LING 131-A Fall Semester
Marc Verhagen Wednesday, December 19, 2018

Text Message Spam Detector

By Mingfan Chang, Sarah Hotung, John Laurentiev, and Meiqi Wang

I. General Project Aims

The goal of our SMS (i.e. text message) spam detector is to determine whether a given text message is a spam message or a genuine ("ham") message. Similar to email spam, SMS spam is often used to deceive someone into giving away their money or personal information (social security number, bank account/login information, etc.). Spam filtering software is common across email platforms and is used to prevent such emails from reaching a user's inbox. SMS spam is an issue that has begun to proliferate with the increase in messaging application user bases. Our program is designed to, given a text message, determine if that message is spam, which could in turn be used by spam filtering software to block such messages or send them to a sort of mobile spam folder.

Examples:

Spam	Thanks 4 your continued support Your question this week will enter u in2 our draw 4 Â£100 cash. Name the NEW US President? txt ans to 80082	Your unique user ID is 1172. For removal send STOP to 87239 customer services 08708034412
Ham	Me too. Mark is taking forever to pick up my prescription and the pain is coming back.	CHEERS LOU! YEAH WAS A GOODNITE SHAME U NEVA CAME! C YA GAILxx

Our aim in this project is to use the NLTK package's machine learning functionality to determine which features are most relevant for detecting SMS spam, as well as to see which classifier model works best to achieve this goal.

II. Finding and Creating Labeled Data

In order to create an SMS spam detector, we need to find a corpus of text messages that had already been labeled as “spam” or “ham” messages, which we can use to train our classifier so that it will be capable of labeling unlabeled text message data as accurately as possible. For this purpose, we have found a corpus of SMS messages that had been tagged by researchers at the University of California at Irvine (UCI) called the SMS Spam Collection[1]. This corpus consists of 5,574 messages that have been tagged for research purposes as either “ham,” or legitimate messages, or “spam.” In order to build this corpus, researchers were able to extract 425 spam messages manually from a UK website specially designed for users to report spam text messages, whereas the main chunk of 3,375 ham messages were selected at random from the National University of Singapore (NUS) SMS Corpus (NSC), a 10,000-message corpus of legitimate messages collected by volunteers at NUS. Additionally, 450 more ham messages were added from Caroline Tagg's PhD Thesis at the University of Birmingham, and the remaining 1002 spam messages and 322 ham messages were collected by incorporating the SMS Spam Corpus v.0.1 Big into the final data set.[2]

Since we know that we will ultimately need some amount of text message data on which to run our classifier on once we have built it up so as to evaluate and analyze its accuracy, we need to gather a secondary corpus of text messages to act as our test data set. After scouring the web, we have decided that it makes the most sense to just use a portion of the data set we already have as our test data set, since it is already nicely compiled and will give us the chance to see how our classifier does when we give it similar data to the training data in terms of its ability to label the new messages as spam or ham. We have therefore decided to set aside 500 messages from our labeled data set (which we will call ‘development_data.cvs’) and save them in a separate file called ‘test_data.csv.’

Once we download our development set of labeled spam and ham messages, we need to ensure that we upload it into our code in the correct format such that we are able to extract a list of tuples consisting of the label ‘ham’ or ‘spam’ (as a string), followed by the given message (also represented as a string). We will need to process our data a bit before we feed it into our load_labeled_data() function, which we will discuss in the next section, but before we do we need to make sure our data file can be found within our file folder named SMS. This has required some trouble shooting, since we’ve found that while joining the file name with ‘\\’ and the directory works for PCs, it does not work for Macs, and while a single ‘/’ does seem to fix the problem temporarily, we wanted to make sure our file would be able to be found by all operating systems. We have therefore decided to use the os.path.join() function along with a directory variable of os.getcwd() to ensure that our development_set.csv file will be locatable within our

[1] UCI. “SMS Spam Collection Dataset.” *RSNA Pneumonia Detection Challenge | Kaggle*, 2 Dec. 2016, www.kaggle.com/uciml/sms-spam-collection-dataset/home.
[2] www.kaggle.com/uciml/sms-spam-collection-dataset/home.

SMS folder on any operating system, and can be loaded by our load_labeled_data() function so that we can use this labeled data to create feature sets and properly run our classifier.

III. Processing the Data

Before actually loading the data, we needed to take some steps to clean and pre-process our corpus to ensure that the data we were working with was in an adequate state for feature extraction. We did this by removing all stopwords from the message data so that we would only be working with content words in our data set. This way, when it comes time to extract features, the messages will only consist of actual content words and phrases that we might be looking for, and all common words like “the” or “and” will have been removed. We also stemmed the message data using the Porter stemmer, as it is a widely-used stemmer that is not as aggressive as most others. This allowed us to turn words like “winner,” “winning,” and “winnings” to the more common stem “win,” so that we only have to search for the stem of the word in our features, not all possible morphological disambiguations. This removed a few extra steps for us later on when it came to defining the features, since all the words in the messages were already in their simplest forms.

IV. Creating Features

Below are the features that we have ultimately decided to test out using our three classifiers (which we will discuss more in the section to come):

find_website: detects whether the message contains any webpage links. We found out that webpage links appear with a really high frequency in spam messages such as: *Santa calling! Would your little ones like a call from Santa Xmas Eve? Call 09077818151 to book you time. Calls1.50ppm last 3mins 30s T&C www.santacalling.com.* When extracting this feature, we first used regular expression trying to extract the whole webpage lin, e.g. ’\A(http://)?(www)?.*\Z. However, by testing the show_most_informative_features, this feature seems inefficient. By our intuition, the result should not be like this. After several attempts, we have found out that extracting "www" could also detect the webpage link appeared in messages and we get a good performance with this feature.

has_slang: detects whether the message contains any words that could be categorized as “text slang,” specifically “lol,” “lmao,” “wtf,” “bff,” “omg,” or “rofl.” We have found these by leafing through our list of messages by hand and checking for any similarities between ham messages that didn’t seem to be showing up in spam messages, and these specific slang words all appeared throughout the ham messages but never appeared in spam. We have added the has_slang feature by using re.search to find the regular expression ‘(lol|lmao|wtf|bff|omg|rofl)’ in the given message.

has_emoticon: detects whether or not an emoticon is present in the message. This is another feature that seems to be a good indication of a ham message--the presence of emoticons, or combinations of punctuations made to look like faces that spammers tend not to send (i.e. ‘:-)’, ‘:P’, : ^)’, etc.). Coming up with a regular expression to capture the exact types of emoticons we expected to appear in ham messages is a bit tricky, since in theory, there are infinite types of emoticons one can create with a keyboard with a bit of imagination, but we have decided to narrow it down to the most straightforward types--horizontal faces (like the ones above) made up up some sort of eyes, nose, and mouth, so that the regular expression we are searching for is '[:;]\'?\s?(-?|*?|\^)?\s?[\)\(DPp3O0o]'.

is_spam_call: detects whether the message contains a word of “call” or “TXT” followed by a string of numbers which could represent a phone number. We realized the high frequency of the pattern: *call 12345465* appeared in spam messages. Then we used regular expression to extract this feature such as: *txt|text|TXT|TEXT|call|CALL) ([A-Z]{,10}|[0-9]+*

length_of_message: assigns a message a bucket value of either “long”, “medium” or “short” based on its number of characters. We saw that the maximum spam message length in the training set was 519 characters, while the maximum for ham was 173 characters, suggesting that very long messages tend to be ham, and motivating our cut-off choice of 200 between medium and long message lengths. Looking at average lengths, ham messages were around 117 characters, while spam averaged to be 56 characters long, suggesting that very short messages were also more likely to be spam. We took the midway point between these averages (86) as the cutoff point between short and medium length messages. We speculated that messages in the “medium” category were more likely to be ham, while “long” or “short” messages were more likely to be spam.

contains_gibberish: detects whether the message contains a string that contains at least one letter and one number. Many spam messages seem to contain such strings of “gibberish”. We chose to exclude strings that begin with a number such as “1hr” or “5mins” since these are terms commonly used by senders of ham texts. Returns True if such a string is found, and False if not.

find_win/award/urgent/prize/free/claim: detects whether the message contains with “win”,”award”,”urgent”,”prize”,”free”,”claim”. When we went over the dataset, we realized the high frequency of these words appear in spam messages. We used regular expressions to extract each of these feature.

capitalization: detects whether a message contains a fully capitalized word (e.g. "FREE"). Spam messages seem to have a tendency of writing words like "WIN" or "PRIZE" to catch the receiver's attention, and so may be an indicator that a message is more likely to be spam.
contains_name: detects whether a text message contains one of the person names in the set of names in the Names.txt file. Names tend to be used regularly by senders of ham messages, unlike in most spam messages. Returns True if a match is found, and False if not. (Initially tried to achieve name detection using POS tags; however, many slang terms beginning with a capital letter [such as "Lol" or "WTF"] are tagged as proper nouns by NLTK's POS tagger, making such a method unreliable.)

contains_common_ham_bigram: compares the bigrams of a given SMS message against the most common bigrams among all ham messages in the training set. Returns True if a match is found, and False if not.

contains_common_spam_bigram: compares the bigrams of a given SMS message against the most common bigrams among all spam messages in the training set. Returns True if a match is found, and False if not.

The show_most_informative_features() function gives us some feedback about the accuracy of the features we have chosen. Here's what the most informative features are for the Naive Bayes and the Max Entropy Classifiers show before we go through and test all the permutations of features for actual accuracy (the Decision Tree Classifier does not have a built in show_most_informative_features() function):

Naive Bayes:

```
    find_award = True                 spam : ham   =   225.3 : 1.0
    find_website = True               spam : ham   =   210.9 : 1.0
    contains_gibberish = True         spam : ham   =    22.3 : 1.0
    is_spam_call = True               spam : ham   =     9.7 : 1.0
    find_win = True                   spam : ham   =     5.9 : 1.0
    length_of_message = 'short'       ham : spam   =     5.8 : 1.0
    length_of_message = 'medium'      spam : ham   =     5.5 : 1.0
    is_spam_call = False              ham : spam   =     3.8 : 1.0
    has_emoticon = True               ham : spam   =     2.8 : 1.0
    contains_gibberish = False        ham : spam   =     1.7 : 1.0
```

Maximum Entropy:

```
    -5.459 find_website==True and label is 'ham'
    -4.171 find_award==True and label is 'ham'
```

```
-2.070 length_of_message=='short' and label is 'spam'
-1.861 is_spam_call==False and label is 'spam'
 1.681 find_claim==True and label is 'spam'
-1.524 contains_gibberish==True and label is 'ham'
 1.180 find_prize==True and label is 'spam'
 1.089 length_of_message=='long' and label is 'ham'
-1.046 is_spam_call==True and label is 'ham'
-0.960 has_emoticon==True and label is 'spam'
```

V. Choosing and Running Classifiers on Training Set

When picking which classifiers to run on our data, we want to use three different classifiers to add depth and accuracy to our spam detector. The three classifiers that we have found to be the most effective for this purpose are the Naive Bayes Classifier, the Decision Tree Classifier, and the Maximum Entropy Classifier, all of which are described in the NLTK book[3] and which are available through NLTK's packages. The Naive Bayes Classifier considers the effect of each feature in combination with the prior probability of each possible label, which is determined by checking the frequencies of said labels in the training set. It assumes that all features are independent of each other (thus "naive", since features often are dependent on each other to some degree) in choosing the most likely label for its input. NLTK's NaiveBayesClassifier takes a training set input to be trained on and returns a classifier that can be used to classify strings as spam or ham SMS messages.

The Decision Tree Classifier builds a tree with branching decision nodes. At each level of the tree, the classifier checks a particular feature and, based on its value, follows a branch to the next decision node. After following each decision node down the tree, eventually a leaf node is reached that provides a label for the input. Decision trees are easy to understand and interpret, making them an attractive choice of classifier. Some downsides of such classifiers are that the set of data they are trained on becomes smaller the further down the tree the classifier gets, and they require the checking of features to occur in a specific order.

The Maximum Entropy classifier model is a generalization of the model used by the naive Bayes classifier. But rather than using probabilities to set the model's parameters, it uses search techniques to find a set of parameters that will maximize the performance of the classifier. In particular, it looks for the set of parameters that maximizes the total likelihood of the training corpus. The Maximum Entropy classifier model leaves it up to us to decide what combinations of labels and features should receive their own parameters. The intuition that motivates this

[3] Bird, Steven, et al. "Natural Language Processing with Python– Analyzing Text with the Natural Language Toolkit." *NLTK Book*, www.nltk.org/book/.

classifier is that we should build a model that captures the frequencies of individual joint-features, without making any unwarranted assumptions.

VI. Evaluating and Running the Classifiers

Now that we have chosen our classifiers, we need to figure out which features or sets of features provide us with the most accurate results for each of our chosen classifiers. Here, we are not only concerned with accuracy in the sense of the classify.accuracy function that NLTK provides, but also precision, recall, and f-measure, which take a look at the relationship between true positives, false positives, and false negatives, and can give us a much better idea of how our sms spammer is doing in terms of classifying spam as spam and ham as ham. We have attached all of our observations on each classifier's respective accuracy, precision, recall, and combined f-measure for various combinations of features in the Appendix

For the naive Bayes classifier, accuracy and recall seem to be proportional to the number of features included. On the other hand, precision appears to be optimal when using only some of the features that are most informative and omitting the rest. For example, using just the length_of_message, contains_gibberish and find_website features, recall is a mere 46%, whereas precision comes to 84%. As the less impactful features are added to the feature set (e.g. capitalization and is_spam_call), we see a slight drop in precision but a notable increase in recall, resulting in an improved f-measure for the model. Accuracy maxes out at around 95%, while precision and recall each max out at about 84% and 81%, respectively. When computing the f-measure for each combination examined, including all features seems to be the best bet, with a resultant score of 80%.

For the decision tree classifier, it can be seen that the combination of different features result in different results. For example, the precision of the website feature is higher than its recall, the reason being that perhaps some of the spam messages don't have websites in them. Another example is that the accuracy of feature combination of "Length, Gibberish, website & keywords" is 92.69%, while when another feature "capitalization" is added to it, the accuracy didn't change, thus the feature might not contribute much to the overall results. However, in another data analysis, when we have the combination" of "word length & gibberish" first, the accuracy is 90.12%, while when key words feature is added to the combination, the accuracy becomes 91.50%, and when all features expect the bigram feature are added together, the accuracy become highest: 94.86%. The precision across features is similar to each other while the recall varies across different features; it is highest with the feature combination of all features except bigrams and with feature combination of "Length, gibberish, website, slang, emoticon & spam call".

When it comes to the maximum entropy classifier, our results seem to be directly related to the list of most informative features, which has proven to be extremely integral when picking sets of features to test accuracy, precision, recall, and f-measure on this particular classifier. As with the other classifiers, we are beginning our analysis by testing out all the features, so as to have an idea of what our jumping off point should be. These results are actually fairly high to begin with: 95% accuracy, and 85% across the board for precision, recall, and f-measure. With a score of 85% for f-measure, this is already the highest score we've achieved out of all three classifiers, which might be reason enough to accept this batch of features with the Max Entropy classifier as the ultimate route we want to take. It's always worth it to delve a bit deeper into the analysis, however, and see if we can't bring those scores up just a tad by playing around with the feature sets on this classifier as compared to the other two.

Before taking a look at the informative features list for Max Ent, we want to test out some sets of features based on our intuitions to see how they'll do, then turn to the list for inspiration. As a preliminary test, we'll run Max Ent on a few simplistic sets--has_slang and has_emoticon only, and then contains_name only--to see what happens. This can ensure that what we would expect to happen happens, i.e. the classifier does not do a very good job, since it only has one or two features to work with, all of which identify only ham, not spam. Running these feature sets confirms our suspicions--not only is the accuracy much lower, but the precision and recall for both of these is 0%, since only the ham messages are getting picked up. We can therefore expand our search to include more of a variety, such as the find_word features for "claim," "urgent," etc., is_spam_call, and contains_gibberish (all good for finding spam), mixed in with other features like contains_name, capitalization, and has_slang. Looking at the results, however, we can see that we're still getting a bit of a mixed bag--despite the fact that the accuracy looks reasonable, we can see that the precision, recall, and combined f-measures are still not where we'd like them to be. In order to maximize these scores, which is what we want to ensure that the most possible spam messages get labeled as spam and ham messages get labeled as ham, let's turn to the list of most informative features for inspiration.

We can start out by testing only a few of the features that are towards the top of the list, and also aim for features that show up on the list for the Naive Bayes Classifier too. When we run Max Ent on the set of find_award, find_website, and is_spam_call, we can see that the recall is way higher than with any of our previously tested combinations, but the precision is still not great, yielding a f-measure of 64.9%, which we can definitely improve upon. If we take all of the features from the list of most informative for Max Ent except for the two extra word ones, "prize" and "claim," we get the highest results we've seen so far except for the initial ones--accuracy jumps up to 94.6%, precision is 81.54%, recall is 88.33%, and f-measure is 84.80%. We can try one last combination of the same features plus the two extra word ones, find_claim and find_prize, but this only brings all the results back down and gives us an f-measure of 81.74%, which is still lower than what we want. Finally, let's try one last run on all the features as a group to see what we get. The results are identical to the set of features without the most common ham and spam bigrams, which means they are the most fruitful results to date--95% accuracy and 85% precision, recall, and f-measure. We can therefore determine that, after all of our analyses of the various combinations of sets of features to use with the Max Ent classifier, our original instincts when coming up with the features were actually on the right track, and moving forward, we want to continue using all of our predefined features to run this classifier on the data.

Now that we've decided which features are the most useful in terms of blocking spam with each of the three classifiers, we can run our code on our test data and take a look at the results we get back. When we run the three classifiers, we want to print out the accuracy of each as compared to the labels that are already assigned to that data. For example, when we run the Naive Bayes classifier on the test set using all of our features after shuffling the data, we get an accuracy rate of 96.39%, which is exciting to see! Checking the accuracy on the test set is the final step in our process before we look forward to future ideas for improvement and expansion.

VII. Conclusions and Ideas for Expansion

Given more time, there are ways we could further refine and expand upon our SMS spam detector. Some of the data contained strings of symbols (e.g. Â£ and &lt;#&gt;) that we were hesitant to remove in the event that they proved useful in detecting spam messages. Figuring out why such strings appear, and testing the model filtering out such strings in pre-processing of the data could prove beneficial. For features, we created some that searched for the presence of certain words, such as "free", "win" or "prize", since such words seemed to appear quite often in spam messages. To expand upon this, one could utilize Wordnet to capture related synonyms of these words and include those terms as part of this word search feature. Performing error analysis would be a method to try to improve our model - looking at test data that had a mismatch between the actual label and the label given by the classifier(s) to try to determine why they were incorrectly labeled. We could also invest more time in coming up with new features for our feature set.

Appendix

Naive Bayes	Accuracy	Precision	Recall	F-Measure
All features except bigrams	94.59%	79.41%	80.60%	80.00%
Only word length feature	86.77%	--	0%	--
Word length & gibberish	90.98%	81.81%	40.30%	54.00%
Word length, gibberish, slang & emoticons	90.98%	81.81%	40.30%	54.00%
Word length, gibberish & key words (win, free, etc.)	93.19%	82.00%	61.19%	70.08%
'www' only	88.58%	100%	13.43%	23.68%
Length, gibberish & 'www'	91.78%	83.78%	46.27%	59.62%
Length, gibberish, website & contains name	91.78%	83.78%	46.27%	59.62%
Only gibberish	89.18%	64.29%	40.30%	49.54%
Length, gibberish, website & key words	93.39%	82.35%	62.69%	71.19%
Length, gibberish, website, key words & capitalization	93.39%	82.35%	62.69%	71.19%
Length, gibberish, website, slang, emoticon & spam call	94.19%	78.79%	77.61%	78.20%
All features	94.59%	79.41%	80.60%	80.00%

Decision Tree	Accuracy	Precision	Recall	F-Measure
All features except bigrams:	94.86%	85.45%	78.33%	81.74%
Only word length feature:	85.57%	--	--	--
Word length & gibberish:	90.12%	87.50%	46.67%	60.87%
Word length, gibberish, slang & emoticons	90.12%	87.50%	46.67%	60.87%
Word length, gibberish & key words (win, free, etc.)	91.50%	90%	60.00%	72.00%
website only	87.94%	87.50%	11.67%	20.59%
Word length, gibberish, website	91.50%	87.50%	58.33%	70.00%
Length, gibberish, website & contains name	91.50%	87.50%	58.33%	70.00%
Only gibberish	90.12%	88.24%	50.00%	63.83%
Length, gibberish, website & key words	92.69%	89.13%	68.33%	77.36%

Length, gibberish, website, key words & capitalization	92.69%	89.13%	68.33%	77.36%
Length, gibberish, website, slang, emoticon & spam call	94.67%	85.19%	76.67%	80.71%
Word length, gibberish, website & emoticon	91.50%	87.50%	58.33%	70.00%
gibberish, website	91.50%	88.10%	61.67%	72.55%
gibberish, emoticon	90.12%	88.24%	50.00%	63.83%
Word length, website	87.95%	87.50%	11.67%	20.59%
Word length, emoticon	85.57%	--	--	--
Website, emoticon	87.95%	87.50%	11.67%	20.59%

Maximum Entropy	Accuracy	Precision	Recall	F-Measure
All features except bigrams	95.00%	85.00%	85.00%	85.00%
Has slang & has emoticon	86.30%	0.00%	0.00%	--%
Contains name only	86.20%	0.00%	0.00%	--%
Find claim, find prize, find urgent, find free	88.90%	100.00%	25.00%	40.00%
Find claim, find prize, find urgent, find free, find win, find award	89.70%	100.00%	26.67%	42.11%
Capitalization, contains name, gibberish, find website, length, has slang	91.70%	87.50%	58.33%	70.00%
Contains name, has slang, find free, find website	87.90%	87.50%	11.67%	20.59%
Is spam call, gibberish, contains name, has slang	90.90%	100.00%	40.00%	57.14%
Find award, find website, gibberish	91.70%	88.37%	63.33%	73.79%
Find award, find website, is spam call	90.60%	53.85%	81.67%	64.90%
Find award, find website, length, is spam call, gibberish, has emoticon	94.60%	81.54%	88.33%	84.80%
Find award, find website, length, is spam call, gibberish, has emoticon, find claim, find prize	95.00%	85.45%	78.33%	81.74%
All features	95.00%	85.00%	85.00%	85.00%

References

1. Bird, Steven, et al. “Natural Language Processing with Python– Analyzing Text with the Natural Language Toolkit.” *NLTK Book*, www.nltk.org/book/.

2. UCI. “SMS Spam Collection Dataset.” *RSNA Pneumonia Detection Challenge | Kaggle*, 2 Dec. 2016, www.kaggle.com/uciml/sms-spam-collection-dataset/home.