Are Emoji Predictable based on the Text from Tweet?

CHEN, Hsuan-Yu
National ChengChi University
g1710171@gl.aiu.ac.jp

1 Introduction

As the development of technology, people's demand for 'accompany' gradually transferred from real world to 'virtual' world. And then came the social network software, such as Facebook, Line. Twitter, which has a 'quite' different form for user to post, is one of the growing social network software. With its 'special' limitation for the text form in 'Tweet', Twitter has 'recently' became research target for lots of researchers in the NLP(Natural Language Processing) domain. Not only its short text form interested the researcher, but also the immense usage of 'Emoji'. In this paper, I will mainly dig into the relation between the text and the emoji, and try to find if there is some unveiled pattern afterwards through visualization ,sentiment analysis, and other machine learning algorithms.

2 Dataset

After targeting the Twitter, we started to collect the Tweet from the Twitter. Through the API provided by Twitter, one can streaming the tweets that containing the keyword interested with Tweepy module. Then it came to the first question, how emoji was "actually" represented by the computer? Then I looked into the official emoji website concerning coding. They provided the unicode that representing each emoji. And this was also the first time I learned about unicode, from why it is necessary and how it is implemented in programming languages. After knowing the system, I started to collect the Tweet. Totally, I collected data twice, the first one was a small set only has 8000 tweets in order to build the analysis workflow faster first. After I built the workflow, I collected a bigger set again having about 45000 tweets this time. And I will explain how I use these two data sets in the following.

3 Methodology

3.1 Applied Module

Python - Tweepy, Pandas, Numpy, t-SNE, scikit learn, TensorFlow, Xgboost
R - ggplot2, ggimage, emoGG

3.2 Total Workflow

The basic workflow of my analysis started from preprocessing the text data, visualization, then goes to analysis, which is the classical way in machine learning project. However, due to the various problems machine learning are going to solve, times required by each step varies a lot. In the natural language processing domain, it is a common sense that it is the preprocessing data part takes the most of the time instead of building the model. It is because real datas unlike those classical data set, such as iris, which are mainly has been arranged in a clear form for user to analyze. Real data comprised of various types of data from categorical to numeric and missing data is also very common. How to tackle these problems is the first obstacle for us. Let's start with collecting and preprocessing data first!

4 Preprocessing

As mentioned in the data set parts, Tweets can be collected based on the keywords somebody are interested. After the collection, I looked into the structure of data first. Even text is the main content we see on the Twitter. A Tweet actually has more information, such as the post time, post id, post location, etc. Hence, by transforming the Tweet from JSON to dictionary type in python, one filtered out the needed information easily. After getting about 8000 text data from the original data, I started to "tokenize" the Tweet, which means to separated each word in a text as a element, with the nltk module. Due to the various content in the text, there are actually lots of unrelated information to our analysis, such as URL, marks, etc. Besides, I also filtered out lots of pronouns that may affect our analysis with the stop word list that included in the nltk module. After the above steps, data know only consisting emoji and important words in each Tweet. Afterwards, I built a dataframe with the emoji and words data that each emoji as emojis and each words as columns. The structure of this data frame is 7175 rows and 263 columns, and containing only 1 or 0 to represents if a word has appeared with a emoji together.This is our preliminary data for the visualization part. We can get a clearer idea by the graph of data frame below.

	445	98b	3df	48a	3dd	5a5
ab	0	0	0	0	0	0
abarklk	0	0	0	0	0	0
abby	0	0	0	0	0	0
abc	0	0	0	0	0	0
abetting	0	0	0	0	0	0
abhay	0	0	0	0	0	0
able	0	0	0	0	0	0
absolute	0	0	0	0	0	0
absolutely	0	0	0	0	0	0
absurd	0	0	0	0	0	0
abt	0	0	0	0	0	0
abu	0	0	0	0	0	0

Figure1. Part of the word-emoji data frame

Visualization

In the actual problem, there are lots of information we can get only by visualization, such as the distribution or the frequency or the basic statistics of data. Graph can tell us more than we imagine based on how we implement it. And as the visualization tools growing rapidly in the machine learning field. One of the powerful tools is ggplot2, which is package invented by Hadley Wickham in 2005. It has become one the most popular packages in R recently. And because it is open source, more and more additional graphic tools based on ggplot2 are published now. Hence, I decided to use ggplot2 in R as my visualization tool instead of matplot in python.

In order to plot the emoji in out data, because of the high dimension in our original data, we have to reduce the dimension to 2D first. Traditionally, when talking about dimension reduction, people come up with principal component first. And one of the important idea of PCA is that we should be able to catch most of the information only by few component. Hence, I tried the PCA first. Nevertheless, I only got about 50% information in the first and second components, which was not a good result compared to others application of PCA. To figure the possible reason of this outcomes, I looked back to the theory of PCA. And the core purpose of PCA is to capture the "linear relation" between the variables, such as the stock price between lots of companies. Thus, this concept was no doubt unsuitable to our data. When confusing about this problems, I read the paper from Laurens van der Maaten, which implement a state-of-art dimension reduction method "t-SNE". Unlike PCA, t-SNE reduce the dimension based on nonlinear function. I thought this fit my data better than the PCA. Hence, I implemented the t-SNE to reduce the original data to two dimensions. Next, I plotted the emoji based on the data and got the graph below.

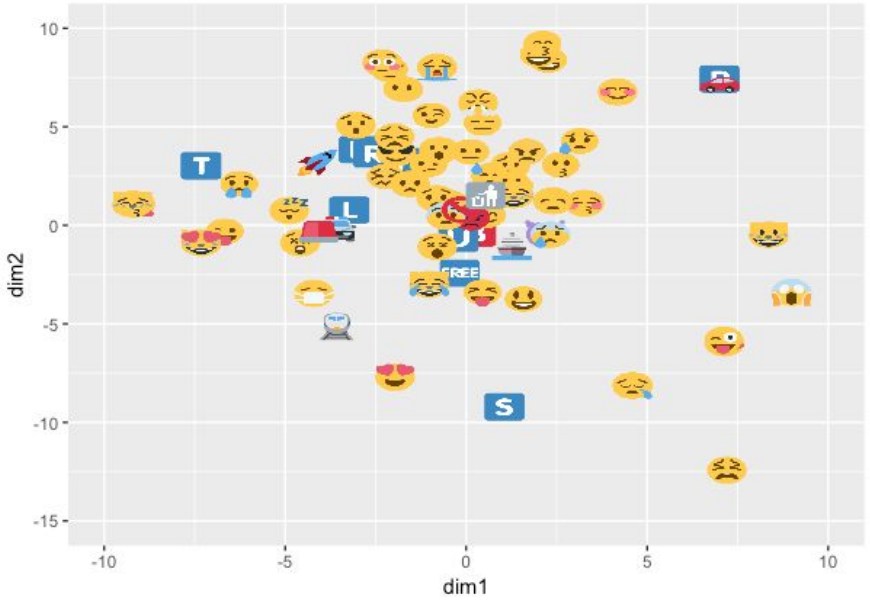

Figure2. Emoji relation plot

Even my data set only has 8000 tweets, I still got some interpretable results. We can observe that those negative emotion emoji such as angry, sad, crying, gather around the center much than those positive emotion emoji. We can suspect that there are some representative words for negative emotions. Those representative words were widely used in expressing negative emotions. Thus, location of those negative emojis are close. On the other hand, those positive emojis mainly diverged in the graph. From this, we can suspect that words used to express positive emotions are more various than negative emotions, causing their distance between each other are more distant. This is what we can basic information that we can get from the visualization. However, there is a unsolved technical problem. Because there were some emoji unable to accessed through the packages, emojis here only have about 35 percent of the original data. Maybe we can get better results after we solve this problem or a more simple way, get more data.

5 Approrach

5.1 Algorithm Introduction

Before the analysis, I reflect my topics again. My target is to find if we can predict the emoji based on the text, which is an typical classification problem that based on several variables to predict the class. Normally, we will have lots of variables and few class to be predicted. However, in my case, I have about 7000 variables(words) and 263 class(emojis) now, which is a very unbalanced ratio between X and Y. And we imagine that if we build the model directly, we will definitely get terrible outcome. Therefore, I decided to refine the problem again.

I looked in the current data and compute the frequency for each emoji. The top three frequent emojis are “Tears of Joys”, “Heart Eyes” , and “Crying”.We can see from the graph below. Besides, people actually don’t use only one emoji in per Tweet. Hence, each data(a Tweet) can actually contained several emojis, which cause the problem complicated. To make it simple, I decided to take only the three classes above as my predict variables. Afterwards, I collected the data again based on these three emoji. This time, I collected 13000 tweets for each emoji and separated each 10000 tweets as training set and 3000 as testing set. This time, I arranged my data as a different form, setting each column as a word, each row as a tweet, and there is three columns at the last of data that representing which emoji group this tweet belong.

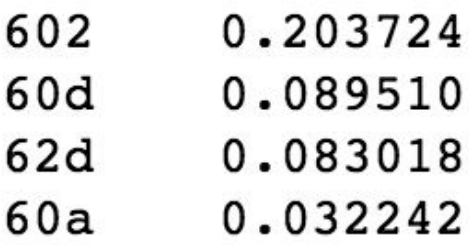

Figure3. Proportion for emoji from data

To choose the model for classification, I came with naive bayes, Xgboost, and neural network at first as my nominee. Naive bayes classifier often has great performance on the text classification and the NLP field even its concept is very simple. Besides, Naive Bayes classifier has a absolute advantage that beat most of the algorithms, which is the computational speed. The training speed of Naive Bayes Classifier was far more faster than other algorithms due to its simple step of training. Hence, I decided to take Naive Bayes classifier as my baseline of model.

Next, I came to Xgboost. Xgboost is a widely chosen modules recently in Kaggle and has helped many team to win the prize. Xgboost actually is named gradient booster in definition. It is also a evolution of decision tree. Even Xgboost also categorized as a ensemble method, however, unlike the random forest that build numerous trees “parallely” and vote together to predict, Xgboost builds trees one after another “sequentially”. The concept is that the algorithms build a shallow tree(max depth around 3~5) at first and to compute the error, then build the tree next but try to decrease the former error a little bit. Users can tune how many estimators(trees) to be built in the algorithms and this is the most important parameter in the Xgboost. The reason I choose it is that it usually works quite well on large data set and I think the concept of “modify the error sequentially” can learn the pattern of my data better than other algorithms.

To build the neural network, I chose the TensorFlow API, which is the basic structure of many high level deep learning API, such as Keras, and it is still growing rapidly now. TensorFlow also released the JavaScript version recently which is called “TensorFlow.js” that uses can run deep learning on their web browsers(link). As the deep learning theories and tools keep improving(?), there are numerous cases showed that deep learning has better performance than the traditional algorithms. The powerful point of deep learning is that it can learned more complicated, nonlinear pattern in the data. And in the NLP field, the evolution of neural network, recursive neural network, has also dominating for a long time. Hence, I decided to build a basic fully connected neural network as a preliminary practice to the deep learning and also see how well it can learn from the data.

5.2 Algorithm Implementation

First, as I have mentioned, I took Naive Bayes classifier as my base line. And I implement it with scikit-learn. The scikit-learn's normal step of training and testing helped me finish it quickly. Looking into the result of training and test accuracy, I got **50%** and **33%**. And I also looked into the confusion matrix, it has the outcome below.
(graph of confusion matrix)
From the confusion matrix, we can see that proportion of predicted wrong and true are separately averagely in three categories, which can be imagined as averagely random guess of each three categories . And the model only performed a little bit better than random guess.I came up with two possible reasons for the result of poor performance on the test set. The first is that we can perceive that there is about 23% of difference between the training and test performance, which is a signal of overfitting. However, even overfitting is a problem that people goes back to the model usually, I started to suspect that if my training data and testing data are from totally different distribution, which means that people's usage of word in Tweet is quite different. Hence, I decided to go to the neural networks and Xgboost to see if I can solve this problem.

Two hypotheses:

1. **Naive Bayes didn't capture the pattern of the data successfully.**
2. **Test data and train data are from totally different distribution(different usage of words)**

Secondly, I want to know if Xgboost can capture the underlying pattern of my data or . It has two method for user to build a Xgboost model, One is their own API, another is a way similar to model building in scikit-learn and I built the model with the latter. Same as other model in scikit-learn, one has to call the classifier function into a variable and tune the parameter, such as number of estimators or learning rate. I used the default setting first, and then fit the x, y data to train the model. After computation for about fifteen minutes, model was built. Then I fit in the train data again and the test data to see their accuracy. To my surprise, I got a "slightly better" results compared to the Naive Bayes classifier. The train accuracy was **38.47%** and the test accuracy was **36.08%** this time.

Compared to the Naive Bayes, the Xgboost model gave us a slightly more general model, that testing accuracy was a little higher and the difference between train accuracy and test accuracy wasn't that distant. However, because the train accuracy was not great enough either, we couldn't conclude that this model is more general or it just didn't capture the pattern of data successfully. To solve(?) this question(?), I started to tune the model to see if I can get better result on different set of parameters. With the guideline from AARSHAY JAIN, I tuned the model's parameters in the following order:

learning rate -> number of estimators -> max depth

There are still lots of tunable parameters, and the three I chose here were the parameters I thought would affect the model much. Learning rate mainly decide if the model will be overfitting. With a higher learning rate, such as the default value 0.1, model will update at a faster pace and cause the final outcome overfitting. But we can prevent this problem with early stopping, which means that setting the number of estimators at a comparing low number. Number of estimators means how many trees we are going to build sequentially.And this also has strong connection with the learning rate. A higher learning rate should better has few number of estimators. Besides, this also affects the computation time. More estimators will cause the computation time longer. Hence, a great balance between learning rate and number of estimators should be chosen carefully.The last one if the max depth of each tree. Because each tree is a shallow decision tree, how "deep" the shallow tree going to be can also cause the model to be overfitting or not and affect the computation time. The result below was the outcome of default model and model after tuning.
(graph of train and test accuracy of different parameters.)

Parameter Set	**Training Accuracy**	**Testing Accuracy**
defualt: Learning_rate=0.1,n_est=100,max_depth=3	38.47%	36.08%
Learning_rate = 0.05	37%	37.9%
Learning_rate = 0.01	34.6%	36.8%
Learning_rate=0.05, n_est=200	38.72%	36%
Learning_rate=0.05, n_est=300	40.2%	35%
Learning_rate=0.05, n_est=50	35.83%	37%
Number of estimators = 300	36.11%	38.02%
Lr_rate=0.05, n_est=200, Max depth=4	40%	35.8%

During the tuning process, first, we can see that when we tuned the learning rate from 0.1 to 0.05, we actually got a more general model that has better performance on test set. But if we keep tuning it to 0.01, it got worse result than the 0.05. Hence, we chose 0.05 as our learning rate. Next, I tuned to number of estimators from 100 to 300. This step took much more computation time to train the model(but we got better result on the testing set. Higher testing accuracy is possible if we keep tuning number of estimators higher but it will also take more computation time simultaneously. Thus, I selected 300 as the number of estimators in the model.) Before tuning the max depth, I also built another model that only changed the number of estimators to 300 and kept other parameters as default, and I reached a 38.02% accuracy on the test set. Normally, more estimators could have higher probability to overfitting, which means has worse performance on the testing set. However, it looks quite

“great” on the accuracy. Then when I checked the confusion matrix of this model and the model with 0.5 learning rate and 200 estimators. Reason of this was showed by the data. We can take a look at the two confusion matrices.

```
[[  77,   35, 1035],
 [  79,   47, 1167],
 [  81,   60, 1302]]

[[ 125,   99,  923],
 [ 129,   99, 1065],
 [ 156,  113, 1174]]
```

Figure 4. Confusion matrix of Xgboost
(Top: n_est = 300, Down: n_est = 200, learning rate = 0.05)

Even the model with 300 estimators has higher accuracy, this is because that it predicted lots of test data as the third type(Eye with Heart). And on the other hand, the model with 200 estimators and 0.05 learning rate even has lower testing accuracy. But it has slightly more general than the former model. Its distribution of each type didn’t concentrate at the third type that much than the former model. Hence, we will keep tuning with the latter model.

The last one is the max depth. I gave the first try to tune it from 3 to 4. On the training accuracy, it outperformed than all the others. However, its testing accuracy a little bit lower than the original model that max depth equals 3, besides, it is always to choose a simple model. Thus, we take the model with 0.05 learning rate, 200 estimators , and max depth equals 3 as our final model.

In the final model, we reached 36.11% and 38.02% on training and testing accuracy. Compared to the Naive Bayes, the model “sounds” to have better performance and more general. But when we look into the two confusion matrices below.

```
[  77,   35, 1035],
[  79,   47, 1167],
[  81,   60, 1302]])
```

Figure 5. Confusion matrix of Xgboost

we can find that Xgboost extremely concentrated on the third type and the Naive Bayes is more separated on each type. Recall the two possible problems we have above. Through the Xgboost model, I am more confident that the training set and the testing set came from

people that with totally different using method of emoji causing the algorithms to predict the data bad. Hence, we have only captured a tiny group of people's usage of emojis because out data volume is too small. Nonetheless, it too subjective to conclude only based on the result of two classifier. And I will keep testing my hypothesis with the neural network next.

Finally, I was wondering if neural network could have a better performance than the two classifiers above. To build the network, I built the basic structure with the Mofan's guideline from this website(link) and adjust a little bit with personal need. The basic hyperparameters were provided below:

Optimizer	Adam optimizer
Loss Function	Softmax Cross Entropy
Mini Batch Size	512
Number of Iterations	10001

Adam optimizer is chosen because numerous projects have showed that adam perform better than normal gradient descent optimizer with its features of learning rate decay, and the momentum. The next parameter is the loss function, this mainly based on what kind target we are going to predict. In my case, I am going to predict the appearance of three emojis. And I set that each y is mutually exclusive to each other, which means that each data only has one emoji as y even people may use several emoji in a text. In this kind of case, I applied the softmax cross entropy as my loss function. Softmax let prediction for each type of target into probability from 1~0 and sum to 1. The next two parameters are batch size and number of iterations. Compared to update the parameter after looking all the data, mini batch can help us update faster and approach the minimum in a smoother way. And I chose the number based on the size of my data. This four hyperparameters won't be changed through the tuning part.

Besides, to prevent overfitting, I also implement "dropout" through the layers. The concept is that the neural network will automatically, randomly let some neuron becomes 0 when training based the the specified probability. This makes the neural network harder to learn a "specific" feature, which may cause overfitting. This method is widely suggested as a regularization method for neural network. The parameter(dropout probability) I use for each layer is 0.9 for the first layer because we want to preserve more information at first and 0.5 for following layer. 0.5 has been proved averagely regularize the best based on Pierre Baldi and Peter Sadowski's paper.

The first structure I tried was a two layers network, which has a three nodes hidden layer and a three nodes output layer(three categories). The train accuracy and test accuracy were 67% and 33%. (short explanation). I set this as the base line of the neural network. Then we go to the tuning part. Because I have lots of input features(5587 words), I boldly set the units

number in the hidden unit to 100 first, and I reached 73%, 33% on train set and testing set accuracy. Compared to other neural network, the training accuracy is quite low. Hence, I chose to add an additional layer to see the performance. I added another hidden layer with 100 units next and run the model(this layer's dropout probability is 0.5). This time, I got 65% on the training and testing accuracy 35%. However, I discovered another problem when I was training the model, which is the training accuracy was growing but the testing accuracy only oscillate around 33 percent. And due to the random starting point in each time. I once reached 40% on the test accuracy.However, average outcome is around 35% and the pattern through training is oscillating. It's very obvious overfitting. Then I decided to make a simpler model. Besides, in deep learning, a long but few units in per layer network will usually outperform a short but more units in per units network. I changed the structure of model to 50 units for each hidden layers, and the model now is [5587(input), 50, 50, 3(output)]. This time, the final training accuracy decreased the testing accuracy still oscillate around 33%. From then on, I kept trying on different structures. Nonetheless, no matter how I change it. The result seemed only changed in the training accuracy. And I also checked the confusion matrix to see if the neural network predicted similar to Xgboost that concentrate in one type. Here is the result

Predicted	0	1	2	All
True				
0	173	205	231	609
1	152	185	213	550
2	200	200	243	643
All	525	590	687	1802

Figure 6. Confusion matrix of neural network

The prediction actually distributed averagely in three types. And I confirmed my hypothesis now, the only way I can elevate testing accuracy is getting more data because the current model learned too many noises in the training set.

6 Results

We compare the results of three models again. Here is the training and testing accuracy for each final model under different algorithms.

Algorithms	**Training Accuracy**	**Testing Accuracy**
Naive Bayes	50%	33%

Xgboost	36.11%	38.02%
Neural Network	70%	34%

It's obvious that we have better performance on training set with the neural network. With more complicated model, training accuracy is possible to keep increase, but this is meaningless it can't generalize to other data.Then we take the training accuracy into consideration together. If we only look at the accuracy, Xgboost may seems to have near result to the neural network. But when we look deeper into their confusion matrices below.

We can find that Xgboost concentrate on certain type much more than the neural network. This outcome also appeared in the model of Luda Zhao and Connie Zeng's work. Neural network looks to have more general results on the prediction even the accuracy isn't that high. Hence, if we are actually select one model to predict, I think the neural network will be a better choice. Beside, the neural network has much more space than the Naive Bayes classifier to improve.

7 Conclusion

Through the analysis, we found that it is not that easy to predict the emoji just only based the "separated" word because there are too many noise in each tweets. A simple model is hard to find the nonlinear pattern underlying the data. A more complicated model may help us get better result on the predicting part. And I came up with 4 solutions that maybe can elevate the testing accuracy:

1. **Getting more data**
2. **More complicated structure(deeper layers or more neurons)**
3. **Different neural network(such as recursive network)**
4. **Take the size of windows into consideration**

First, getting more data. This is no doubt that every researchers are looking for. More data always tells us more stories and help us learn more. However, it also brings more noises at the same time. Secondly, more complicated structure in neural network. Because I am still a beginner in deep learning, I tuned the parameter randomly without some guidelines. Maybe when I have more experience in the future, I can tune the model with a more efficient way instead of just random guessing the number of units and layers. Third, the neural network now take the word separately. However, sentence always has sequential relation. I think when we also take the sequential relation into consider can help us improve a lot. And this will come to the recursive neural network. The last one is the size of window, which is a important part in the research of short text in NLP. Like time series, word may also has more

relation that near itself. How many word near the center and where should be the center are also topics we can investigate in.

To put in a nutshell, through this project, I learned a lot from data preprocessing, data visualization, and different algorithms. Through actual implementation, I also got more insight about each algorithms and be able to my thought it with Python. Not only the programming part, work through the whole process also made me think of more tricky problems that may only existing in this kind of data. In the future projects, I am confident that I can come with a more complete workflow before and implement through it with a more efficient way from the experience this time.

8 Reference

[1] https://marcobonzanini.com/2015/03/02/mining-twitter-data-with-python-part-1/
[2] http://papers.nips.cc/paper/4878-understanding-dropout.pdf
[3] https://morvanzhou.github.io/tutorials/machine-learning/tensorflow/
[4] https://web.stanford.edu/class/cs224n/reports/2762064.pdf
[5] https://xgboost.readthedocs.io/en/latest/python/python_api.html
[6] https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
[7] https://lvdmaaten.github.io/tsne/