Spam Call Detection

SPAM CALL CLASSIFICATION USING GOOGLE SPEECH RECOGNITION AND MACHINE LEARNING ALGORITMS

Subhajit Chatterjee (SXC190070) and Sai Sreekar Reddy (SXS190008) | CS 6375.005 | 05/03/2020

ABSTRACT

With ever growing adaptation of cellphones as primary medium of communication, the persistent issue of dealing spam calls and texts from scammers has been growing significantly in last few years. In 2019, there was an 18% (Appendix :[24]) increase in spam calls around the globe as compared to the previous year, as reported by a Stockholm based firm 'Truecaller'. The U.S. continues to be in the top 10 list of the most spammed countries in the world.

Moreover, with robotically generated calls which matches the audio texture of human speech, it has become an easy weapon for scammers to bombard consumers with spam calls. These calls pertain to various topics, e.g. Debt Collection Calls, Political Propaganda, Fake Charities, Internal Revenue Service calls, etc.

In this project we tried to use some of the state-of-the-art machine learning algorithms and speech recognition tools to classify, hence detect the spam calls based on the content of the call. We have collected audio recordings of spam calls from publicly available sources along with several non-spam audio recordings and utilized google speech recognition tool to transcribe the audio to text. Further, we applied multiple algorithms for text classifications, e.g. Tree based, Gaussian Naïve Bayes Classifier, and Support Vector Machines

DATA COLLECTION & PREPROCESSING

Source Data Creation and transcription

- The raw audio files that were downloaded from the internet (Appendix: [1][2][3][4][5]) were in different formats such as *mp3*, *flac*, and *wav*.
- In order to facilitate easy editing of the audio files, all of them were converted into one uniform format - *wav*.
- Also, since the audio is being converted into text, in our application, stereotype audio was not necessary. Therefore, we have force-converted all the audio files into mono-channel, to maintain the uniformity across all the files.
- Though all the files are in *wav* format, many of them were quite long in duration. Hence all the wav files are trimmed for a duration of 180 seconds uniformly.
- We have used these trimmed wav audio files to transcribe into text using Google's Speech to Text conversion API.
- Transcript of each file has been stored as a single row with the corresponding label (spam or non-spam). All the transcripts have been exported to a single csv file which is used for further text processing as described below.
- We have written an automated script [Appendix -6] in python that performs all the tasks mentioned above. The script reads the files as described in first step, converts them to *wav*, converts them to mono-channel audio, trims each audio to 180 seconds, uploads each audio to Google Cloud Storage bucket located at (Appendix-[7]), transcribes each file into text using Google Speech-Text API and writes the transcribe to a local csv file (Appendix-[8]).
- We created a dataset of 1,234 recording transcriptions, with 103 tagged as Spam=1 (rest Spam= 0)

Data Preprocessing and Vectorization

- The transcripts in text form couldn't be directly used for any meaningful classification, hence they had to be vectorized

- As part of preprocessing before vectorization, we did the following steps:
    - Convert all characters to lowercase
    - Remove punctuation & special characters
    - Remove English stop words
    - Lemmatize the text to convert to root words (e.g. ‘playing’ and ‘plays’ both converts to ‘play’)
    - Used 2 words at a time as ngram token, since this retains more information and context of associations in the data
- In vectorization, we performed both Count Vectorization and Term-Frequency-Inverse Document Frequency (TF-IDF) based vectorization-
    - Count Vectorization:
        - Here we count the frequency of each token within a single transcript, the more frequent a word, the more weightage it has.
        - We used Count Vectorized data for our own implementation of decision tree and naive-bayes, since integer counts of each token (considered as each features) are convenient to discretize
    - TD-IDF Vectorization:
        - There may be trivial words like ‘call’ or ‘please’ which are not usual stop words, but more frequent in our data context (pertaining to phone calls).
        - The higher frequency of these trivial words shall put more weightage on them, which may not give any meaningful disproportionation of the data into spam/non-spam
        - Hence, the higher frequency tokens need to inversely weighted against their occurrences across all transcripts, i.e. if a word is present frequently in all transcripts, then it won’t be contributing much to segregation of spam/non-spam

        - Finally, we normalized the data using l2 (root of sum of squared distances) norm, which would help with classifiers like Support Vector Machines (scaled data is better classified)
    - Final vectorized data are available in google cloud platform:
        - Count Vectorized data (Appendix: [10])
        - TF-IDF Vectorized data (Appendix:[9])
    - Script is provided here (Appendix:[11])

ALGORITHMS IMPLEMENTED

Own implementations:

Cosine Similarity (script: (Appendix:[11]):

Before trying out any other classification algorithm, we tried finding out the similarity amongst transcripts based on their cosine similarity:

Cosine Similarity of vectors (**a,b**) = **a**.**b**/|**a**|.|**b**|

Where, the closer the two vectors are in a n-dimensional space, the smaller is the angle of separation between, hence the higher the cosine value.

We used all 1234 transcripts (since cosine similarity doesn't need any target labels, no need of separate train/test split and validation) and used their TF-IDF vectorized data to find the cosine similarity. The observations are discussed in the later section.

Decision Tree based Bagging (script: Appendix [23])

- We have used the same decision tree code from Project 1 & 2.
- We have set 20% of the raw data as the test data and the rest 80% has been used for training the decision tree classifiers.

- We have trained decision trees of depth d = 5, 10, and 15. For each depth, we have trained the number of estimators n = 10, 20, and 30.
- Decision tree feature selection has been made based on *sqrt (total features in dataset).*
- In the core decision tree implementation, it was previously a sequential execution. Given the size of the dataset for this project, it would take a few hours to finish the execution for each decision tree.
- Hence, we have implemented the multiprocessing strategy to make the execution faster. The segment of the algorithm that computes the mutual information at each split of the decision tree has been parallelized. In our systems, the code uses 8 CPU cores to parallelize the task of computing mutual information. All the processes shared a common universal dictionary to which they wrote the computed mutual information.
- This strategy has reduced the overall computing time of decision tree algorithm by 31%.

Gaussian Naive Bayes based Bagging (script: Appendix [22]):

- We have implemented our own Gaussian Naive Bayes Classifier.
- Naive Bayes, as a base classifier with 20% data set aside for testing gives 78% test accuracy.
- We have trained n = 10 classifiers with varying training sample size for a set of each 10 classifiers. Sample sizes given are s = 75%, 80%, 85%, 90%.
- The prediction of class for each example is independent of others. Hence there is no need to execute the prediction sequentially. Instead, we have converted it into parallel execution using a multiprocessing worker pool that manages as many parallel processes as there are cpu cores. In our case, it is 8.

- In order to maintain the one-to-one correspondence between the example and their class label, we are passing the index of each example to the *predict* method. This index is used to correctly update the class label of an example.

sklearn implementations (script: Appendix [13]):

Decision Tree Based Bagging -with Grid Search hyper-parameter tuning

- In attempt to find the best hyper-parameters for the decision tree classifier we performed a Grid-Search along with Bagging with following parameter grid:
    - Base Decision Tree depth: 5, 10, 15, 20, 25,30,40,50
    - Decision Tree max-features selection criteria: sqrt, log2 (since the no. of features are higher, we focused on feature selection criteria)
- We kept the bagging estimators and bagging max features fixed in this grid search.
- Cross Validation: 5-fold cross validation was used to get mean cross validated scores
- Accuracy and AUC-ROC scores were used to find the best hyperparameters
- Using the best hyper-parameters chosen, we performed train v/s test performance analysis on different Bagging estimators and Training Sample sizes

RandomForest -with Grid Search hyper-parameter tuning

- Similar to Decision Tree, followed a Grid Search approach to find the suitable hyperparameters:
- Following Parameters were considered:
    - Max-Depth of tree: 5,10,20,30,40,50,75
    - No. of Trees: 50,100,250,500,1000
    - Max_feature selection criteria: ‘sqrt’ and ‘log2’
- Bootstrap=True was considered
- Similar to Decision Tree, Accuracy and AUC-ROC estimates for each hyperparameter setting was considered to find the best case

- Using the best hyperparameter setting, we did the test v/s train performance analysis using multiple training size and post-pruning parameter (ccp-alpha)

Support Vector Classifier -with Grid Search hyper-parameter tuning

- Like the previous two, we performed Grid Search with following parameters:
    - Kernel: Linear and Poly (since each SVC run is expensive, we tried with these two kernels to observe the output trend)
    - C values: 0.1, 1, 10, 100
- The SVC was bagged as well while performing grid search.
    - No. of bagging estimators: 20
    - Max_features: 0.5
- 5-fold cross validation was considered
- Out of the grid search output, we chose best hyperparameter output based mean accuracy and auc-roc values
- Using the best hyper-parameter setting we did the train v/s performance assessment with various sample size and bagging max-features proportions

Gaussian Naive Bayes based Bagging

- We used Gaussian Naive Bayes (GNB) classifier with bagging, with following parameters:
    - No. of bagging estimators: 10, 20, 30
    - Training sample sizes = 75%, 80%, 85%, 90%
- Here also we evaluated both accuracy and auc-roc score to evaluate best case parameters

OBSERVATIONS & INFERENCES

Own implementations:

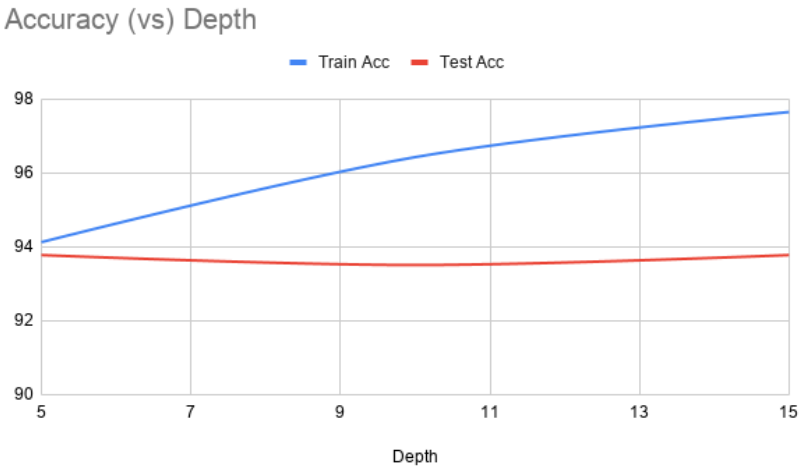
Cosine Similarity

Based on the cosine similarity matching on the entire vectorized data (1234 examples), following results are obtained:

		Matched Label	
		Non-Spam	Spam
Source-Label	Non-Spam	1063	68
	Spam	18	85

- The raw data of matching is available here [14]
- As observed, the accuracy of similarity matching is ~93%, which serves as good benchmark for our further sophisticated learners

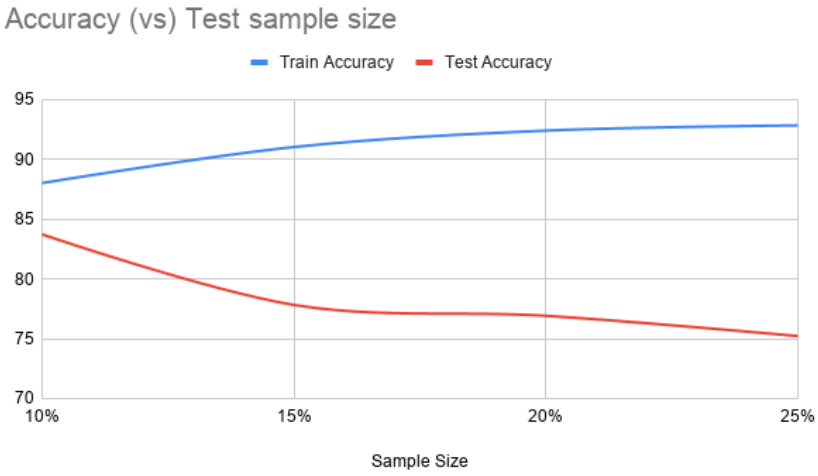
Decision Tree based Bagging (raw data: Appendix [12])

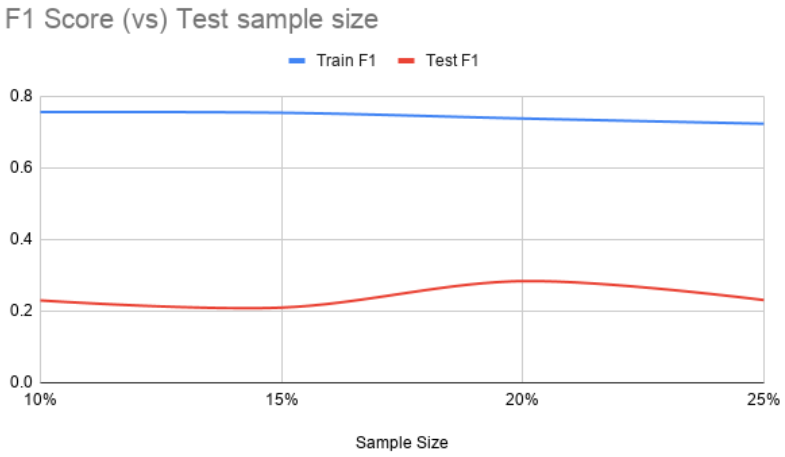


- We can observe that the test set accuracy is very close to the training set accuracy.
- Also, as the depth is increased the training set accuracy is increasing and moving more towards overfitting. But the corresponding test set accuracy is stable across all the depths considered in this case.

Further testing with higher depths needs to be done in order to determine whether the model moves towards overfitting or not. It may also be possible that the test set accuracy will fall with higher depths.

Gaussian Naive Bayes based Bagging (raw data: Appendix [12])



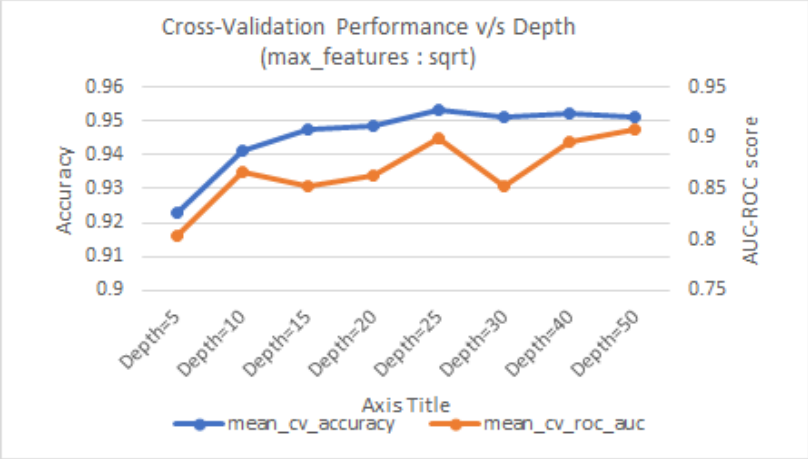


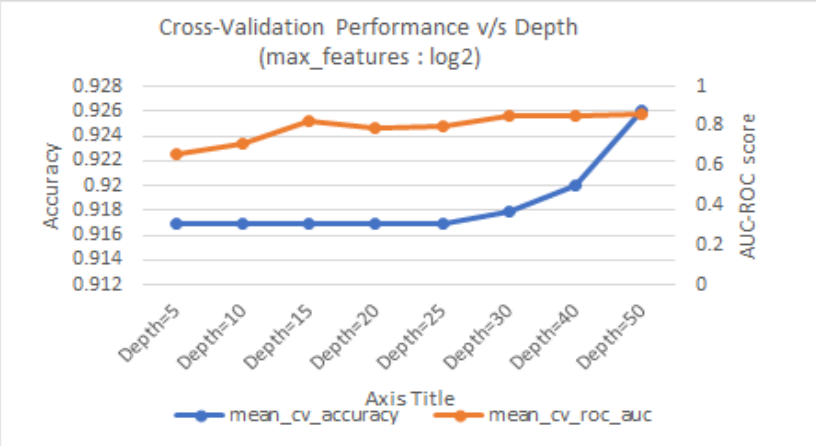
- In the above graphs Accuracy, AUC score, and F1 score have been plotted against the size of the test dataset.
- We varied the size of the test dataset from 10% to 25% of the main dataset, in the steps of 5.
- For each test sample size, we kept the number of estimators constant (n = 10) and trained the model. We can observe that as the test sample size increases, the test accuracy has decreased while the train accuracy has increased. This indicates that we are moving towards overfitting the model if we increased the test sample size.

sklearn implementations:

Decision Tree Based Bagging -with Grid Search hyper-parameter tuning

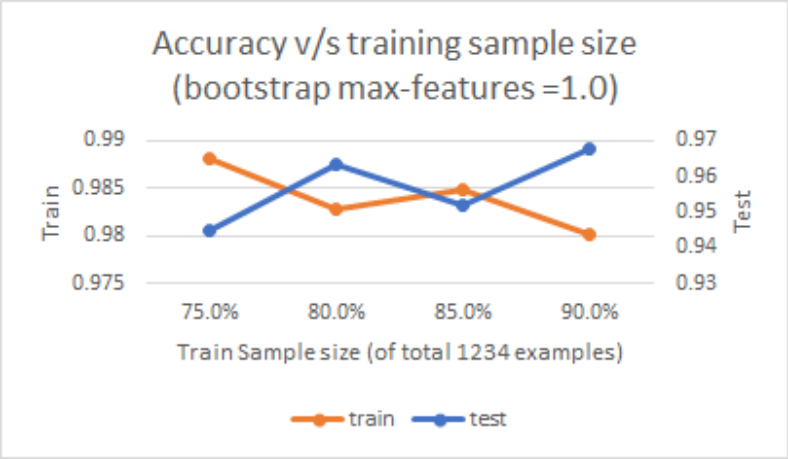
- Based on above mentioned hyper-parameter grid, the following performance trends were observed:

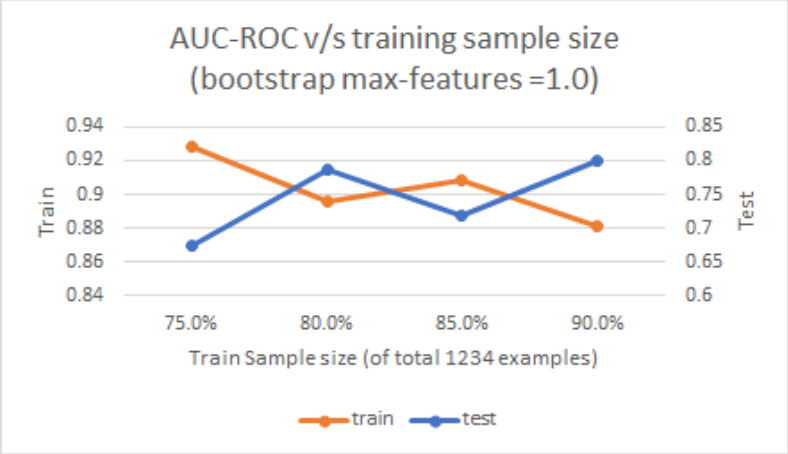


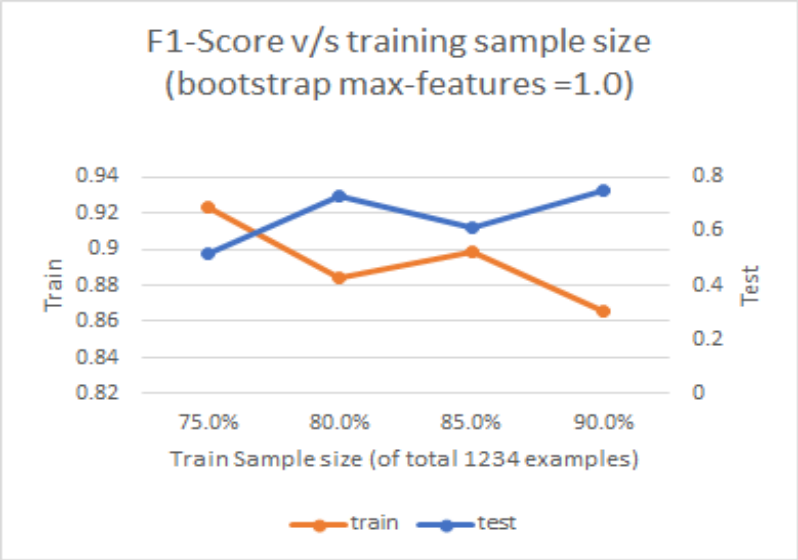


- Based on the accuracy and auc-roc values we can observe the 'sqrt' selection criteria performs better.
- Also, based on the best accuracy and auc-roc values considered together (Appendix [15]), we considered the following parameter for further analysis: max_depth=25, max_features=sqrt
- We further used the above parameters and did a performance assessment on
    - Training Sample size (stratified sampling): 75%, 80%, 85%, 90%

    - Bootstrap max-features (this was not varied in grid-search): 0.5,0.75,0.1
- We evaluated Accuracy, AUC-ROC and F1-score for all the above combinations. For all graphs, refer here (Appendix [16])
- Below are the graphs for the best parameter combination (Max-Depth =25, max-features selection criteria: 'sqrt', bootstrap max-feature proportions=1.0, no. of bootstrap estimators = 20):







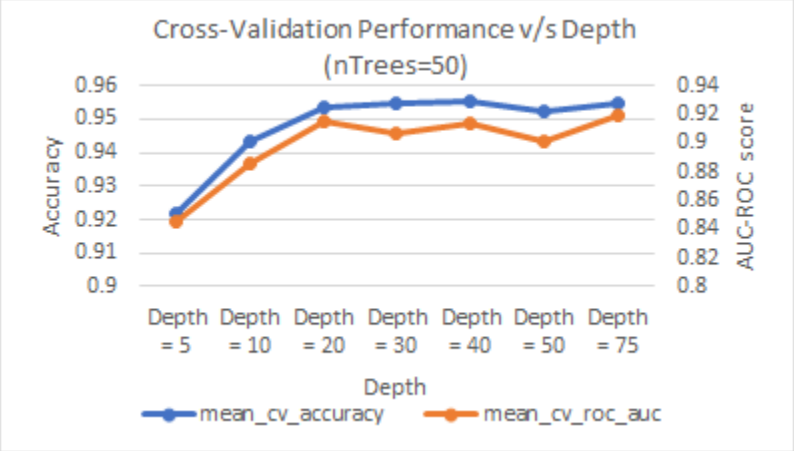
- As observed, the decision tree classifier performs quite well with respect accuracy, auc-roc and f1-score. Following are the confusion matrices for 80% Training sample setting

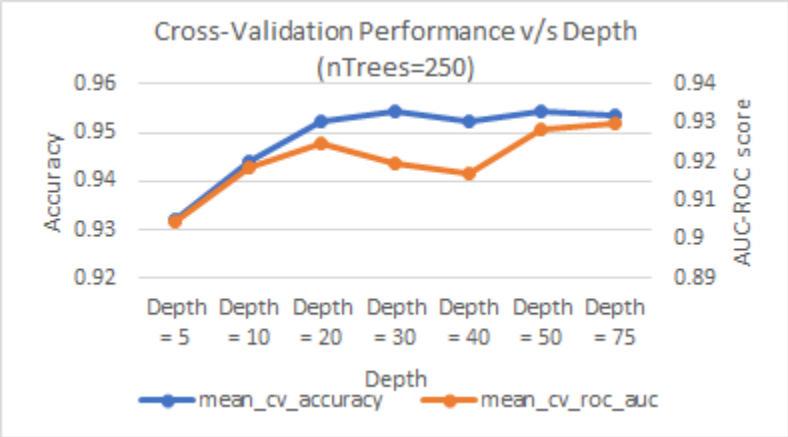
Train (80%)		Predicted	
		0	1
Actual	0	905	0
	1	17	65

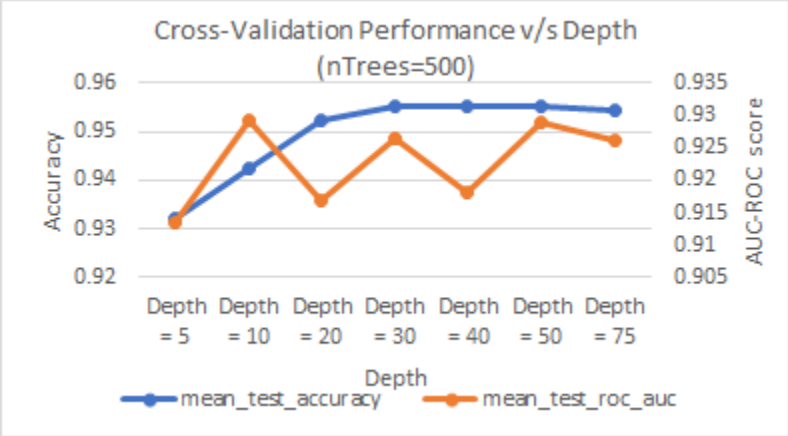
Test (20%)		Predicted	
		0	1
Actual	0	226	0
	1	11	10

RandomForest -with Grid Search hyper-parameter tuning

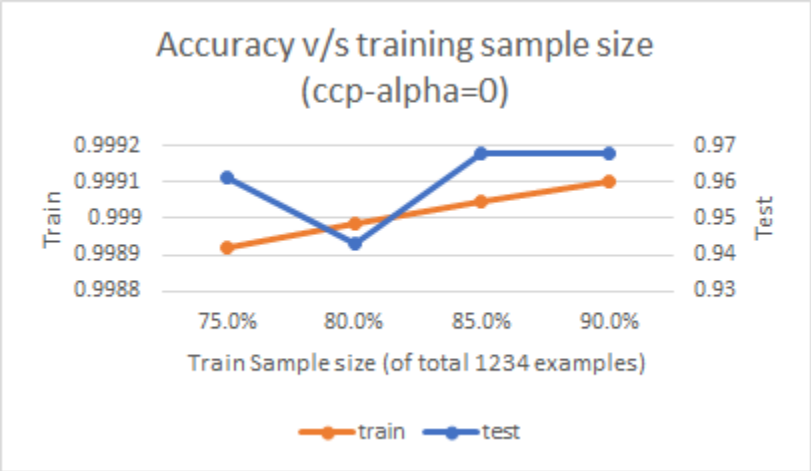
- Based on above mentioned hyper-parameter grid, the following performance trends were observed
- The following trend shown is for the no. of trees=50 ,250 and 500 and max-feature selection criteria='sqrt', corresponding to which best scoring results were obtained in grid search (entire list of graphs - (Appendix [17]))

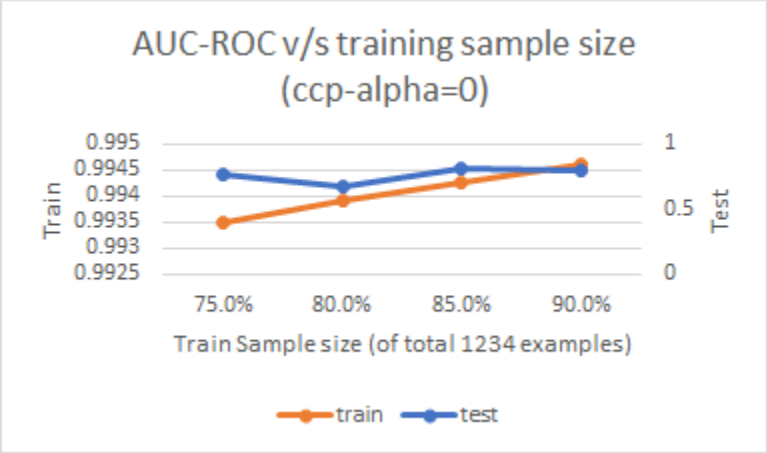


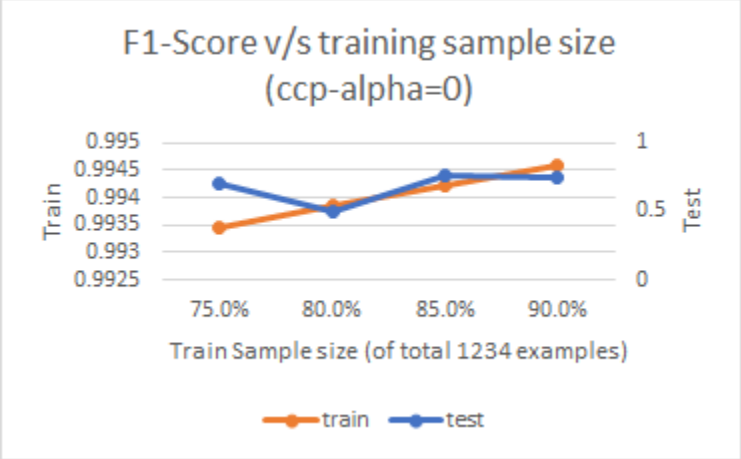




- Based on the grid search, following hyperparameters were chosen for further analysis:
    - max_depth=50,
    - n_estimators=500,
    - max_features='sqrt'
- Using the above parameters, train v/s test performance was assessed based on
    - Post-pruning parameter (ccp-alpha): 0, 0.01, 0.1, 1.0, 10
    - Training Sample size (stratified sampling): 75%, 80%, 85%, 90%
- The entire list of graphs for the above settings are available here (Appendix [18])
- Below are the graphs for best settings observed:





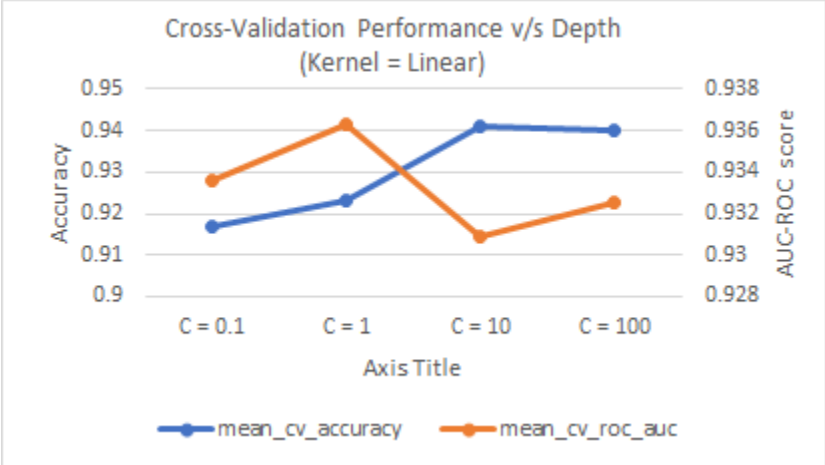


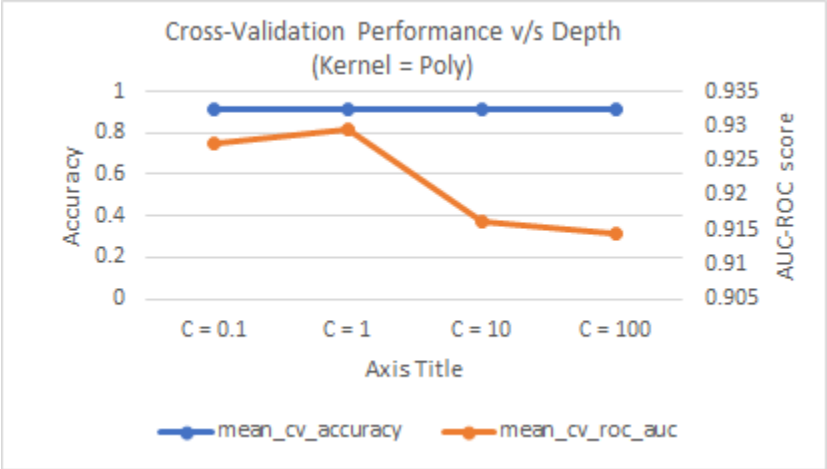
- As we observed, RandomForest also gave excellent results for both test and training samples, comparable to DecisionTree classifier.

- The AUC and F1-score values are better than decision tree on average, signifying better separation power of this classifier between spam and non-spam
- As can be observed in the entire list of graphs (above link provided), the higher the post pruning parameter, the more it penalizes the accuracy, for this problem overfitting is already controlled by searching for max depth and no. of trees in cross validation. Hence, results corresponding to 0 post pruning is reported

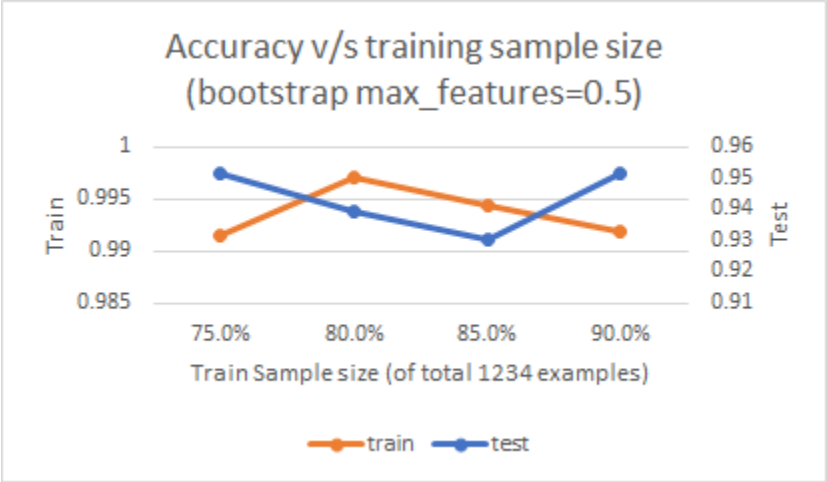
Support Vector Classifier -with Grid Search hyper-parameter tuning

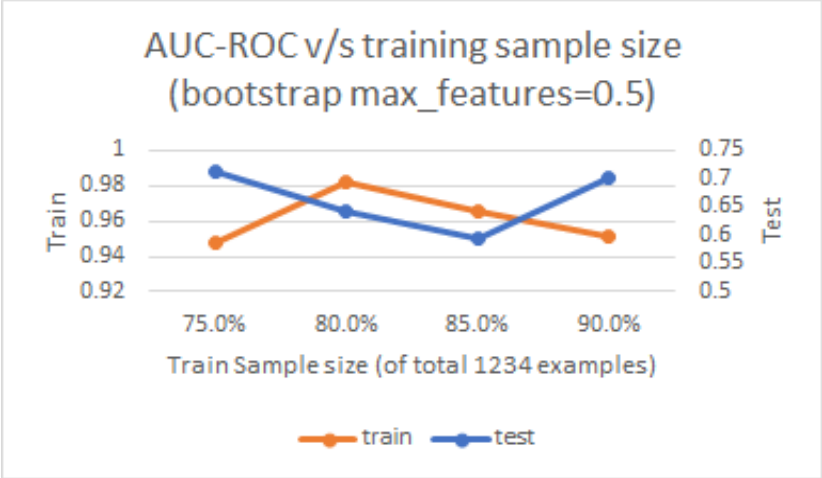
- Based on above mentioned hyper-parameter grid, the following performance trends were observed:
- No. of Bagging Estimators: 20, 5 fold cross validation (mean accuracy and auc-roc reported) .
- The entire output data is available here (Appendix [19])

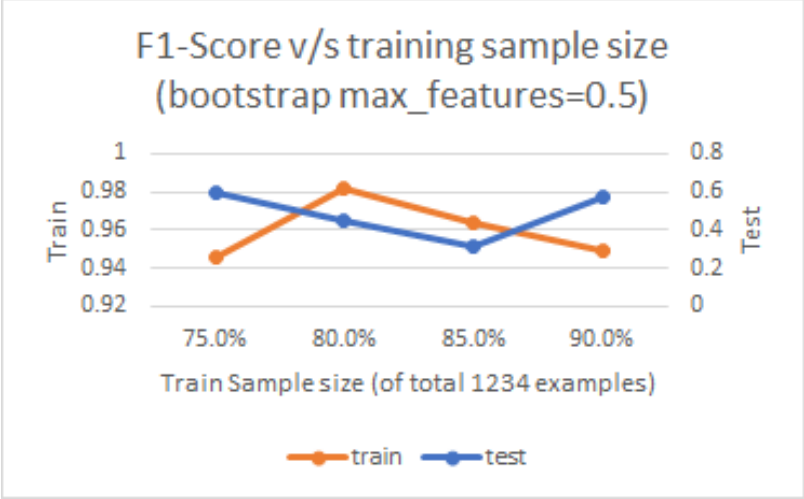




- Better results are observed for linear kernel, hence gaussian kernel was not explored any further
- It might be that the data could be linearly separable based on certain features (keywords) which are specific in case spam calls (‘credit card’, ‘IRS’ , etc. tokens)
- Based on the observed trend, the following hyperparameters were chosen for further analysis
    - C=10
    - kernel=linear
- Using the above parameters, we did test v/s train performance analysis for
    - Bootstrap max_features: 0.5, 0.75, and 1.0
    - Training Sample size (stratified sampling): 75% , 80% , 85% , 90%
- The entire list of charts is available here (Appendix [20]):
- Below are the best-case charts:


Accuracy v/s training sample size
(bootstrap max_features=0.5)
Train
1
0.995
0.99
0.985
Test
0.96
0.95
0.94
0.93
0.92
0.91
75.0%
80.0%
85.0%
90.0%
Train Sample size (of total 1234 examples)
train
test

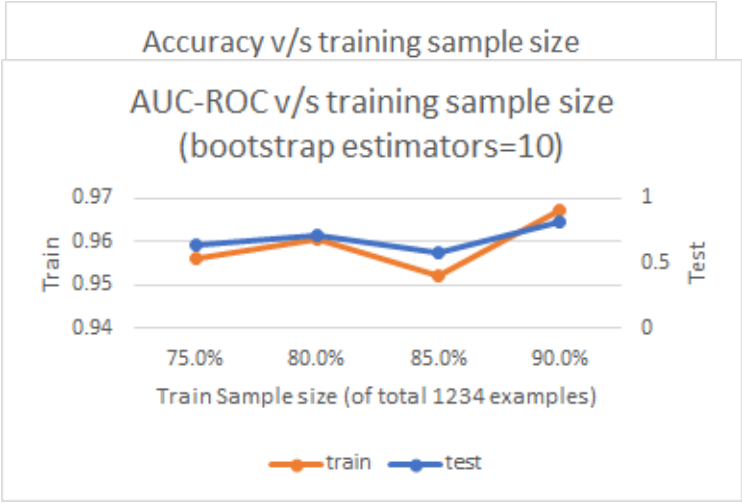

AUC-ROC v/s training sample size
(bootstrap max_features=0.5)
Train
1
0.98
0.96
0.94
0.92
Test
0.75
0.7
0.65
0.6
0.55
0.5
75.0%
80.0%
85.0%
90.0%
Train Sample size (of total 1234 examples)
train
test


F1-Score v/s training sample size
(bootstrap max_features=0.5)
Train
1
0.98
0.96
0.94
0.92
Test
0.8
0.6
0.4
0.2
0
75.0%
80.0%
85.0%
90.0%
Train Sample size (of total 1234 examples)
train
test

- Based on the observations, the support vector classifier also does reasonably well in classifying the data.
- The test accuracy is comparable to the tree-based classifiers; however, the auc-roc and f1-score values shows lesser separation power of linear SVC as compared to the complex axis-parallel decision boundaries of tree based classifiers

Gaussian Naive Bayes based Bagging

- Entire list of graphs available here (Appendix [21])
- Below are the trends of the for best case parameters:



- In GNB classifications we observed quite poor results in terms of test accuracy and auc-roc values.

FURTHER WORK

On top of the present analysis, we wish to would expand our investigation on the following dimensions:

- Explore audio quality features like loudness, frequency, etc. to assess the tonality of a speech and use that to augment context classification (can help predict tone of anger, humor, etc.)
- Expand the present database on larger scale to include more noise (make spam calls a low event rate case) and apply learners to come up with more resilient classification models

APPENDIX

1. http://www.openslr.org/12/ : LibriSpeech
2. http://www.robots.ox.ac.uk/~vgg/data/voxceleb/ :VoxCeleb
3. https://freesound.org/search/?q=spam&f=&s=score+desc&advanced=0&g=1 :FreeSound
4. Youtube Playlist of Lenny Spam Calls
5. Podcast episodes of normal non-spam conversations
6. Automated Transcription Script : 1.Multithread_transcription.py (submitted)
7. Processed audio files in google cloud storage :

   https://console.cloud.google.com/storage/browser/spam-recog-src (all .wav files)

   https://console.cloud.google.com/storage/browser/audio-sr (some more .wav files considered)
8. Raw transcripts file: transcripts_master.csv (submitted in 'data' folder)
9. Tf-Idf data (converted to pickle (.pkl) format for faster load into python workspace)

https://console.cloud.google.com/storage/browser/_details/spam-recog-src/tfidf_master.pkl?project=speech-recog-1587108253310

10. Count vectorized data (converted to pickle (.pkl) format for faster load into python workspace)

    https://storage.cloud.google.com/audio-sr/vectorized-data-master.pkl?folder&organizationId

11. Submitted file : 2.Text_Preprocessing.py in code folder

12. Raw data used for plotting graphs (for own implementations) :

    https://docs.google.com/spreadsheets/d/1Jn0oH8iUZX3YsijKgxTWglIltHpXHT6S3XoWPEGXjBA/edit#gid=0

13. Sklearn implementations: submitted script: 3.sklearn-implementations.py

14. Cosine similarity Raw Data: submitted data: cosine_similarity.csv

15. Submitted: output_DT.xlsx

16. Submitted: DT.xlsx

17. Submitted: output_RF.xlsx

18. Submitted: RF.xlsx

19. Submitted: output_SVC.xlsx

20. Submitted: SVC.xlsx

21. Submitted: GNB.xlsx

22. Submitted: 5.GNB-own implementation.py

23. Submitted: 4. Ensembles_Dtree.py

24. https://truecaller.blog/2019/12/03/truecaller-insights-top-20-countries-affected-by-spam-calls-sms-in-2019/