SMS SPAM Classification

Group Siuper XR

Introduction

This project will be focusing on the classification on spam messages in SMS. Spam classification is a classical topic in the machine learning field. The purpose of the project is to use artifical intelligence methods to build a prediction model which will help us classify emails accurately into spam and non-spam texts.

Two methods are utilised to extract features from the text data. The first way is count-based feature extraction and the second is Word2Vec. Then, three traditional machine learning models (SVM/Random Forest/XGBoost) are used to model and build on both sets of features. Subsequently, two forms of neutral network structures (LSTM/TextCNN) are applied to the embedding features obtained from Word2Vec. Eventually, we find that among all the models we attempt, the one stacking three traditional Machine Learning models has the best performance with a 0.95 F1 score on test set.

1 Preprocess

1.1 Data Desciption

The dataset we use is a public dataset available on Kaggle. The data is extracted randomly from following sources: Grumbletext Website, NUS SMS Corpus, Caroline Tag's PhD Thesis, and SMS Spam Corpus v.0.1 Big. Details. There are two columns in total: The first column labels whether the message is spam or ham, and the second column contains the actual SMS message. A total of 5572 SMS messages were analysed in the dataset.

1.1.1 Read Data

The dataset has a total of 4825 non-spam SMS messages and 747 spam ones. Below shows a few line examples of the data.

	label	text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

1.1.2 Train Test Split

By fixing a test size proportion of 0.2, we retained 3859 non-spam messages and 598 spam messages in the training subset, and 966 non-spam messages and 149 spam messages in the test subset.

1.2 Data Cleaning

Our preprocessing target is to transform the text message into what our model can interpret and then train on. We use the tokenizer built in the nltk library to extract tokens from every message.

For cleaning, we construct a set of stop words to filter out some of the common and plain words in the text messages. These are the words which are not very helpful in the task of classification. Next, we use some masks to substitute for certain parts of the text message. For instance, the numerical number is substituted by the word "number", website header substituted by "httpaddr" and dollar signs substituted by the word "dollar". These are done in line with the idea of normalizing the text content. After cleaning, we can then finally pass on the message to the tokenizer.

After we apply the data cleaning pipeline to the text, we get the following data format (extracted randomly from the train data set):

```
184                          [going, nothing, great, bye]
2171                                [wont, wat, wit, guys]
5422            [ok, k, sry, knw, number, siva, tats, askd]
4113     [stand, away, heart, ache, without, wonder, cr...
4588                       [finished, work, yet, something]
Name: clean_text, dtype: object
```

2 Feature Engineering and EDA

Two methods are used to construct features on "clean" data after tokenisation. The former is count-based feature extraction: to use a dictionary set to store words seen on the train set and transform every corpus into a one-hot vector, which has the same size of the dictionary. The latter is Word2Vec, a more advanced method where every word is given an embedding (representation).

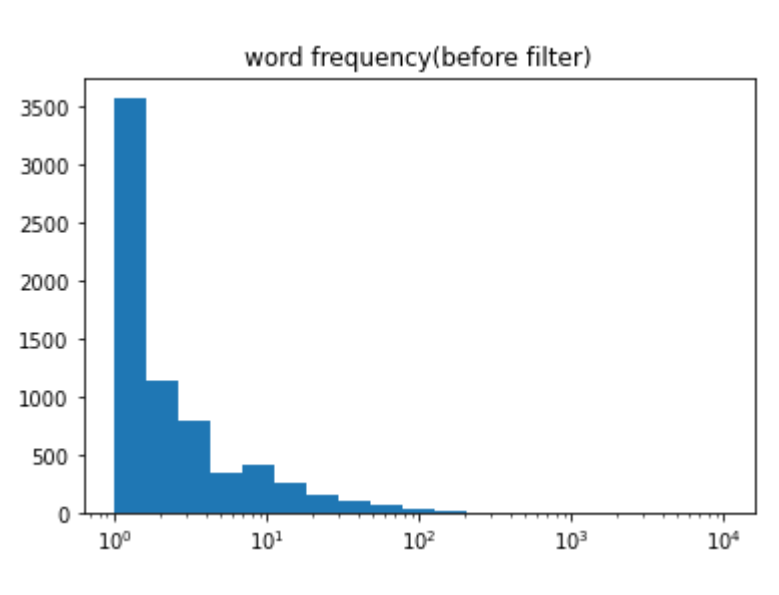
2.1 Count-based Feature Extraction

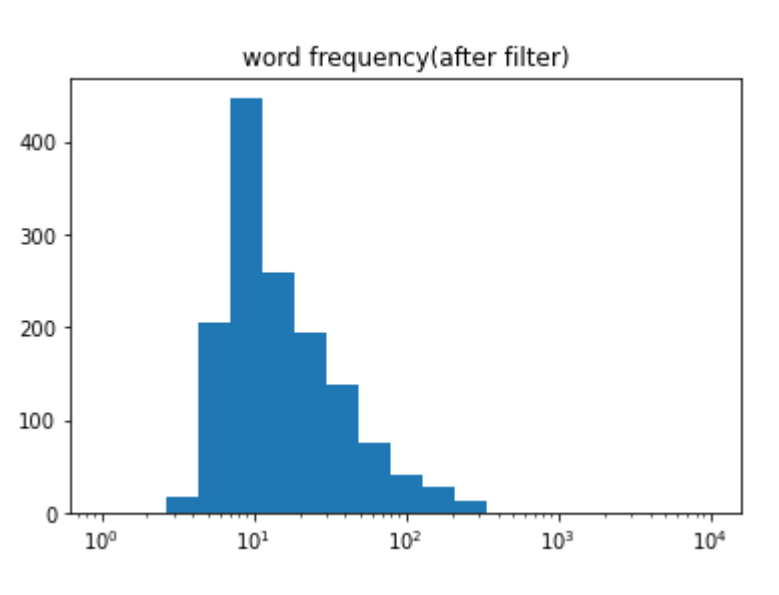
Count-based feature extraction uses the idea of constructing a one-hot vector for every message in order to count whether a word of the dictionary is present in the current message. A dictionary of words is first constructed based on the training set to fix the order of the dictionary. Every text message is iterated through the dictionary to check if it has occurred, irregardless of the frequency. If the word appeared, the position in the dictionary will be set to one, otherwise it will be left as zero.

After the processing, we would obtain a matrix of size N*D, where N represents the size of the dataset and D represents the length of the dictionary. The rows of the matrix corresponds to a one-hot vector representing each text message.

2.1.1 Visualization

In this section, we try to visualize the distribution of the occurring frequency of every word in the dataset. As shown above, the occurring frequency before filter follows an approxmiately exponential distribution, which is very common in corpus. We choose n=5 as a threshold, meaning that only words occurring more than 5 times in the training set are kept. This results in a dictionary of size 1420.





After filtering, there is a spike in word frequency distribution around 10 times. However, we note that this is not a big figure when compared to the size of the dataset (5574), which indicates that each message contains rather different set of words. This is a form of good news since count-based method will work well when spam messages contain some unique words (which could be possibly learned by moelds.

2.2 Word2Vec

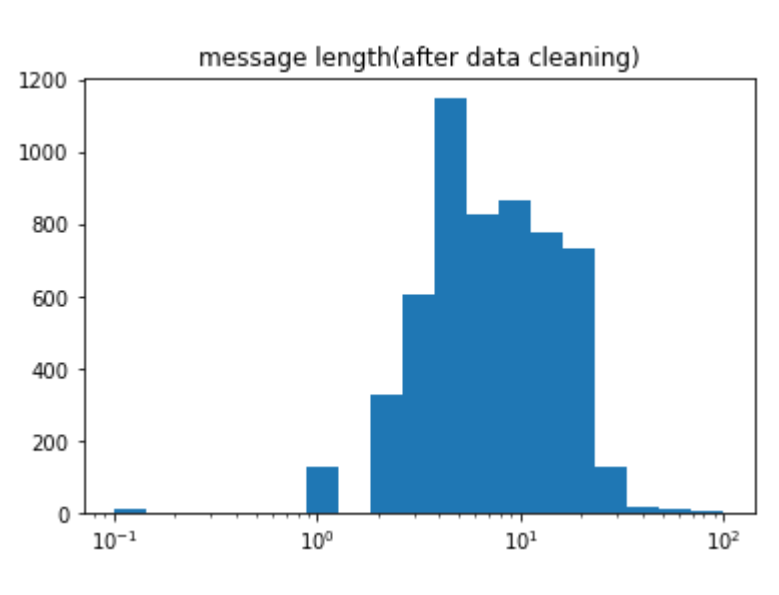
Word2Vec is another embedding method which can be used to give each word a "representation" in a high-demensional space. A main difference between Word2Vec and Glove that has been taught in class is that the latter mainly functions as a dimensionality reduction of a concurrency matrix, while the former is more "predictive" in nature. Word2Vec constructs the loss function on the probability of a word occurring around a center word (CBOW) or the probability of a word occurring in the center of some context words (skip-gram). Essentially, Word2Vec is theoretically designed to pay more attention to the order of the word and this is exactly what we would like to utilise in this section of feature engineering.

We choose 10 as the window size, and 100 as the dimension of the embedding space. Below we use TSNE to visualize the embedding of the Top200 most occurring words in the dictionary (what we get in previous section) on the training set.

The colour of the points corresponding to how many times they occur in the training set. Some data points are close to one another while others are not. This demonstrates the way Word2Vec expresses the occuring probability between two different words.

Before we feed our embedding data to our downstream model, two issues have to be tackled. Firstly, we may always encounter unseen words in the test set no matter what kind of embedding method is used. To solve this, an unseen word will be embedded as a zero-vector with the same size as the dimension chosen in the Word2Vec model.

Secondly, every message has a different number of tokens and most models are specifically designed to be trained on rectangular data. The plot below shows the distribution of message length after data cleaning steps. Most of them are below 50, so the message(sequence) max length is set to 50. We truncate every message to the threshold length, and then pad them with a zero vector of same dimension if they don't have enough words. At last, we obtain a tensor with size N*50*100, where N represents number of messages in the dataset. 50 corresponds to sequence length and 100 corresponds to the dimension of the embedding space.




```
count    4457.000000
mean        9.112856
std         6.937174
min         0.000000
1%          1.000000
50%         7.000000
99%        29.000000
max        77.000000
Name: text_len, dtype: float64
```

3 Modelling

3.1 Traditional ML Method

In this section, we will perform 3 widely-used ML methods on the two kinds of features extracted in Section 2 (Count-based and Word2Vec). They are, namely, the Support Vector Machine, Random Forest and Extreme Gradient Boosting (XGBoost).

The loss function of the Support Vector Machine comprises of two parts: L2-norm of weights and a margin indicating the tolerance in error. The target is to minimize the L2-norm weight directly, subject to the margin being small enough. This is somewhat opposite of the idea of regularized version of regression, which is to minimize the error by adjusting the respective weights. Combining with Kernel method, the SVM model could map data points to high-dimension space, which is useful for both regression and classification purposes.

Random forest is a straightforward improvement of the decision tree method. However, under the hood, the bagging is also improved in many different aspects: tree depth, number of tree nodes, minimum impurity decrease, etc., are used to fight over-fitting; sampling different features from the feature space when training individual estimators are used to promote diversification of base estimators.

Loss function of Extreme Gradient Boosting (XGBoost) also comprises of two parts, with one addressing misclassification and the other penalizing on tree structure (the latter is further divided into total number of leaves and magnitude of output values). "Boosting" comes from the fact that training is conducted in a sequential fashion. And to build a new estimator, we would take former estimators into consideration to build the current estimator. Speaking at a high level, every new tree is built to learn the "mistakes" from former trees.

3.1.1 Count-based Features

Hyperparameter settings for three different models are listed below, and the rest of hyperparameters are kept to the default setting (we don't want to tune too much in the early stage):

1. SVM
   - C = 0.1 (penalty term, the strength of the regularization is inversely proportional to C)
   - kernel = 'rbf'
2. Random Forest
   - n_estimators = 100 (number of base estimators)
3. XGBoost
   - n_estimators = 100 (number of bae estimators)
   - eval_metrics = 'logloss' (explicitly set the loss function)

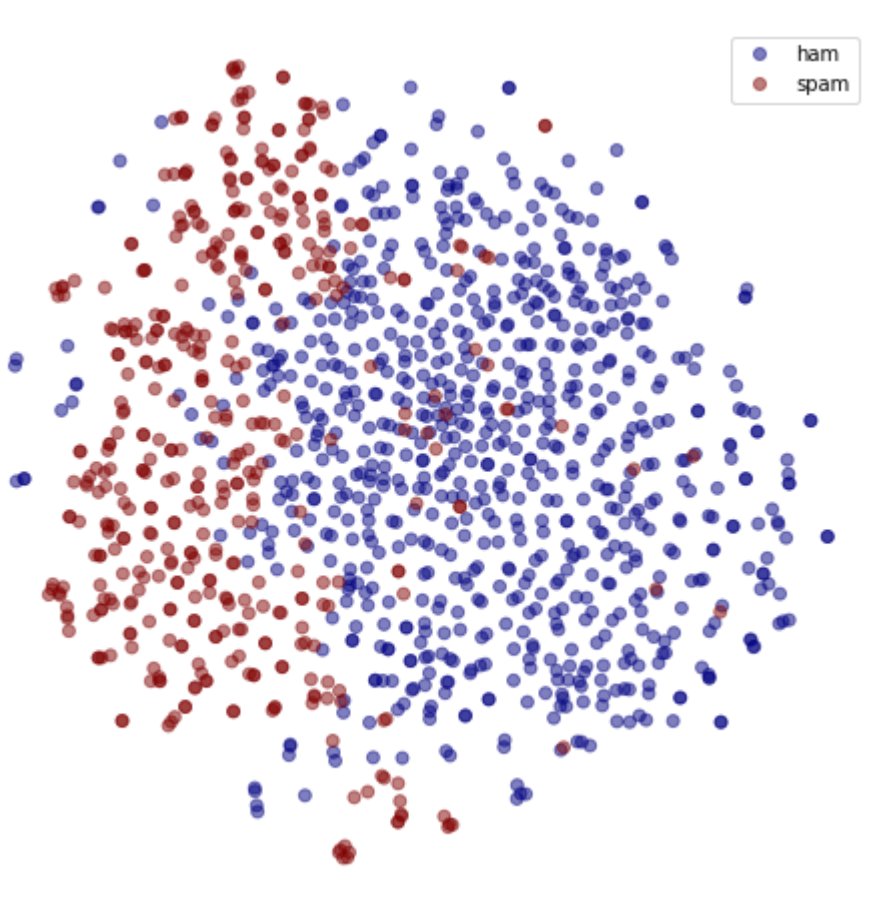
All three models showed outstanding performance on the training set. However, as the dataset used in this study is unbalanced, we have significantly more negative classes than positive classes: 4825 observations of ham and 747 observations of spam. In extreme cases, the trained model will tend towards the prediction of the major class (the class with the largest number of observations in the dataset), and as a result, show lower precision and recall in other minor classes. More generally speaking, models tend to "learn" to classify the major class well but perform badly fpor minor classes. Our target is to single out spam messages. Therefore, the precision and recall on minor class (label = 1) are much more important than the major class (label = 0), and we should put more emphasis on the test set, in particular the recall results. Our results show that SVM and random forest both attained 100% precision on the spam class with a recall around 87% for the test set.

	SVM train		SVM test		Random Forest train		Random Forest test		XGBoost train		XGBoost test	
	0	1	0	1	0	1	0	1	0	1	0	1
precision	0.9951	1.0000	0.9807	1.0000	0.9997	1.0000	0.9807	0.9924	0.9706	0.9488	0.9746	0.9612
recall	1.0000	0.9682	1.0000	0.8725	1.0000	0.9983	0.9990	0.8725	0.9933	0.8060	0.9948	0.8322
f1-score	0.9975	0.9839	0.9903	0.9319	0.9999	0.9992	0.9897	0.9286	0.9818	0.8716	0.9846	0.8921
support	3859.0000	598.0000	966.0000	149.0000	3859.0000	598.0000	966.0000	149.0000	3859.0000	598.0000	966.0000	149.0000

In addition, we identified "important words" in the dataset through the training period. For tree-based methods (random forest and XGBoost), we could easily extract the relative feature importance from built-in attributes, which calculates the importance based on the impurity decrease under the hood. For SVM, we have to fetch support vectors from the trained model (these vectors are samples which form the decision boundary). The words are then ranked according to the frequency of their appearances in these observations. As expected, the top words in each classifier are quite similar among three models. What is more interesting is that if we project the support vectors into two-dimensional space, the decision boundary it forms is pretty distinct. This indicates that even if we ignore the order or meaning of words, a count-based method to extract features is already strong enough to utilize the information contained in the messages. A probable reason is that spam classes tend to contain certain words that do not appear or appear far less often in ham classes.

	SVM	Random Forest	XGBoost
0	number	number	number
1	call	numberp	call
2	get	txt	txt
3	free	call	text
4	txt	free	www
5	text	www	sorry
6	go	claim	later
7	send	mobile	numberp
8	numberp	win	send
9	reply	service	reply

Building on the above insight, we can utilize TSNE to visualize the decision boundary of SVM in two-dimension space. More specifically speaking, we train the TSNE model using cosine similarity on the one-hot vectors of support vectors. However, there are some problems with this approach. If two one-hot vectors have totally different non-zero positions, their cosine similarity would be 0. This does not, however, necessarily mean that they are completely uncorrelated. This problem is also the biggest disadvantage of the count-based method, as they ignore the possible "meaning" of words.



3.1.2 Word2Vec Features

Word2Vec presents features in 3D dimension matrix. To make it compatible with the traditional ML methods, we will flatten the last two dimensions. After the transformation, the feature matrix shape would be N*5000.

We could see a significant decline in performance on the test set. A possible explanation is "curse of dimension": we only have 5574 observations in this dataset, and what we use for training is only 80%. As such, the number of features is even larger than the number of observations in the training set. Moreover, word embedding itself contains some information on a word level, and to simply flatten them will mean that these information will be destroyed. Hence, we see that traditional ML methods fail to learn useful information in the feature space. In the next section, we will attempt to mitigate this issue by introducing two methods in the neural network domain which are specifically designed to learn from text.

	SVM train		SVM test		Random Forest train		Random Forest test		XGBoost train		XGBoost test	
	0	1	0	1	0	1	0	1	0	1	0	1
precision	1.0	1.0	0.9590	0.9474	1.0	1.0	0.9588	0.9076	0.9767	1.0000	0.9529	0.8644
recall	1.0	1.0	0.9938	0.7248	1.0	1.0	0.9886	0.7248	1.0000	0.8462	0.9834	0.6846
f1-score	1.0	1.0	0.9761	0.8213	1.0	1.0	0.9735	0.8060	0.9882	0.9167	0.9679	0.7640
support	3859.0	598.0	966.0000	149.0000	3859.0	598.0	966.0000	149.0000	3859.0000	598.0000	966.0000	149.0000

On a side note, we found that using C = 100 results in over fitting on the train set as the penalty set is too small. However, using C = 0.01 will mostly predict 0 values due to a overly high penalty. What we can do is to split the dataset into training and validation data subsets, and look for an appropriate C value. After a few tries, we found that C = 0.1 gives the best value.

```
C=0.01
              precision    recall  f1-score   support

           0       0.87      1.00      0.93       772
           1       0.00      0.00      0.00       120

    accuracy                           0.87       892
   macro avg       0.43      0.50      0.46       892
weighted avg       0.75      0.87      0.80       892

C=0.1
              precision    recall  f1-score   support

           0       0.87      1.00      0.93       772
           1       0.00      0.00      0.00       120

    accuracy                           0.87       892
   macro avg       0.43      0.50      0.46       892
weighted avg       0.75      0.87      0.80       892

C=1
              precision    recall  f1-score   support

           0       0.95      1.00      0.97       772
           1       0.96      0.68      0.80       120

    accuracy                           0.95       892
   macro avg       0.96      0.84      0.89       892
weighted avg       0.95      0.95      0.95       892

C=10
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       772
           1       0.97      0.72      0.82       120

    accuracy                           0.96       892
   macro avg       0.96      0.86      0.90       892
weighted avg       0.96      0.96      0.96       892

C=100
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       772
           1       0.97      0.72      0.82       120

    accuracy                           0.96       892
   macro avg       0.96      0.86      0.90       892
weighted avg       0.96      0.96      0.96       892
```

3.2 Neural Network

In this section, we introduce 2 common methods used in NLP: one is LSTM and the other one is TextCNN.

LSTM is a more modern version of recurrent neural network. Here we give a brief explanation of a single LSTM block. For example, if we have a batch of size N, input dimension space D (in our current setting, D = 100), and a hidden state with dimension H, the input of every LSTM block would consist of three things: a memory cell Ct-1, a hidden state Ht-1 carried from last LSTM block, and an input Xt at every step, with shapes N*H, N*H and N*D respectively. Inside the LSTM block, we have 3 gates to control what goes in and what goes out: input gate, forget gate and output gate. It also has a dedicated cell (candidate memory cell) designed to update memory cells. They are all simple MLPs which aggregate of two parts: current input (N*D) and the last hidden state (N*H). The resultant shape of the output would be N*H. After this step, we use the output of input gate, candidate memory cell and forget gate to update the memory cell. Then, the hidden state is updated by the new memory cell and the output of the output gate. The output, hidden state and the memory cell could then be recorded for further use.

TextCNN takes ideas from convolution in computer vision. Think of the input sequence and its dimension like a picture (with shape L*D). Using kernel size of second dimension size D, after sliding across the whole sequence with defined steps and strides, this kind of 1-dimension of convolution would "squeeze" the L*D input to L*1. After this step, we apply another aggregation pooling layer, like max pooling or average pooling to transform the output into a scalar. We could redo this procedure under the same kernels (gives us more channel after convolution in the first step) or under different kernels. Thus, by using kernels with different widths (first-dimension of the kernel), we expect the model to learn the local information as well as global information (just as LSTM).

In our project, in order to apply these two methods in classification problem, the output of LSTM and TextCNN are all flattened and then fed into a fully connected layer before going through a dropout layer.

3.2.1 Word2Vec Features

Like before, we determine the hyperparameters (in both models and training period) before training the model. Here, we also add a baseline for comparison:

1. FC
   - 128(relu)->32(relu)->1(sigmoid)
2. LSTM
   - hidden_size = 64
   - num_layers = 2, bi-directional=True
   - dropout rate = 0.5
   - FC layer: 64(relu)->16(relu)->1(sigmoid)
3. TextCNN
   - output_channels: all set to 16
   - kernel_size: [2, 5, 10, 20, 30, 50]
   - pooling layer: average pooling
   - dropout rate = 0.5
   - FC layer: 32(relu)->16(relu)->1(sigmoid)
4. Training Period
   - batch_size = 64
   - num_epochs = 200
   - optimizer = 'Adam'
   - learning_rate = 0.005 / 0.001
   - metric for evaluating validation set: f1-score
   - early stopping = 10 / 20

The baseline fully connected model attained 90% precision and 82% recall for spam class on test set, and TextCNN improves the recall to around 88%. However, LSTM fails to beat even the baseline! This result is a little disappointing as we expect neural networks to be able to learn more information from datasets than traditional methods.

The reason behind this can be summarized into two. One, there are several limitations in training our current neural networks: firstly, the size of dataset, the dataset contains only 5574 samples while the number of trainable parameters in two neural networks are both around 200K, which exceed a usually optimal ratio(1:10). Secondly, the embedding we use is also trained on a relatively small amount of corpus which may lead to overfitting in certain situations. Besides, the total unseen words in test set is about 873 (not a small number comparing to the dictionary size we get in section 2.1), which further affects our model performance on test set. Lastly, we did not spend much time finetuning hyperparameters, thus limiting the power of neural network.

Two, there may exist some misalignment between the objective of embedding models and the supervised neural network. More specifically speaking, Word2Vec is a kind of unsupervised learning method. The method is designed for representative learning in certain corpus, so the loss function used to evaluate how good the model is certainly does not ensure the trained model gives the best embedding for our downstream task. A better choice is to add an embedding layer before we feed the data into the supervised neural network, but in our current setting, the result is also not very satisfying. The main reason behind this, we guess, is that there are quite a few unseen words in the test set.

	FC train		FC test		LSTM train		LSTM test		TextCNN train		TextCNN test	
	0	1	0	1	0	1	0	1	0	1	0	1
precision	0.9918	0.9861	0.9695	0.9084	0.9661	0.9206	0.9686	0.9219	0.9925	0.9930	0.9737	0.9685
recall	0.9979	0.9465	0.9876	0.7987	0.9896	0.7759	0.9896	0.7919	0.9990	0.9515	0.9959	0.8255
f1-score	0.9948	0.9659	0.9785	0.8500	0.9777	0.8421	0.9790	0.8520	0.9957	0.9718	0.9846	0.8913
support	3859.0000	598.0000	966.0000	149.0000	3859.0000	598.0000	966.0000	149.0000	3859.0000	598.0000	966.0000	149.0000

3.3 Further Improvement

Ensemble (E2E optimization) is an effective method to achieve a final boost on the performance of models. Here we conduct the ensemble in two ways, which are also two main areas of ensemble: stacking and bagging.

Stacking could be viewed as another layer after we have constructed a few models. Stacking layer, which itself is a ML model, takes in the predictions of these models as features and gives out a single prediction. The whole training process could then be conducted on this two-stage model, and a usual choice in the training period is a "K-fold training" just like the cross-validation to decide the parameters in the model of stacking layer.

Bagging could be viewed as a method of aggregation. A very simple way to preform bagging is to do an arithmetic average of the predictions of the models, which is the default way how random forest aggregates the prediction of base estimators, while the aggregation could also be done in more complicated ways like using a validation set to find optimal weights or an optimal threshold for classifying (especially in multi-class classification problems). Furthermore, we can also choose a base estimator type to train a bagging classifier using different subsets of features every time for a single base estimator, and aggregate the prediction using different methods in the output layer. This kind of idea is also adopted in random forest classifiers already.

Here, in our group project, we try to conduct these two methods on models we build in the previous section. For the stacking method, we train a logistic regression on top of those models, while for the bagging method, we use simple average aggregation.

3.3.1 Stacking

We try stacking method on the two sets of features we have by building a logistic regression on top of 3 traditional ML models using count-based features and another on top of 3 traditional ML models + 2 neural networks.

Stacking classifier 1 improved the recall on spam class in test to 92%, while stacking classifier 2 doesn't improve much compared to the strongest model (TextCNN). Empirically, the success of a stacking classifier depends on whether individual estimator is strong enough (high precision and recall). This conclusion could be validated by the following table which shows the coefficients of the final logistic regression model in both classifiers. The first classifier has a relatively balanced weight on random forest and SVM, while the second classifier is dominated by the TextCNN model.

	stackingclassifier1 train		stackingclassifier1 test		stackingclassifier2 train		stackingclassifier2 test	
	0	1	0	1	0	1	0	1
precision	0.9982	1.0000	0.9877	0.9928	0.9928	0.9930	0.9737	0.9609
recall	1.0000	0.9883	0.9990	0.9195	0.9990	0.9532	0.9948	0.8255
f1-score	0.9991	0.9941	0.9933	0.9547	0.9959	0.9727	0.9841	0.8881
support	3859.0000	598.0000	966.0000	149.0000	3859.0000	598.0000	966.0000	149.0000

	stackingclassifier1	stackingclassifier2
rf	5.220093	-0.219571
svm	2.820567	1.550972
xgb	0.986937	0.978804
textcnn	NaN	6.432822
lstm	NaN	0.759506

3.3.2 Bagging

The bagging method is also done in two ways: average over SVM and random forest (two strong estimators in count-based features) or average over the former mentioned two and the TextCNN model trained on features extracted by Word2Vec.

The results below show that simply bagging two traditional MLmodels trained on count-based features gives a comparable result to the stacking classifier 1 in the previous section. Taking the TextCNN model into account does not seem to do much in improving the result.

	bagging classifier1 train		bagging classifier1 test		bagging classifier1 train		bagging classifier1 test	
	0	1	0	1	0	1	0	1
precision	0.9997	1.0000	0.9867	0.9927	0.9977	1.0000	0.9797	1.0000
recall	1.0000	0.9983	0.9990	0.9128	1.0000	0.9849	1.0000	0.8658
f1-score	0.9999	0.9992	0.9928	0.9510	0.9988	0.9924	0.9898	0.9281
support	3859.0000	598.0000	966.0000	149.0000	3859.0000	598.0000	966.0000	149.0000

4 Summary

In this SMS messages dataset, traditional machine learning models, such as SVM and random forest classifier, appear to perform well on test sets after training on features that are extracted in a straightforward fashion: fixed-size dictionary building and counting occurring times. Common deep learning model structures in NLP like TextCNN achieved comparable results to the aforementioned machine learning models, but LSTM on the other hand, did not produce a good result.

In practice, while traditional ML method is a good start to classify spam messages, we strongly recommend a deeper look at how neural networks perform under different sets of hyperparameters and modified structures (like the End2End structure we mentioned), provided that there are more data and computing resources.