Springboard

SMS Spam Detection

Karolina Bzdusek

May 10th, 2019

Springboard

Contents

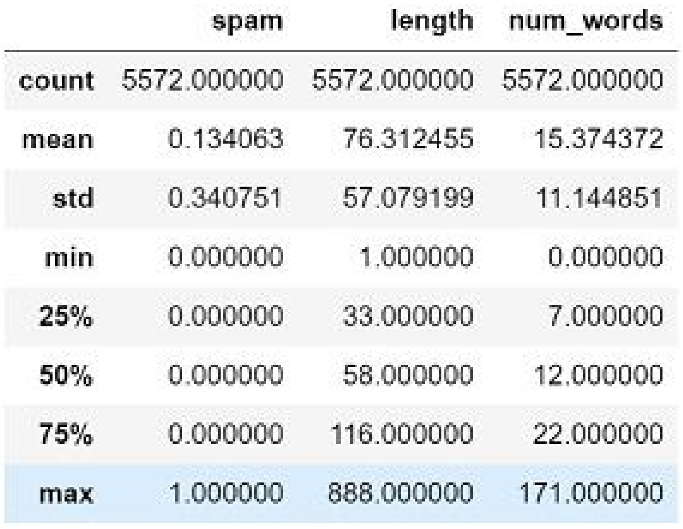
1. Introduction

Everybody has an experience of unwanted messages or emails, or on the other some "lost" messages, meaning they ended up in spam folder. Both of these scenarios are undesirable and therefore building spam filter of high quality is in high demand.
The aim of this project is to determine whether SMS is spam or not, explore various classifiers and mainly to explore basic techniques from Natural Language Processing (NLP).
The SMS Spam Collection v.1 is a public set of SMS labeled messages that have been collected for mobile phone spam research. It has one collection composed by 5,574 English, real and non-enconded messages, tagged according being legitimate (ham) or spam.

Source of data set.
Reference: http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/
The table below lists the provided dataset in different file formats, the amount of samples in each class and the total number of samples.

Application	File format	# Spam	# Ham	Total	Link
General	Plain text	747	4,827	5,574	Link 1
Weka	ARFF	747	4,827	5,574	Link 2

In the table below we can see statistics regarding our dataset. Where spam can have two values (0,1) → (ham, spam), length is 'length' of characters of each message, 'num_words' is number of words of each message.



	spam	length	num_words
count	5572.000000	5572.000000	5572.000000
mean	0.134063	76.312455	15.374372
std	0.340751	57.079199	11.144851
min	0.000000	1.000000	0.000000
25%	0.000000	33.000000	7.000000
50%	0.000000	58.000000	12.000000
75%	0.000000	116.000000	22.000000
max	1.000000	888.000000	171.000000

As it is a NLP task, the dimensionality is high (each unique word is a feature), therefore some cleaning of the text is needed (e.g. punctuation is possibly not needed, or if the message contains two and more exclamations marks in a row, it makes difference whether it's four or ten; lower and upper case of the word can on the other hand reveal some insight whether it's a spam or not, etc). This is one way how to reduce dimensionality before transforming words to vectors. Another would be to determine features that are the most helpful to recognize spam from ham, in other words dimension reduction.

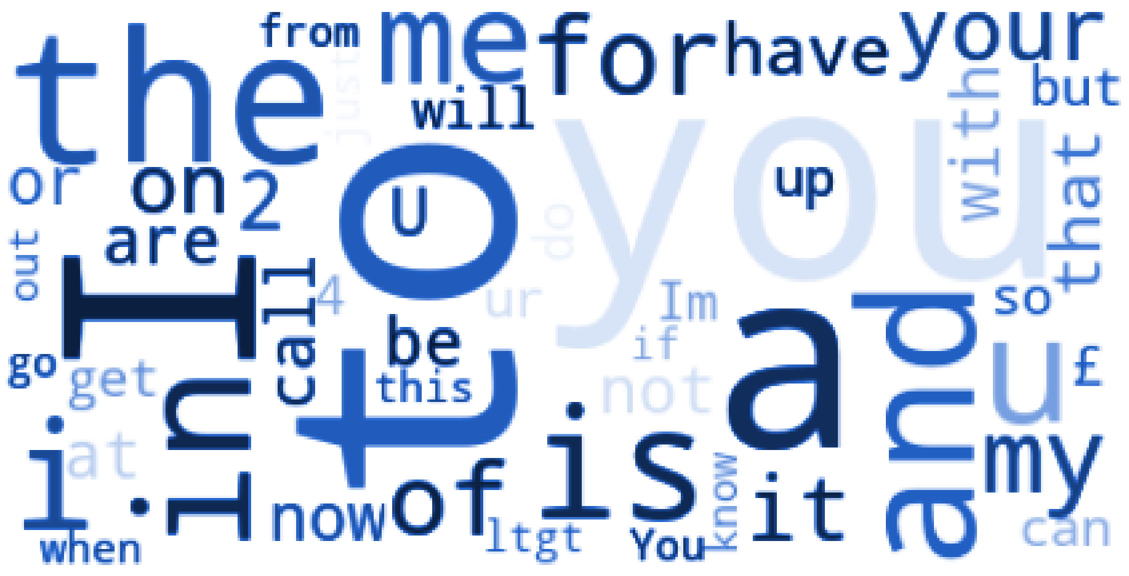
2. Exploratory Data Analysis

2.1 Data Wrangling

As we are dealing with NLP, it is necessary to transform text into numbers. We make basic model - bag of words. We use causal tokenizer TweetTokenizer . Basically, this part is about how to split sentence to the “words” (tokens) or more precise, split to a set of characters from which we can create bag of words. After creating bag of words we then finally can reach to our desired dataframe - words are features in dataframe. Each NaN value we changed to zero value.

2.2 Data Storytelling

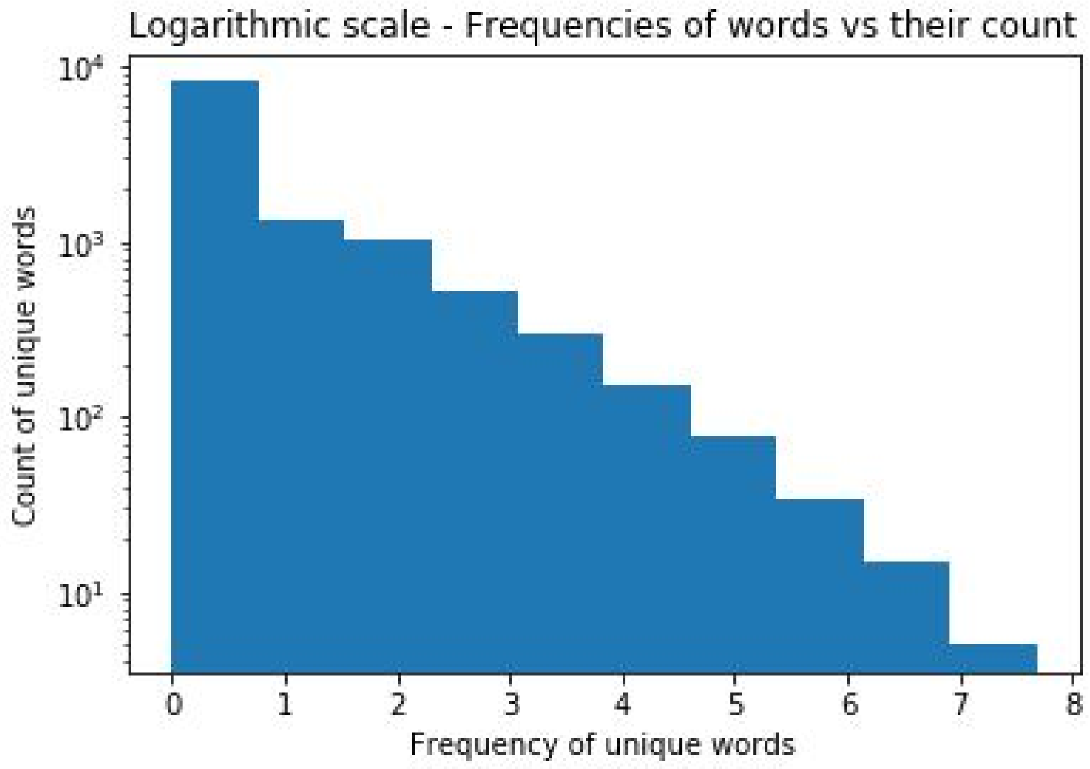
We will look closely on all words (referring as ‘words’), then ‘spam words’ (‘spam’) and then non-spam words (‘ham’).
Firstly take look on words that we have in our corpus. For illustration, here is Word Cloud of our messages:



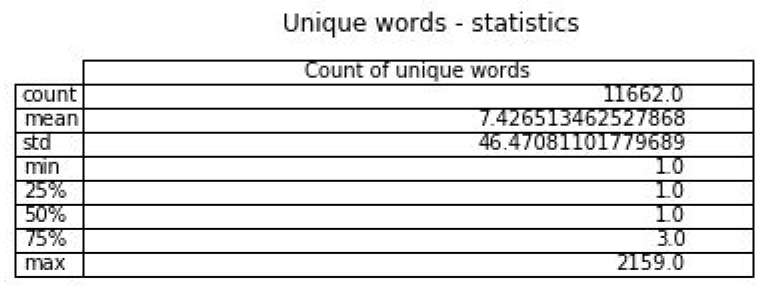
In our corpus we have 11662 unique words → tokens (we will be using words instead of tokens from now on). Messages contain emojis, various abbreviations (such ‘2U’). This

11662 token where counted after our preprocessing. Unique words statistics table shows us, that half of the words occur only once and highest frequency of word is 2159.

Here is visualization of frequencies of words vs. their count in logarithmic scale. Logarithmic scale is useful in cases as ours -- we have a huge amount of word that occur just few time and on the other hand there are few words that are very high in frequency).

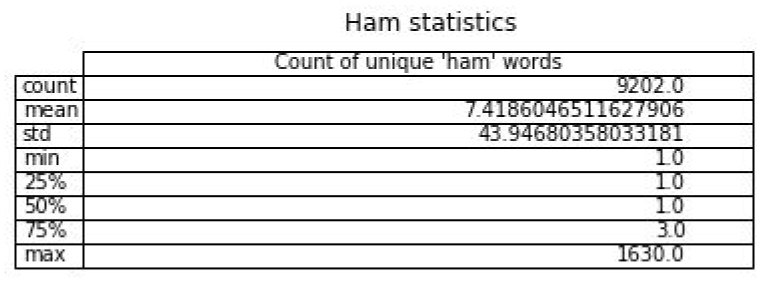


Now, having better understanding of our words, look on the set of words and its subsets: spam and ham words. We start with comparing their statistics:



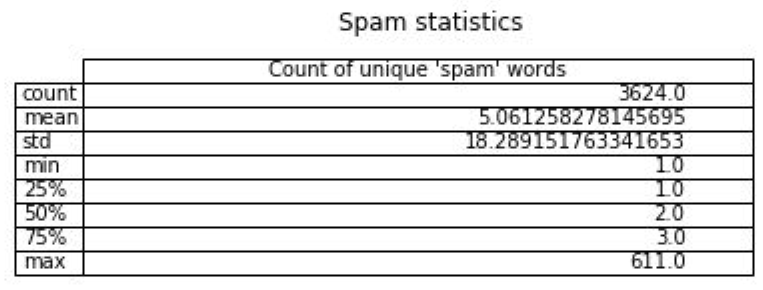
Unique words - statistics

	Count of unique words
count	11662.0
mean	7.426513462527868
std	46.47081101779689
min	1.0
25%	1.0
50%	1.0
75%	3.0
max	2159.0



Ham statistics

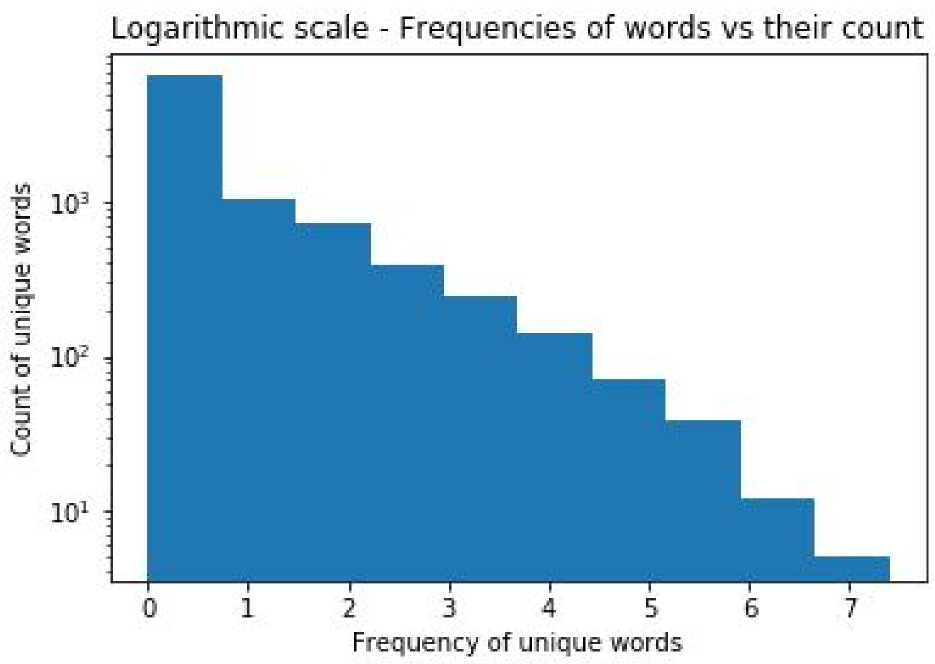
	Count of unique 'ham' words
count	9202.0
mean	7.4186046511627906
std	43.94680358033181
min	1.0
25%	1.0
50%	1.0
75%	3.0
max	1630.0

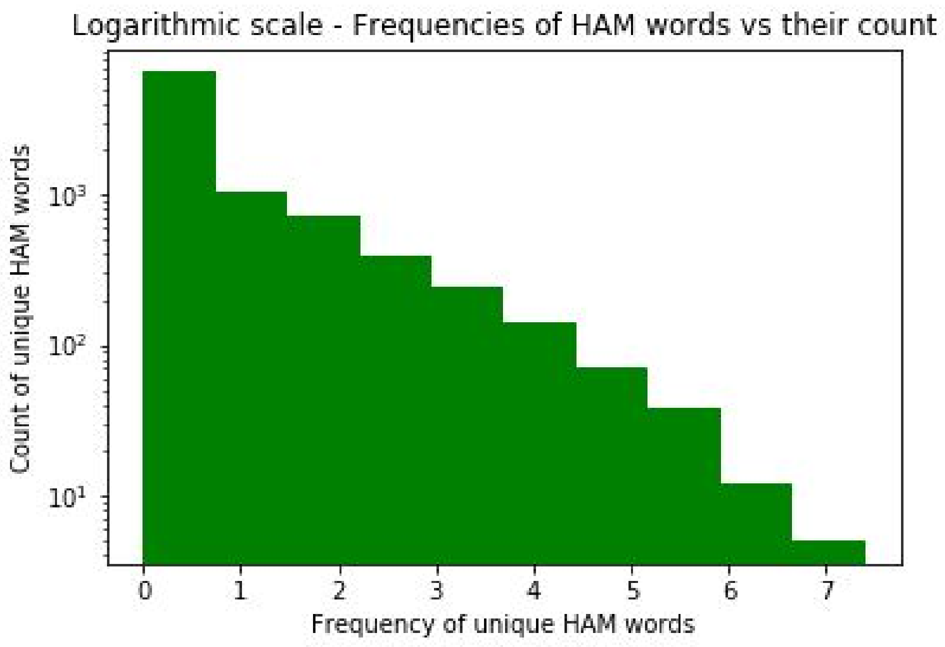


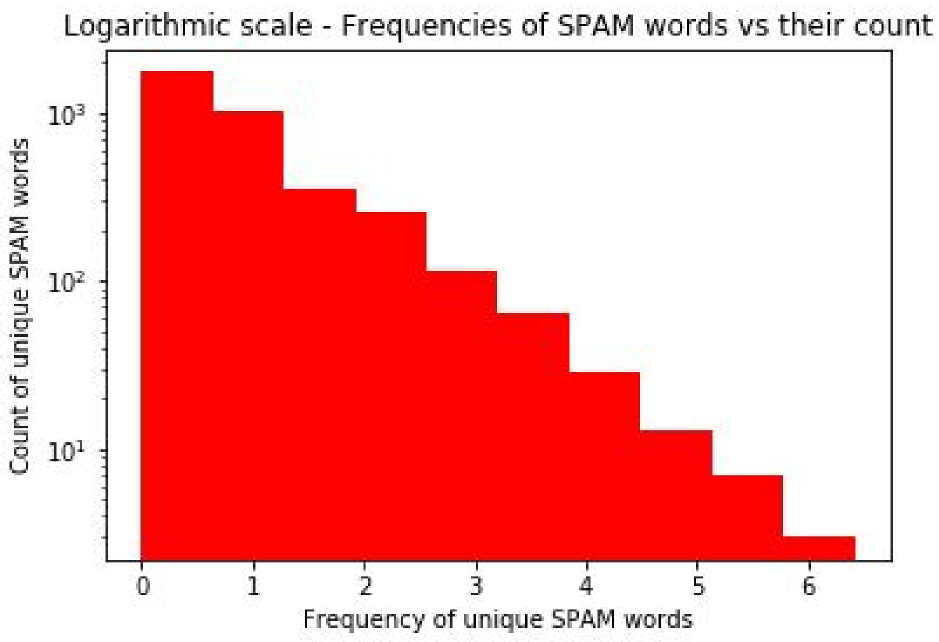
Spam statistics

	Count of unique 'spam' words
count	3624.0
mean	5.061258278145695
std	18.289151763341653
min	1.0
25%	1.0
50%	2.0
75%	3.0
max	611.0

To get better insight, let's look on histograms of frequencies of words/spam/ham versus count of unique words/spam/ham in logarithmic scale.


Logarithmic scale - Frequencies of words vs their count
Count of unique words
10^3
10^2
10^1
0 1 2 3 4 5 6 7
Frequency of unique words


Logarithmic scale - Frequencies of HAM words vs their count
Count of unique HAM words
10^3
10^2
10^1
0 1 2 3 4 5 6 7
Frequency of unique HAM words



The list of the most frequent words that are occuring in spam but not in the ham:

['Call', 'won', 'Txt', 'free', '1', 'from', 'week', 'ur', '£', 'text', 'Nokia', 'or', '4', 'mobile', 'who', 'You', 'contact', 'FREE', 'claim', 'Your', 'service', 'prize', 'txt', 'To', 'STOP', 'only', 'reply', 'our', '16']

2.3 Inferential Statistics

In our data set of SMS we have spam and ham messages. We want to find out, whether some features is more important than others in order to predict whether is spam or ham. For the sake of demonstration we will pick just one word ("Call").

Firstly, we need to find out whether the CLT (Central Limit Theorem) applies:

1) We have 5572 messages, so N is large enough -- check
2) We assume that independent condition is satisfied as well

We state our null and alternative hypothesis and then we test them. We set $\boldsymbol{\alpha}$ = 0.5

$\mathbf{H_0}$: $\mu_{\text{spam-call}} = \mu_{\text{ham-call}}$

$\mathbf{H_A}$: $\mu_{\text{spam-call}} \neq \mu_{\text{ham-call}}$

In this Jupyter Notebook you can find details about testing our hypothesis. We here concluded our findings.

Our 95% confidence interval is <0.1803, 0.2435> . And therefore we can't reject null hypothesis. Word 'call' is widely used in 'ham' communication as in the 'spam'. So word 'call' alone is not sufficient to decide whether message is a spam or ham.

2.4 In Depth Analysis

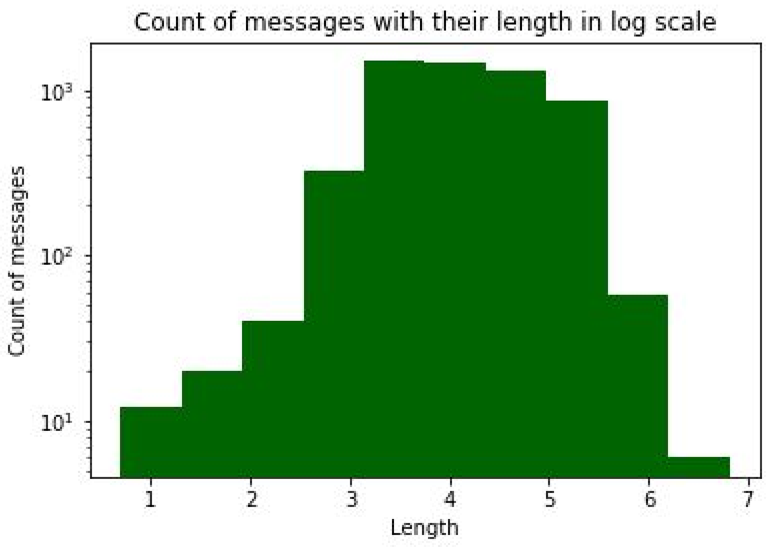
Let's dive into our intuition about data set and what assumption can we valide as good ones. First let's sum up some fact about SMS.
The maximum length of text message that you can send is 918 characters. However, if you send more than 160 characters then your message will be broken down into chunks of 153 characters before being sent to the recipient's handset.

Here is description of the length of whole dataset:

```
count    5572.000000
mean       80.489950
std        59.942907
min         2.000000
25%        36.000000
50%        62.000000
```

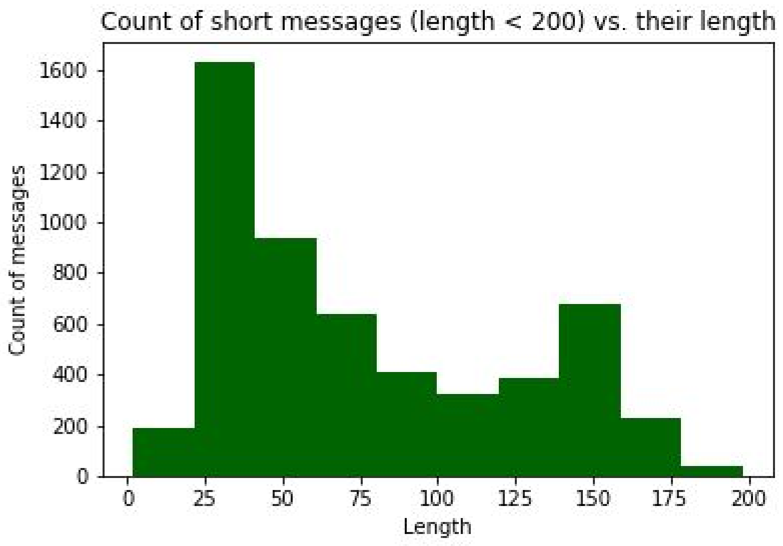
75%	122.000000
max	910.000000

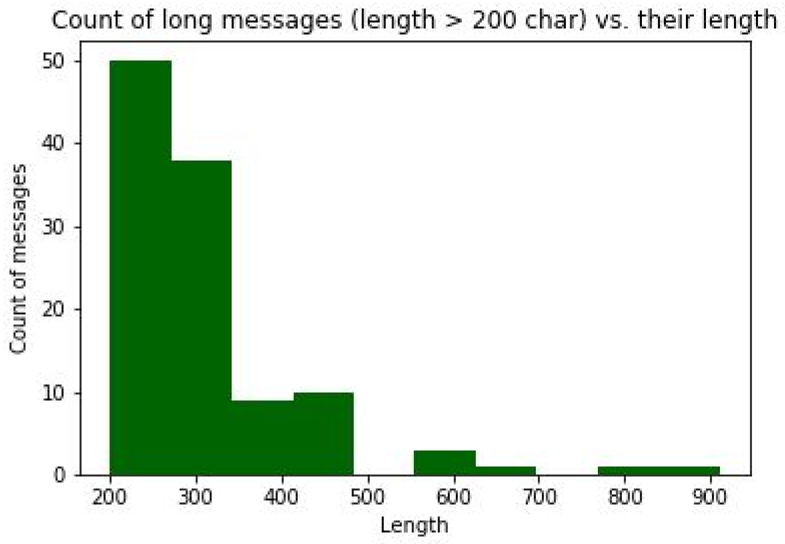
To visualize our distribution, look on the pictures below. First one is distribution in log scale.



The other two are visualized in unscaled distribution. We split the distribution into two pictures, to see first distributions of SMS which length is less than 200 characters and second with SMS's length more than 200 characters (we choose length = 200 only because of resolution reasons).

In our analysis we will split short and long messages whether they belong to the top 25% or not ('length' = 122).





1) *Really short messages tends to be a 'ham'.*

For example, someone texts you just 'Ok' as an answer to some question. As they do not contain a lot of information and these kind of messages are part of well-known conversation, then our intuition tells us that such a short messages have to be a 'ham'.

Let's explore this. We start with short messages ('length' = 122, 75% of all messages in our dataset) and then we look at 50 shortest messages, what is their contain and whether is 'spam' or 'ham'.

Number of short sms spam: 140 from 747 spam messages

Spam ratio = 0.18741633199464525 (short spam/all spam messages)

Number of short sms ham: 4025 from 4825 ham messages

Ham ratio = 0.8341968911917098 (short ham/all ham messages)

Unique content of 50 shortest messages:

'Ok', 'Yup', '645', 'Ok.', ':) ', 'Ok..', 'Okie', 'U 2.', 'Ok...', 'G.W.R', 'Y lei?', 'Yup...', 'ALRITE', 'Okie...', 'Where @', 'Oh ok..', 'Ok lor.', 'Nite...', 'Havent.', ':-) :-)', 'Thanx...', 'Thank u!', 'Beerage?', 'U too...', 'My phone', "I'm home.", 'Yup ok...', 'How come?'

2) *What about long messages? Are they spam or ham? Too long messages would be 'ham' as well - spam usually tries to "sells something" and therefore too long messages would be big nuisance (although even short spam messages are nuisance). Therefore we will look at messages that are longer than two standard SMS (>300 characters).*

There is actually only 41 messages in our dataset being longer than 300 characters and all of them are 'ham'.

So according length of the message we can say that if message is too short or too long, it is most probably a 'ham'. However, if it is message having about 160 characters, we can't determine whether it is 'ham' or 'spam' using length as the only feature.

3) Messages containing more than 2 exclamation marks tends to be spam.

Intuitively we would say this so, although sometimes people are overusing exclamations marks when they are feeling strong emotions. After closer look at our data, we cannot say whether this is True or False as we have only few messages having more than 2 exclamations marks (N=78).

4) What about 2-grams such as "call" and "buy"? Can we say they tend to be a spam?

After filtering out messages containing "call" and "buy" we got 70 messages and all are labeled as a 'spam'. Spam ratio for messages containing spam is 0.09. Although ration is small but we haven't found any messages containing those two words and being 'ham'.
Link to GitHub with Jupyter Notebook.

3. Modelling

We have analyzed three various models (three various classifiers), using different sets of features.

1) 'length'
2) 'length' + 'num_words'
3) 'length' + 'num_words' + 'text'

Note: By 'text' feature we mean here TFIDF matrix, with its corresponding words (tokens) after preprocessing.

Our task is binary classification problem (is it a SMS ‘spam’ or ‘ham’?). We will used following supervised machine learning classifiers for our predictive models: Logistic Regression, Support Vector Machine, Random Forest.

For evaluation we do not use accuracy score (error rate score) as it works badly in imbalance dataset. Better choices are:

- The Area Under the ROC curve (AUC)
- F1 Score
- Cohen’s Kappa

3.1. Data Preprocessing

1. **Numerical**

   Create engineered features ‘length’ and ‘num_words’. The feature ‘length’ simply tells us how many characters each SMS message has (spaces included). The feature ‘num_words’ tells us how many words SMS messages has (after splitting words by space).

   Text

   We proprocess text by applying TweetTokenizer(), then transforming to the TFIDF matrix, and apply one more transformation - PCA (principal component analysis) to reduce dimension of the matrix (dimension is high, as unique words from corpus/messages are features).
2. **Data Splitting**

   We splitted data 80:20 (80% train data, 20% test data)
3. **Weighting**

   We are handling imbalance dataset and therefore we set in each classifier parameter class_weight='balanced'.

4. **Scaling and feature selection**

   For our third model (when we using 'text' feature as well) is scaling necessary step. TFIDF matrix contains number within interval <0, 1> and therefore we use MinMaxScaler to scale numerical features.

5. **Hyperparameters**

   - classifier

   - parameters of classifier (defined defaults parameters and set of parameters which are being used are listed in Note below)

   - with 'text' feature: n-grams,

   TweetTokenizer/get rid of punctuation

   PCA, number of components,

   TFIDF, min/max document frequency.

6. **Evaluation**

   Our data set is imbalanced data set as well, where ~ 13% of messages being spam messages. Sometimes you can have extremes such as having 1-2% of "negative" value (complainants, cancellation of flight, etc). If we were using a guess for our predictions being positive values, we would gain accuracy score 98-99%! However, we want not only to have a high accuracy score, we want to have good precision and recall. This mean we want to have good balance between false positive and true negatives (misclassified spam as hams and vice versa). F1 score is the suitable option when one needs to have a balance between Precision and Recall AND has an imbalanced data set.

   Cohen's kappa statistic is a very good measure that can handle very well both multi-class and imbalanced class problems. We have binary problem and therefore we decided to choose F1 score as our metric.

NOTE: Before we look closely to our models and used classifier, let's sum up some default settings for our classifier:

Logistic Regression, defaults: solver='liblinear', penalty='l1', C=1.0

Solver = {‘liblinear’, ‘sag’, ‘newton-cg’, ‘lbfgs’}

Penalty = {‘l1’, ‘l2’}

C = {0.1, 0.5, 1.0}

Random Forest, defaults: bootstrap=True, n_estimators=100)

Bootstrap = {True, False}

N_estimators = {1, 50, 1000}

Support Vector Machines, defaults: kernel='poly', C=1.0, coef0=0.0

C = {0.1, 0.5, 1.0}

Coef = {0.0, 0.5}

Kernel = {‘linear’, ‘poly’ , ‘sigmoid’}

In the following subsections, a structure is following: we will closely study 3 models, where hyperparameters features will be fixed. We observe how model will perform with changing other hyperparameters for each classifier and describe our observations.

3.2. Results

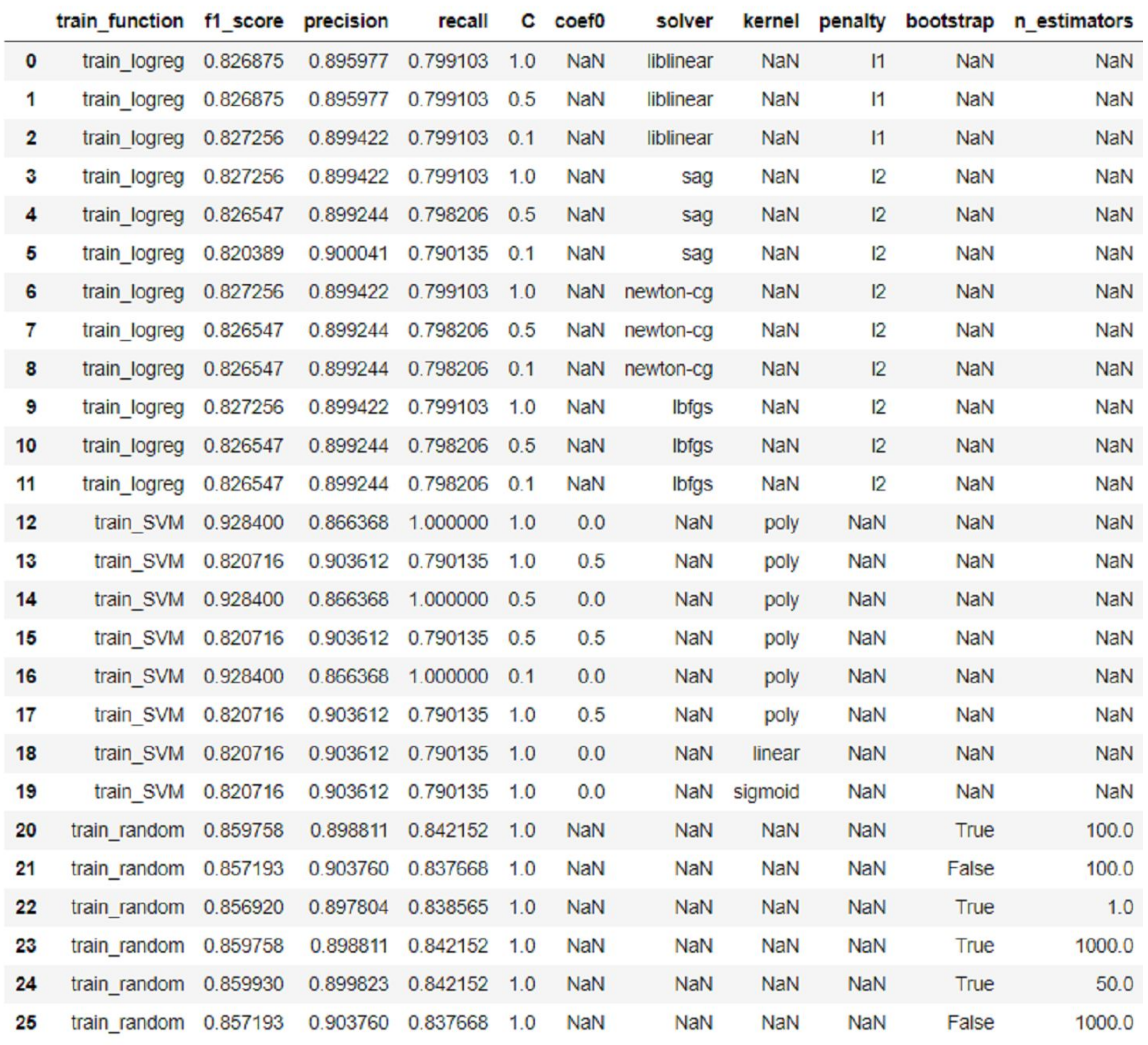
3.2.1 ‘Length’ model

Jupyter Notebook with a code :

https://github.com/bkarolina/DS-Projects-Springboard/blob/master/Capstone%20Projects/SMS_length.ipynb

This is the simplest option using only one feature “length”. It is not necessary to scale this feature as in this case it is the only feature, however in our code you line when you can turn off/on this possibility for future experiments.

For Logistic Regression and Random Forest classifiers the changing of other hyperparameters have a small effect. In the case of SVM changing C and coef0 hyperparameters there is visible difference in f1_score results.
We see, that SVM performs the best with f1_score = 0.9032.



	train_function	f1_score	precision	recall	C	coef0	solver	kernel	penalty	bootstrap	n_estimators
0	train_logreg	0.826875	0.895977	0.799103	1.0	NaN	liblinear	NaN	l1	NaN	NaN
1	train_logreg	0.826875	0.895977	0.799103	0.5	NaN	liblinear	NaN	l1	NaN	NaN
2	train_logreg	0.827256	0.899422	0.799103	0.1	NaN	liblinear	NaN	l1	NaN	NaN
3	train_logreg	0.827256	0.899422	0.799103	1.0	NaN	sag	NaN	l2	NaN	NaN
4	train_logreg	0.826547	0.899244	0.798206	0.5	NaN	sag	NaN	l2	NaN	NaN
5	train_logreg	0.820389	0.900041	0.790135	0.1	NaN	sag	NaN	l2	NaN	NaN
6	train_logreg	0.827256	0.899422	0.799103	1.0	NaN	newton-cg	NaN	l2	NaN	NaN
7	train_logreg	0.826547	0.899244	0.798206	0.5	NaN	newton-cg	NaN	l2	NaN	NaN
8	train_logreg	0.826547	0.899244	0.798206	0.1	NaN	newton-cg	NaN	l2	NaN	NaN
9	train_logreg	0.827256	0.899422	0.799103	1.0	NaN	lbfgs	NaN	l2	NaN	NaN
10	train_logreg	0.826547	0.899244	0.798206	0.5	NaN	lbfgs	NaN	l2	NaN	NaN
11	train_logreg	0.826547	0.899244	0.798206	0.1	NaN	lbfgs	NaN	l2	NaN	NaN
12	train_SVM	0.928400	0.866368	1.000000	1.0	0.0	NaN	poly	NaN	NaN	NaN
13	train_SVM	0.820716	0.903612	0.790135	1.0	0.5	NaN	poly	NaN	NaN	NaN
14	train_SVM	0.928400	0.866368	1.000000	0.5	0.0	NaN	poly	NaN	NaN	NaN
15	train_SVM	0.820716	0.903612	0.790135	0.5	0.5	NaN	poly	NaN	NaN	NaN
16	train_SVM	0.928400	0.866368	1.000000	0.1	0.0	NaN	poly	NaN	NaN	NaN
17	train_SVM	0.820716	0.903612	0.790135	1.0	0.5	NaN	poly	NaN	NaN	NaN
18	train_SVM	0.820716	0.903612	0.790135	1.0	0.0	NaN	linear	NaN	NaN	NaN
19	train_SVM	0.820716	0.903612	0.790135	1.0	0.0	NaN	sigmoid	NaN	NaN	NaN
20	train_random	0.859758	0.898811	0.842152	1.0	NaN	NaN	NaN	NaN	True	100.0
21	train_random	0.857193	0.903760	0.837668	1.0	NaN	NaN	NaN	NaN	False	100.0
22	train_random	0.856920	0.897804	0.838565	1.0	NaN	NaN	NaN	NaN	True	1.0
23	train_random	0.859758	0.898811	0.842152	1.0	NaN	NaN	NaN	NaN	True	1000.0
24	train_random	0.859930	0.899823	0.842152	1.0	NaN	NaN	NaN	NaN	True	50.0
25	train_random	0.857193	0.903760	0.837668	1.0	NaN	NaN	NaN	NaN	False	1000.0

What we have learned about each classifier with corresponding hyperparameters?

In the case of Logistic classifier, as in the case of Random Forest classifier we can see that change of hyperparameters has really low effect on f1 score (from their default settings). On the other hand, change of hyperparameters in SVM classifier case has visible difference on f1 score.

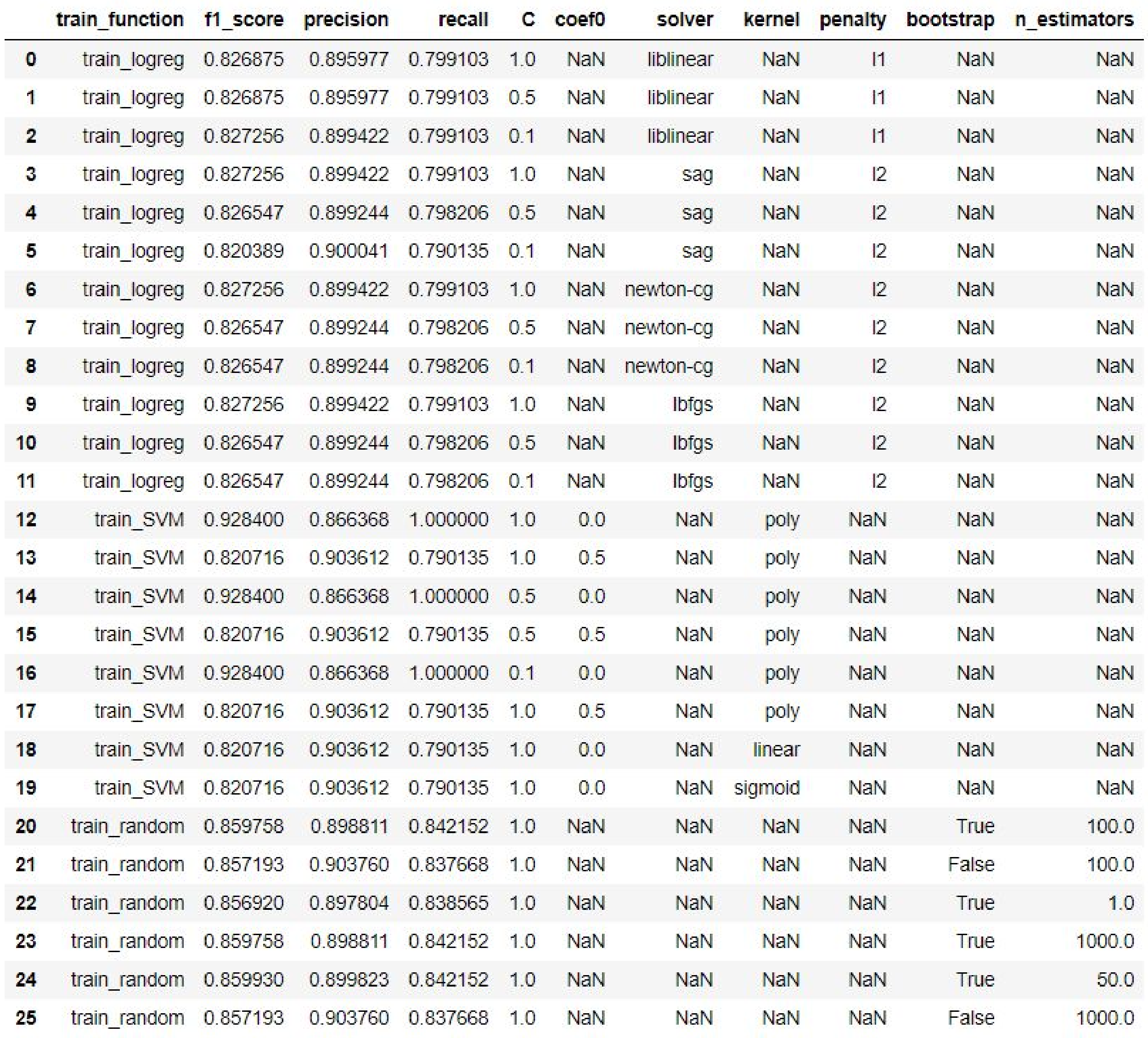
The change of coef0 from 0.5 to 0.0 drop our f1 score. On the other hand, change of C hyperparameter has almost no effect. The kernel “poly” performs the best.

3.2.2. ‘Length’ + ‘num_words’ model

Jupyter Notebook with code:
https://github.com/bkarolina/DS-Projects-Springboard/blob/master/Capstone%20Projects/SMS_length_numwords.ipynb

Here we observed similar results as in the previous case, using just “length” as a feature. This is not a surprise as those two features are “connected” together, correlated. With longer messages we expect the message contains more words and therefore more characters as well.

SVM performs the best with f1_score = 0.9284. Corresponding hyperparameters: kernel=’poly’, coef0 = 0.0, C = {0, 0.5, 1.0}.



	train_function	f1_score	precision	recall	C	coef0	solver	kernel	penalty	bootstrap	n_estimators
0	train_logreg	0.826875	0.895977	0.799103	1.0	NaN	liblinear	NaN	l1	NaN	NaN
1	train_logreg	0.826875	0.895977	0.799103	0.5	NaN	liblinear	NaN	l1	NaN	NaN
2	train_logreg	0.827256	0.899422	0.799103	0.1	NaN	liblinear	NaN	l1	NaN	NaN
3	train_logreg	0.827256	0.899422	0.799103	1.0	NaN	sag	NaN	l2	NaN	NaN
4	train_logreg	0.826547	0.899244	0.798206	0.5	NaN	sag	NaN	l2	NaN	NaN
5	train_logreg	0.820389	0.900041	0.790135	0.1	NaN	sag	NaN	l2	NaN	NaN
6	train_logreg	0.827256	0.899422	0.799103	1.0	NaN	newton-cg	NaN	l2	NaN	NaN
7	train_logreg	0.826547	0.899244	0.798206	0.5	NaN	newton-cg	NaN	l2	NaN	NaN
8	train_logreg	0.826547	0.899244	0.798206	0.1	NaN	newton-cg	NaN	l2	NaN	NaN
9	train_logreg	0.827256	0.899422	0.799103	1.0	NaN	lbfgs	NaN	l2	NaN	NaN
10	train_logreg	0.826547	0.899244	0.798206	0.5	NaN	lbfgs	NaN	l2	NaN	NaN
11	train_logreg	0.826547	0.899244	0.798206	0.1	NaN	lbfgs	NaN	l2	NaN	NaN
12	train_SVM	0.928400	0.866368	1.000000	1.0	0.0	NaN	poly	NaN	NaN	NaN
13	train_SVM	0.820716	0.903612	0.790135	1.0	0.5	NaN	poly	NaN	NaN	NaN
14	train_SVM	0.928400	0.866368	1.000000	0.5	0.0	NaN	poly	NaN	NaN	NaN
15	train_SVM	0.820716	0.903612	0.790135	0.5	0.5	NaN	poly	NaN	NaN	NaN
16	train_SVM	0.928400	0.866368	1.000000	0.1	0.0	NaN	poly	NaN	NaN	NaN
17	train_SVM	0.820716	0.903612	0.790135	1.0	0.5	NaN	poly	NaN	NaN	NaN
18	train_SVM	0.820716	0.903612	0.790135	1.0	0.0	NaN	linear	NaN	NaN	NaN
19	train_SVM	0.820716	0.903612	0.790135	1.0	0.0	NaN	sigmoid	NaN	NaN	NaN
20	train_random	0.859758	0.898811	0.842152	1.0	NaN	NaN	NaN	NaN	True	100.0
21	train_random	0.857193	0.903760	0.837668	1.0	NaN	NaN	NaN	NaN	False	100.0
22	train_random	0.856920	0.897804	0.838565	1.0	NaN	NaN	NaN	NaN	True	1.0
23	train_random	0.859758	0.898811	0.842152	1.0	NaN	NaN	NaN	NaN	True	1000.0
24	train_random	0.859930	0.899823	0.842152	1.0	NaN	NaN	NaN	NaN	True	50.0
25	train_random	0.857193	0.903760	0.837668	1.0	NaN	NaN	NaN	NaN	False	1000.0

Results and observations are the same as in the previous model. That is not such a big surprise as those two engineered features are correlated. The best result of f1 score is for SVM classifier with values 0.9824, kernel = ‘poly’, coef0 = 0.0, C = {1.0, 0.5, 0.1}.

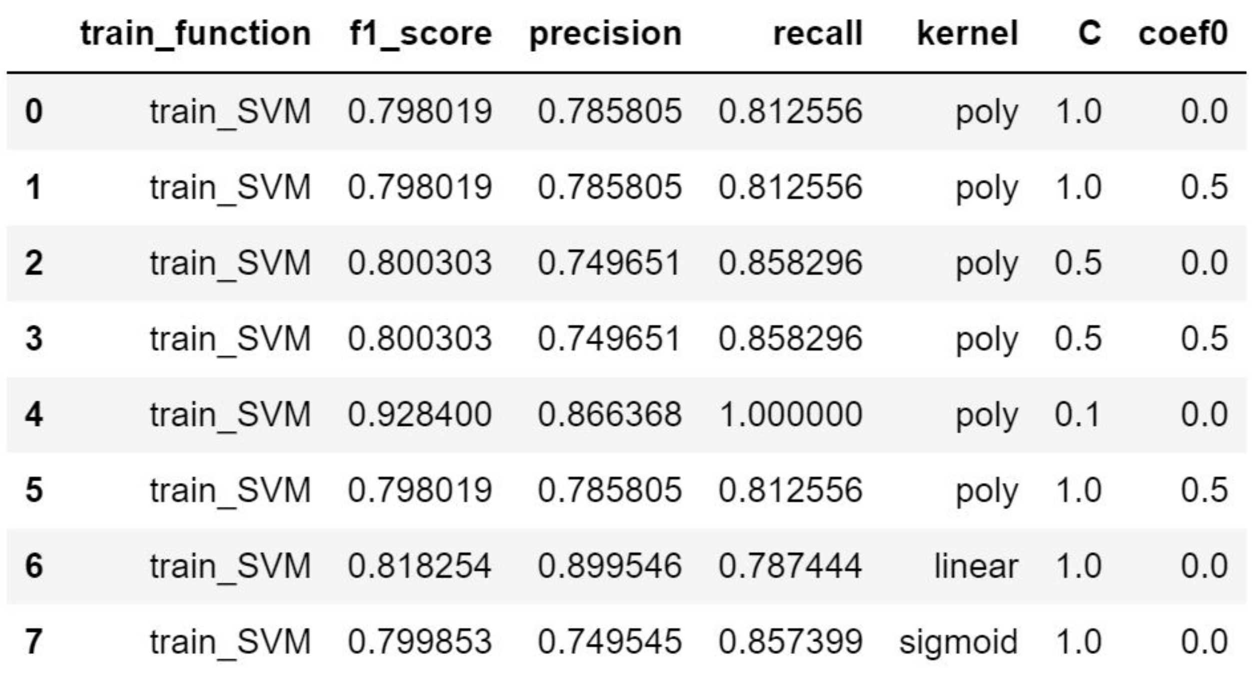
3.2.3 ‘Length’ + ‘num_words’ + ’text’ model

Jupyter Notebook with the code:
https://github.com/bkarolina/DS-Projects-Springboard/blob/master/Capstone%20Projects/SMS_length_numwords_text.ipynb

As mentioned, our ‘text’ feature is TFIDF matrix. Now we are facing problem having more features than data points (significantly more columns then rows), which is common within NLP problems. Therefore computation time is longer time as before.

Especially time consuming is SVM classifiers. For Logistic Regression and Random Forest it takes few seconds to do computation (with worse f1 score) than in case of SVM, which takes several minutes to 2-3 hours (with the whole set of hyperparameters presents in the following tables).

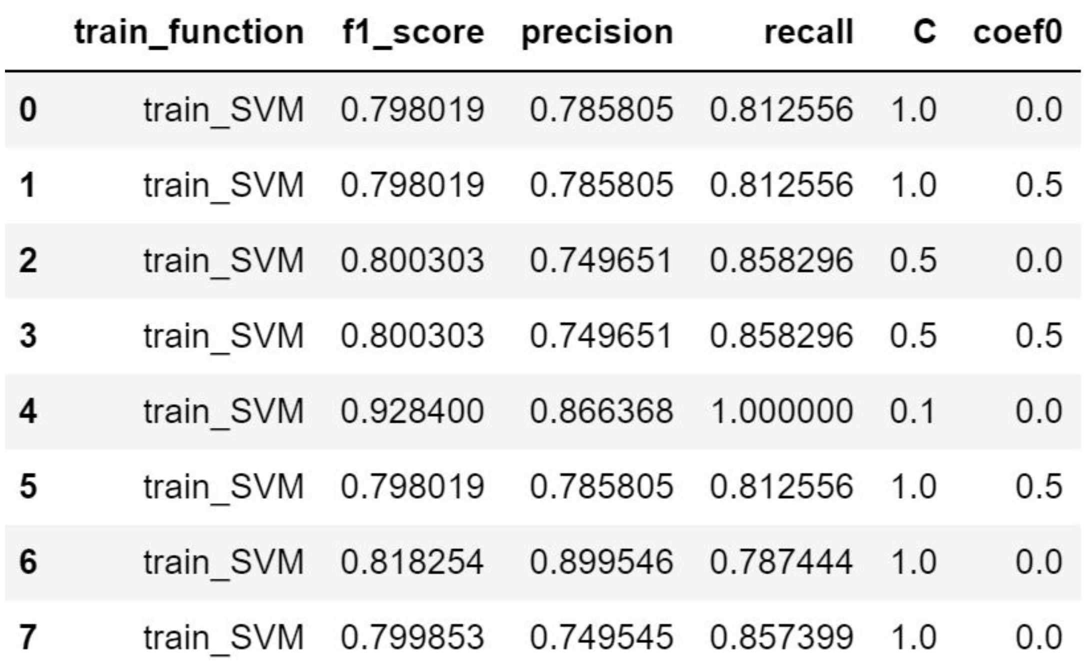
Table for SVM classifier, without using PCA dimension reduction.



	train_function	f1_score	precision	recall	kernel	C	coef0
0	train_SVM	0.798019	0.785805	0.812556	poly	1.0	0.0
1	train_SVM	0.798019	0.785805	0.812556	poly	1.0	0.5
2	train_SVM	0.800303	0.749651	0.858296	poly	0.5	0.0
3	train_SVM	0.800303	0.749651	0.858296	poly	0.5	0.5
4	train_SVM	0.928400	0.866368	1.000000	poly	0.1	0.0
5	train_SVM	0.798019	0.785805	0.812556	poly	1.0	0.5
6	train_SVM	0.818254	0.899546	0.787444	linear	1.0	0.0
7	train_SVM	0.799853	0.749545	0.857399	sigmoid	1.0	0.0

As we can see, in this case the performance haven’t improved. Let’s take a look how this will change using PCA dimension reduction. We present result for PCA with n_components = {100, 500, 1000}, first for SVM classifier only, later for Random Forest and Logistic Regression as well.

Table for SVM classifier, PCA with n_components = 100.



	train_function	f1_score	precision	recall	C	coef0
0	train_SVM	0.798019	0.785805	0.812556	1.0	0.0
1	train_SVM	0.798019	0.785805	0.812556	1.0	0.5
2	train_SVM	0.800303	0.749651	0.858296	0.5	0.0
3	train_SVM	0.800303	0.749651	0.858296	0.5	0.5
4	train_SVM	0.928400	0.866368	1.000000	0.1	0.0
5	train_SVM	0.798019	0.785805	0.812556	1.0	0.5
6	train_SVM	0.818254	0.899546	0.787444	1.0	0.0
7	train_SVM	0.799853	0.749545	0.857399	1.0	0.0

Table for SVM classifier, PCA with n_components = 500.

	train_function	f1_score	precision	recall	kernel	C	coef0
0	train_SVM	0.928400	0.866368	1.000000	poly	1.0	0.0
1	train_SVM	0.928400	0.866368	1.000000	poly	1.0	0.5
2	train_SVM	0.928400	0.866368	1.000000	poly	0.5	0.0
3	train_SVM	0.928400	0.866368	1.000000	poly	0.5	0.5
4	train_SVM	0.928400	0.866368	1.000000	poly	0.1	0.0
5	train_SVM	0.928400	0.866368	1.000000	poly	1.0	0.5
6	train_SVM	0.824923	0.891064	0.797309	linear	1.0	0.0
7	train_SVM	0.928400	0.866368	1.000000	sigmoid	1.0	0.0

Table for SVM classifier, PCA with n_components = 1000.

	train_function	f1_score	precision	recall	kernel	C	coef0
0	train_SVM	0.928400	0.866368	1.000000	poly	1.0	0.0
1	train_SVM	0.928400	0.866368	1.000000	poly	1.0	0.5
2	train_SVM	0.928400	0.866368	1.000000	poly	0.5	0.0
3	train_SVM	0.928400	0.866368	1.000000	poly	0.5	0.5
4	train_SVM	0.928400	0.866368	1.000000	poly	0.1	0.0
5	train_SVM	0.928400	0.866368	1.000000	poly	1.0	0.5
6	train_SVM	0.825124	0.882570	0.799103	linear	1.0	0.0
7	train_SVM	0.928400	0.866368	1.000000	sigmoid	1.0	0.0

Using n_components = 100 was too low in order to learn good/better to correctly recognize spam from ham. In other two cases the results were the same, and better than using n_compenents = 100.

Table for Random Forest and Logistic Regression classifiers,

PCA with n_components = 100.

	f1_score	train_function	C	solver	penalty
1	0.878984	train_logreg	0.5	liblinear	l1
0	0.874926	train_logreg	1.0	liblinear	l1
13	0.858637	train_random	1.0	liblinear	l1
21	0.850227	train_random	1.0	liblinear	l1
14	0.847529	train_random	1.0	liblinear	l1
18	0.842359	train_random	1.0	liblinear	l1
12	0.842359	train_random	1.0	liblinear	l1
20	0.841724	train_random	1.0	liblinear	l1
15	0.841724	train_random	1.0	liblinear	l1
19	0.841340	train_random	1.0	liblinear	l1
17	0.840661	train_random	1.0	liblinear	l1
16	0.840296	train_random	1.0	liblinear	l1
2	0.827256	train_logreg	0.1	liblinear	l1
6	0.816717	train_logreg	1.0	newton-cg	l2
9	0.816717	train_logreg	1.0	lbfgs	l2
3	0.816717	train_logreg	1.0	sag	l2
4	0.816010	train_logreg	0.5	sag	l2
10	0.816010	train_logreg	0.5	lbfgs	l2
7	0.816010	train_logreg	0.5	newton-cg	l2
5	0.815667	train_logreg	0.1	sag	l2
8	0.815667	train_logreg	0.1	newton-cg	l2
11	0.815667	train_logreg	0.1	lbfgs	l2

Table for Random Forest and Logistic Regression classifiers,

PCA with n_components = 500.

	f1_score	train_function	C	solver	penalty
1	0.879706	train_logreg	0.5	liblinear	l1
0	0.879064	train_logreg	1.0	liblinear	l1
17	0.838605	train_random	1.0	liblinear	l1
9	0.831677	train_logreg	1.0	lbfgs	l2
3	0.831677	train_logreg	1.0	sag	l2
6	0.831677	train_logreg	1.0	newton-cg	l2
2	0.827256	train_logreg	0.1	liblinear	l1
21	0.825825	train_random	1.0	liblinear	l1
7	0.822808	train_logreg	0.5	newton-cg	l2
10	0.822808	train_logreg	0.5	lbfgs	l2
4	0.822808	train_logreg	0.5	sag	l2
13	0.821970	train_random	1.0	liblinear	l1
14	0.818401	train_random	1.0	liblinear	l1
16	0.818198	train_random	1.0	liblinear	l1
19	0.818198	train_random	1.0	liblinear	l1
15	0.817342	train_random	1.0	liblinear	l1
20	0.817342	train_random	1.0	liblinear	l1
8	0.814011	train_logreg	0.1	newton-cg	l2
5	0.814011	train_logreg	0.1	sag	l2
11	0.814011	train_logreg	0.1	lbfgs	l2
12	0.806900	train_random	1.0	liblinear	l1
18	0.806900	train_random	1.0	liblinear	l1

Springboard

Table for Random Forest and Logistic Regression classifiers,

PCA with n_components = 1000.

	f1_score	train_function	C	solver	penalty
0	0.881229	train_logreg	1.0	liblinear	l1
1	0.879706	train_logreg	0.5	liblinear	l1
21	0.856935	train_random	1.0	liblinear	l1
14	0.844948	train_random	1.0	liblinear	l1
13	0.841871	train_random	1.0	liblinear	l1
20	0.835431	train_random	1.0	liblinear	l1
15	0.835431	train_random	1.0	liblinear	l1
3	0.834395	train_logreg	1.0	sag	l2
6	0.834395	train_logreg	1.0	newton-cg	l2
9	0.834395	train_logreg	1.0	lbfgs	l2
16	0.828060	train_random	1.0	liblinear	l1
7	0.828009	train_logreg	0.5	newton-cg	l2
10	0.828009	train_logreg	0.5	lbfgs	l2
4	0.828009	train_logreg	0.5	sag	l2
2	0.827256	train_logreg	0.1	liblinear	l1
19	0.824058	train_random	1.0	liblinear	l1
11	0.821087	train_logreg	0.1	lbfgs	l2
8	0.821087	train_logreg	0.1	newton-cg	l2
5	0.821087	train_logreg	0.1	sag	l2
18	0.806262	train_random	1.0	liblinear	l1
12	0.806262	train_random	1.0	liblinear	l1
17	0.797686	train_random	1.0	liblinear	l1

First of all, if each case of different number of components of PCA, Logistic Regression with kernel = ‘liblinear’, penalty=’l1’ and c = {1.0, 0.5} perform the best. The most successful one was in the case PCA = 1000, C = 1.0 with resulting f1_score = 0.8812.
 PCA has helped improve f1_score in general. For a case PCA = 100 we can see that for Random Forest was this option better than for Logistic Regression (except two cases mention at the beginning).

4. Discussion

4.1. Assumptions and limitations

Text messages are collected from United Kingdom. First of all, the messages are in english and therefore our model won’t be useful in other language. For each country it should be re-trained. Not only for obvious reason as difference of language but there some other subtleties, such as in UK they used '£' instead of ‘$’ in USA.

4.2. Future Work

We would still like to have better accuracy (f1 score), as the client (who owns cell phone) would like to avoid any spam messages, on the other hand not to miss ham messages.
There are few things that can be incorporated in the future for deeper analysis:

1. Due to high dimensionality (/high cardinality/high number of features) we could used instead of nltk library other libraries as SpaCy or Gensim.
2. Extracting features like num_punctuation_char, number of stopwords, or any other we deleted (nouns, verbs…), etc.
3. Compare results when stop words/lemmatization/stemming is included. We have not consider include this step in our analysis, as we assumed we would probably lost some information about which words are connected more to the 'ham' or 'spam' messages.
4. Try to use under- and over sampling technique.
5. Try Neural Network models

4.3. Conclusion

In this Capstone Project we demonstrated the core techniques in NLP (bag of words, TFIDF, PCA) on Spam Messages Detection data set.

NLP tasks are tasks with extremely high cardinality. We like to reduce this cardinality, however then text preprocessing is extremely important as our decisions could change the prediction. We have chosen not to use stop words, lemmatization, stemming because we want to let our model learn something about vocabulary that people are using. For example sentence “How ya doin?” and “How are you doing?” has the same meaning, however second one is more formal. First one would appear only is friendly conversation - in message being ham and not spam. And we wanted to keep this kind of information.

When we were dealing with models without 'text' feature, it took approximately same computational time for each classifier. However, after adding feature 'text', SVM classifier took ~2 hours to train model (for all set of parameters). On the other hand, for Random Forest and Logistic Regression it took only few minutes.

PCA helped to improve f1 score for every model.

We found out that our best model was using TFIDF (‘len’ + ‘num_word’ + ‘text’ features), SVM classifier, PCA = 500 (or 1000), kernel = ‘poly’, coef0 and hyperparameter C didn’t play recognizable role, therefore we keep the defaults from sci-kit learn, with f1_score = 0.9284.