the package and has lot of pre-installed methods.

3. Proposed Method

3.1. Naive Bayes

The reason why we used Naive Bayes is that we hoped it can serve as a baseline which can helped us to compare the different performance between machine learning method and deep learning method. We just used the data processed using NLTK[1] and choose 371 words whose term frequency are larger than 6000 as our model's features. The advantage of Naive Bayes is that it is fast and simple; The disadvantage of it is that its accuracy is relatively low.

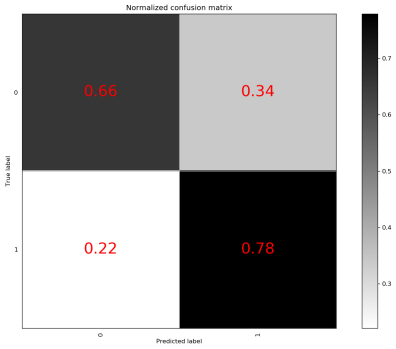


Figure 1. Confusion matrix of Naive Bayes model

3.2. Deep learning model

3.2.1 Multilayer perceptron

Since Naive Bayes is a machine learning method. We also select a simple deep learning model: multi-layer perceptron to serve as our baseline. The architecture we implemented is quite simple (Figure 2). First, a max pooling layer will force different sizes inputs to have the same sizes outputs. The word embedding procedure will give us a embedding matrix of size: length of text times embedding dimension. Since our input texts might be of different lengths, the embedding matrices will be of different size accordingly. Max-pooling layer can solve this problem by using different sizes of kernels to scan the embedding-matrix. After the max-pooling, we use 2 hidden layers to get the final output.

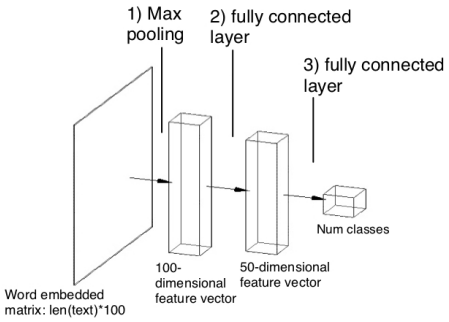
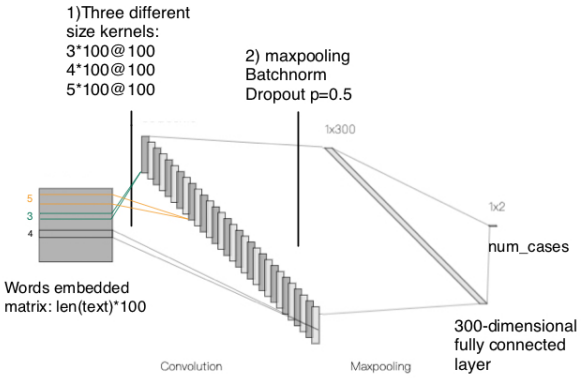


Figure 2. Architecture of multi-layer perceptron

3.2.2 Convolutional neural network

Figure 3. Architecture of CNN



Unlike Multi-layer perceptron, convolutional neural network will take the order of words into consideration. The architecture we implemented is from this paper: *Convolutional Neural Networks for Sentence Classification*[5]. The basic idea of this model is using 3 different sizes of kernels to scan the embedding matrix. Then use max-pooling layer to make the size match and a fully-connected layer to get the final output.

3.2.3 Recurrent neural network

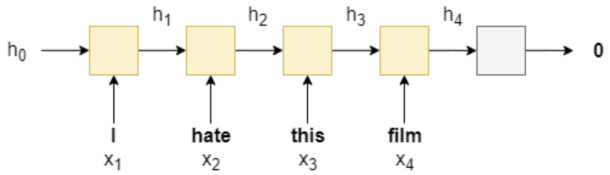
Since the data we deal with is text data, where the words are dependent within one sentence, RNN is a common

method to interpret the dependency in the model. This is important because the previous word may include crucial information about what is coming next. An RNN takes a sequence of words $X_1, X_2...X_T$ each at a time, and for each time t calculates a hidden state h_t. h_t depends on the hidden state h_{t-1} and X_t.

$$h^{<t>} = \sigma_h(W_{hx}X^{<t>} + W_{hh}h^{<t-1>} + b_h)$$

where σ_h is the activation function.When we reach the final state X_T, we send the output from the last state to next layer(in our model a fully connect layer).

Figure 4. Architecture of RNN



Source: https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb

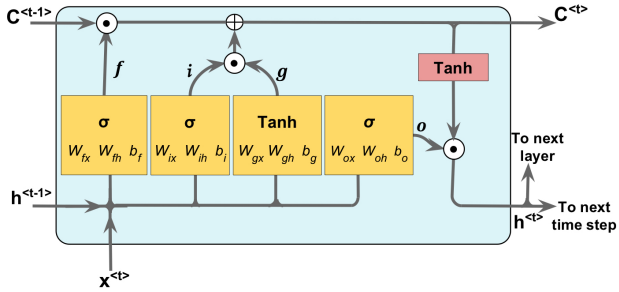
In the figure above, the RNN layer is shown in the yellow box and the fully connect layer is shown in grey. A common problem for RNN is that the gradient is easy to vanish or explode since it back-propogates through time.

Our RNN model includes one RNN layer and one fully connect layer. Drop out is also added after the recurrent layer with the drop out rate equals to 0.5.

3.2.4 LSTM

LSTM(Long short-term memory) is a popular solution to the gradient vanishing problem in RNN. LSTM is an extension for RNN since it enables RNN to remember their input for a long period of time.

Figure 5. LSTM cell

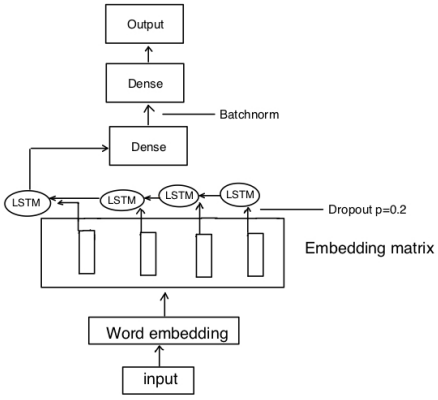


Source:https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L14_intro-rnn/L14_intro-rnn-part2_slides.pdf

The LSTM cell takes the hidden state of the previous time point $h^{<t-1>}$ and the cell state of the previous time point $c^{<t-1>}$ and output the cell state $c^{<t>}$ and the hidden state $h^{<t>}$. LSTM is able to have a long and short term memory by using 3 gates: input, forgot and output gate. Input gate whether or not let the input in; the forgot gate delete the information that is not important; the output gate determine whether the information will impact the output at present state or not. Moreover, since all the gate is activated through a sigmoid function which ranges from 0 to 1. The gradient vanishing problem is solved using LSTM since it keeps the gradient steep enough. Therefore, LSTM usually has relatively high accuracy and short training time compared with traditional RNN.

Two kinds of LSTM models were implemented in this project. The first is normal LSTM which simply consists of one forward RNN layer. The second is bidirectional LSTM which consists of two RNN layer: one forward RNN layer which process the sentence from the first word to the last word and one backward RNN layer which process the sentence from the last word to the first word.

Figure 6. Architecture of Normal LSTM Model



The normal LSTM model we constructed includes one recurrent layer with LSTM cell and two fully connect layer. A drop out with drop out rate equals to 0.2 is added to the recurrent layer and the batch norm is added to between the two fully connect layer.

The bidirectional LSTM model we constructed includes a forward recurrent layer and a backward recurrent layer. The output of the two layers is then merged together and feed into a fully connect layer. Dropout method with drop out rate equals to 0.5 is also before the fully connect layer.

3.2.5 GRU

GRU(Gated Recurrent Unit) is another popular solution to the gradient vanishing problem in RNN. Compared with LSTM, GRU get rid of the cell states and only use the hidden state $h^{<t>}$ to transfer information. GRU only consists of two gates: a reset gate and a update gate. An update gate is similar to the input and forgot gate in LSTM. It decides

Figure 7. Architecture of Bidirectional LSTM Model

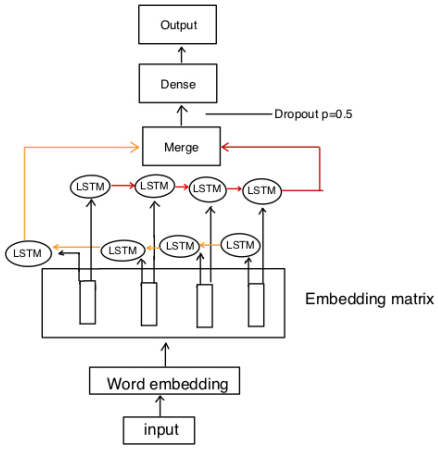
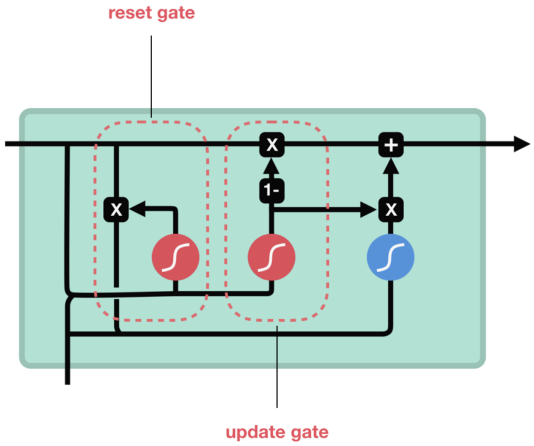


Figure 8. GRU cell



Source:https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

which information the model should add and which information the model should throw away. The reset gate is used to decide how much information the model should forget.

GRU model usually trains faster than LSTM model since it has few parameters to train and thus has fewer tensor operations. However, in real world, it is usually unclear which one of the two will work better.

The GRU model we constructed consists of one recurrent layer with GRU cell and one fully connect layer. Drop out method with drop out rate equals to 0.2 is added to the recurrent layer.

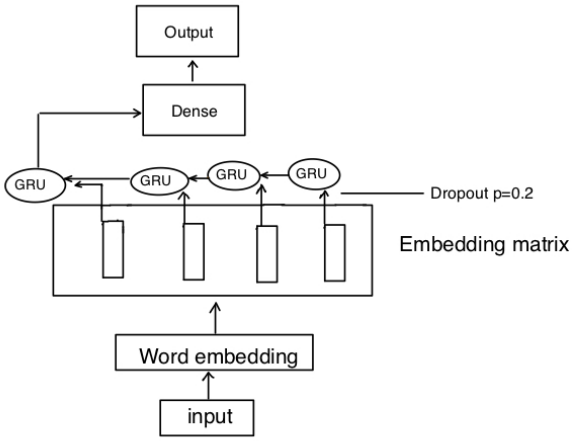
4. Experiments

4.1. Data set

The data set we use is Sentiment 140 which was created by Alec Go,Richa Bhayani,Lei Huang who were Computer Science graduate students from Stanford university.

The data set contains lots of features but we only use the polarity of the tweets and the text of the tweets in our model(1.6 billion). In particular, the polarity of the tweets has 2 different kinds of labels : negative and positive.

Figure 9. Architecture of GRU Model



4.2. Data Prepossessing

Because the raw data contains too much useless information and it may influence our models' performance, so we planed to do some data cleaning work first. The main tool we used is the $nltk$ package and we used regular expression to do pattern matching.

- Stopwords use $nltk.stopwords.words$ except:

  1 Adverb of degree: few, most, more...

  2 Negative: dont, didnt, doesnt arent...

  The reason why we excluded above two parts is that the adverb of degree can denote very strong emotion, and we thought it can help us to predict more precisely; We also wanted to use the information of 'not'. For example, when we say 'I like it', it is definitely a positive attitude, but 'I don't like it' is totally different. For the negative words, we found a special method in $nltk$ called $marknegation$, which can add a tag to the words between not/never and the first punctuation after them. But we only adopted this method for Naive Bayes, because for other models, we useed a pre-trained dictionary to do word embedding, and the dictionary can't detect this tag.

- Pattern matching: words, abbreviation, [a-zA-Z]-[a-zA-Z], $\cdots$, ?, !

  we want to contain all the words, abbreviations like U.S.A., and some words like 'built-in', also some punctuation which can denote strong emotion.

- Substitution: n'/n't→not, 'd→would...

  because the algorithm can only detect not, nut cannot detect don't; for 'd, we wanted to make sure to delete all the stopwords.

- Delete: nouns, number+th/st/nd/rd;

  because we thought they are not meaningful.

- Change to lower case;

- Use porter stemmer to do stem extracting, such as amazing→amaz(naive bayes);

- Use wordnet lemmatizer to lemmstize the verb to a normal form, such as loving→love

We also noticed that people tend to use some certain expressions when they tweet, so we do following substitution to match the dictionary we use:

- '!!!!!' to '!'<repeat>

- website to <url>

- @user to <user>

- number to <number>

- 'loveeeee' to 'love'<elong>

4.3. Embedding

We used the same embedding method for our models except Naive Bayes, we used the glove.twitter.27B dataset as our pre-trained embedding dictionary, the reason why we used it is that this dataset is designed for analysing twitter data and its data size is large enough which has 1.2 million vocabularies, so we can 'look up' almost all the words appear in tweets. Glove[6] is a method designed by Jeffrey Pennington, Richard Socher and Christopher D. Manning, this method not only considers the relationships between words, but also extracts the linear structure between words, so its performance is better than the bags of words model which we adopt with our Naive Bayes model.

4.4. Experiement Setup

Each experiment was carried out under identical condition. For all the text data, we used regular expression, stemming and other method to clean the data. Then we used customized tokenizer and "glove" pre-trained vectors to do word embedding. Words not present in the set of pre-trained words are initialized randomly. After embedding, the 1,600,000 tweets were converted to a $1,600,000 \times 100$ matrix. The embedding matrix are fed to the model.

To begin our experiment, we randomly split the data to three parts, training set, validation set and test set which consists of 70%,20%,10% respectively. The split is stratified to make the target variable balance within each set.

After splitting the dataset into training, validation and test set, we used the training set, and implemented Naive Bayes, MLP, CNN, RNN, LSTM(normal & bidirectional), GRU. Since our target value is binary, we used binary cross entropy to be our loss function. During training, we first froze our embedding layer to make use of the pre-trained word vectors. Once we saw our validation accuracy keeps decreasing. we back-propagated and updated our embedding layer so that we can tune our embedding layer and get a better world vector based on our own data. We also used the validation accuracy to perform early stopping method to avoid over-fitting. In particular, as long as we saw that the validation accuracy remained the same or even decreasing, we stopped our training and chose the model with the highest validation accuracy to be our final model.

We evaluated our model from two perspectives: efficiency and accuracy. In particular, we record the time for each epoch for each model to show its efficiency(since all of our model converges in 5 epochs). We interpreted the accuracy of the model using the prediction accuracy on the test set. We believed that a good model should yield a high accuracy in a relatively fast training.

4.5. Software

Python 3.7 Juyter notebook

4.6. Hardware

Google Colab

5. Results and Discussion

Table 1. Result Comparison

model	run-time	test accuracy
Naive Bayes	really fast	71.9%
MLP	$\approx$ 1 min/epoch	80.5%
CNN	$\approx$ 2 min/epoch	84.15%
RNN	$>$ 4 hour/epoch	/
Normal LSTM	$>$ 50 min/epoch	81.27%
Bidirectional LSTM	$>$ 3 h/epoch	86.72%
GRU	$\approx$1.5h/epoch	86.33%

The table above shows the time for training of each epoch and the test accuracy. The estimate of time may not be very accurate since we use google colab as our cloud computing engine and the connection to google colab isn't really stable so that the time of the training may also be affected by the internet connection besides model complexity.

Naive bayes yields a testing accuracy of 71.9%. All the deep learning models perform better than it, showing that

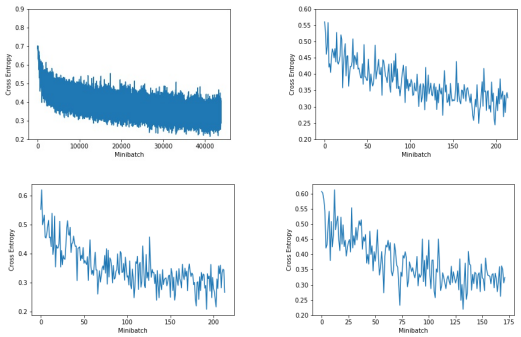


Figure 10. The change of cross entropy of MLP, CNN, LSTM (bidirectional), GRU when training.

the deep learning model is actually more suitable than machine learning method for this kind of problem. Besides, CNN, RNN, LSTM and GRU performs better than MLP since they all take the order of words within one sentence into consideration.

As for the training time, we can see that MLP and CNN both train really fast since the model are both simple. RNN, LSTM, GRU all train quite slow since back-propagates through time is quite time-consuming. GRU trains a bit faster than LSTM since it has fewer parameters to train.

In practice, when taking both the training time and the test accuracy into consideration, GRU is a good method. Moreover, if we aren't so picky about getting the highest accuracy, CNN is also a good choice since it yields an acceptable result and train really fast.

6. Conclusions

In this project, we focused on tweets sentiment analysis problem. After data processing and embedding,we implemented several Machine learning and deep learning models and tried to find the best model for our problem. First, we chose two basic models to serve as our baseline : Naive Bayes and multi-layer perceptron. These two models achieved acceptable accuracy (70% for naive Bayes and 80.5% for mlp) considering its speed (1 min/epoch).Since mlp doesnt take the order of words into consideration, we considered using CNN to correct this drawback and reached 84% with 2 min/epoch.We also tried RNN and LSTM as they are popular methods dealing with text problems. However, our RNN model took us more than 4 hours to train an epoch and we could not wait for its convergence. Due to RNNs back-propagation through time, the gradient vanishing problem became extremely severe. Thus, we tackled gradient vanishing problem with LSTM. We first tried a simple one-direction LSTM and reached an accuracy of 81% and speed around 1 hour an epoch. Then we improved our model into a bidirectional LSTM and achieved 86.77% accuracy while taking more than 3 hours an epoch. GRU came into the picture when the speed became an important concern since it contains fewer parameters. This model gave us an accuracy around 86.33% and speed is (1.5 h/epoch) much faster than bidirectional LSTM.
After comparing the efficiency and accuracy of all the model we get, we decide to use GRU model as our best model with 86.33% accuracy and with 1.5 training time for each epoch. Moreover, if we are not required to get the highest accuracy,CNN is also a good choice since it yields an acceptable result and train really fast.

7. Acknowledgements

We are glad to acknowledge Ben Trevett `https://github.com/bentrevett/pytorch-sentiment-analysis`, who provides a tutorial on sentiment anaylysis. Besides, we would like to acknowledge Alec Go, Richa Bhayani, and Lei Huang who build sentiment 140 which provides us with a well labeled sentiment dataset.

8. Contributions

Data cleaning: Lize Du
Deep learning model constructing: Jitian Zhao, Chen Xing
Machine learning model constructing: Lize Du
Report writing presentation: Jitian Zhao, Chen Xing, Lize Du
For preparing the material of report and presentation, we split the work according to the models we constructed.

References

[1] E. L. Bird, Steven and E. Klein. Natural language processing with python. 2009.

[2] K. Cunanan. Developing a recurrent neural network with high accuracy for binary sentiment analysis. 2018.

[3] X. Fang and J. Zhan. Sentiment analysis using product review data. *Journal of Big Data*, 2(1):5, 2015.

[4] A. Goel, J. Gautam, and S. Kumar. Real time sentiment analysis of tweets using naive bayes. In *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*, pages 257–261. IEEE, 2016.

[5] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[6] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. pages 1532–1543, 2014.