Sapienza Universita di Roma

Master in Artificial Intelligence and Robotics

Machine Learning

Homework 1: Malware Analysis – Drebin Dataset

Cecilia Aponte P. - Matricola: 1822225

1. Abstract:

Malware is software introduced by attackers into electronics with the intent of malicious activity. Nowadays, these attacks are prominent, varied, and evolved from the simpler methods used previously. This means that malware is ever harder to determine for its elimination. Malware developers produce variants to often confuse and evade securities used. These variants are from the same malware are malware families. These use similar behaviors, exploits the same vulnerabilities, and the same malicious objectives.

This is where Malware analysis is teamed with Machine Learning to solve the issue of identification of software the contains or not a virus. Using either supervised or unsupervised learning, a system can identify malware. In this task, supervised learning was employed using the DREBIN dataset that has annotated samples of both samples that contain and do not contain a virus of attack are driven towards mobile Android users. The system learned from the samples given using various algorithms to mark future predictions as Positive Virus (‘1’) or Negative Virus (‘0’). Using a Linear SVC Algorithm resulted in an accuracy of 97.2% and an F-1 score of 0.976 of both negative and positive instances.

2. Introduction:

With an ever-growing quantity of technology that can interface with applications and software for download, the more malicious activities are created and executed for the manipulations of these technologies and their system or owner’s personal data. Although there are growing quantities and varieties of malware which are installed into technology, there is also creation of more sophisticated and precise malware detection systems.

Machine Learning is used to exploit the prior knowledge of information from samples that contain and do not contain a virus to learn the trends and detect malware in advance. One common attack is towards Android mobile users, who without knowing download applications and documents that contain malware. Even though the Android platform requires consent from the user for the application to use, most users blindly grant the permission without noticing this request could be common to malware application packages.

Due to this, malware detection is necessary before downloading and granting such permissions. For the Machine Learning Algorithm to learn the key is to not only provide samples of both positive and negative instances, but also to choose the right features that describe the malware. The samples therefore must contain annotations of all of those chosen relevant features. To extract these features, the Android Operating system permits to extract some types of features that cannot be extracted in other environments. This file contained in the package of an application is called the manifest.xml.

The DREBIN dataset performs a lightweight static analysis and extracts the important chosen features from the manifest file. These features include S1:

Hardware components, S2: Requested permissions, S3: App components, S4: Filtered intents, S5: Restricted API calls, S6: Used permissions, S7: Suspicious API calls, and S8: Network addresses. These features are embedded in a vector space containing all the samples. The Machine Learning Algorithm then learns this vector space to analyze the similitudes and differences of samples that contain and do not contain virus. The algorithm then finds a model that describes this separation. In this case, the model is a linear model meaning one straight line is enough to separate the Malicious samples (‘1’) with the Benign samples (‘0’). This model runs very fast and is therefore a great implementation for detection of any future malware in android applications.

3. Dataset:

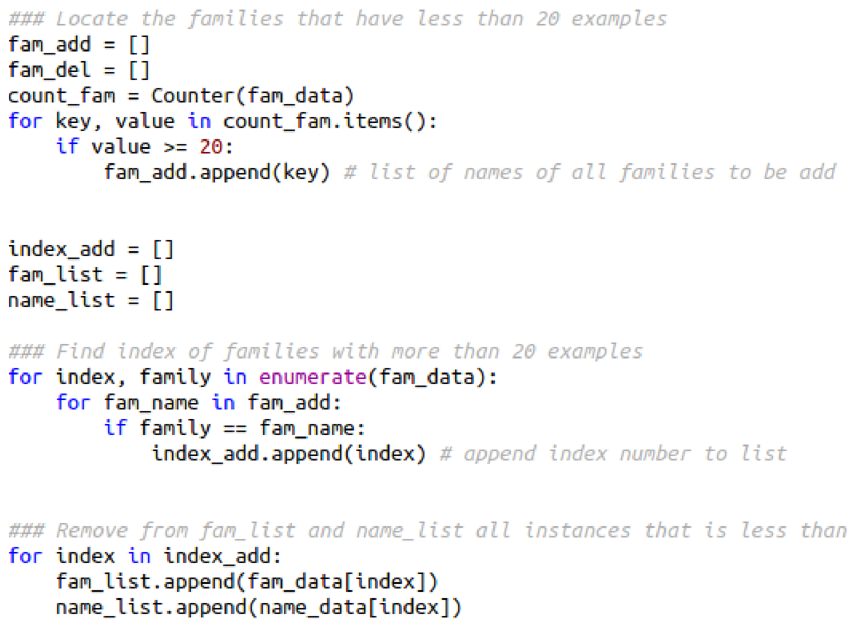
The dataset used to build the model is DREBIN. It performs a broad static analysis to the android application packages’ manifest and gathers the features. It gathers 123,453 benign samples and 5,560 malicious samples. Their implementation also offers an explanation to the users of the components that were identified to be malicious.

The features are extracted as the following sets: S1 which corresponds to the requested hardware components (camera, touchscreen, GPS, etc.); S2 which corresponds to the requested permissions (send sms, etc.); S3 which corresponds to app components and includes activities, services, contest providers, and broadcast receivers; S4 which is the filtered intents (boot_completed, etc.); S4 which is the restricted API calls finding calls for which the required permissions were not requested; S6 which is the used permissions showing the behavior of the application; S7 which is the suspicious API calls referring to those that allow access to sensitive data or resources (getDeviceId, setWifiEnabled, sendTextMessage, etc.); and lastly S8 which is the network addresses as urls.

Malicious activity is usually a combination of these features. A vector space is created with the features (totaling 545,000 different features) that are encountered (set as ‘1’ in the matrix containing all the features). These features are then analyzed to learn and detect the malicious samples.

4. Method:

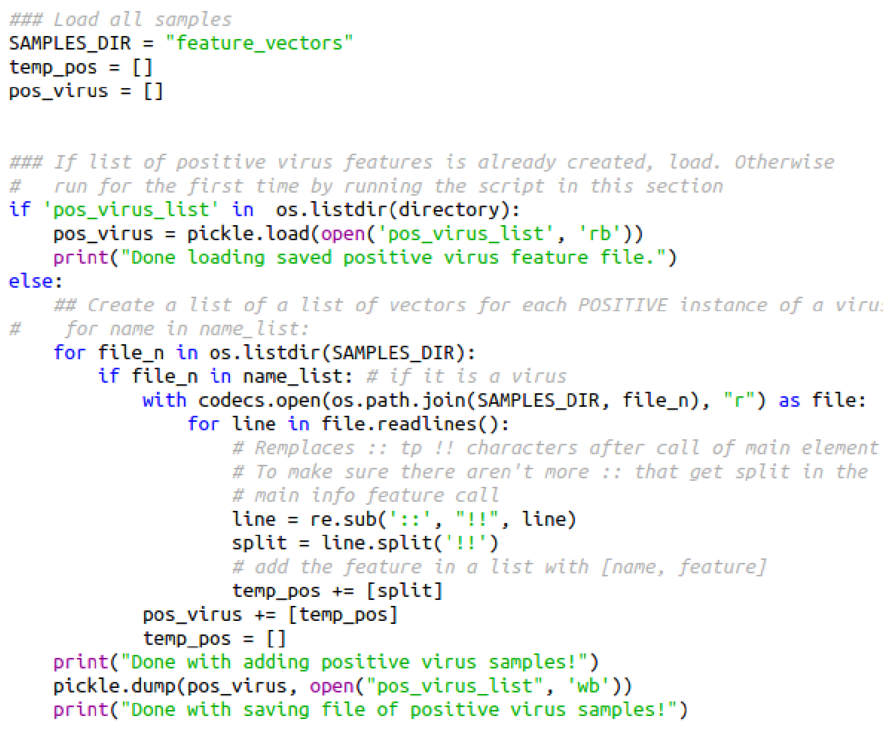
The first step taken to implement DREBIN into an algorithm is to construct the vector space with all the features. To do so, only certain malware families were considered. The families of malware should contain at least 20 samples so that enough data is provided for the algorithm to learn well the characteristics. The method used to do so is shown in *Figure 1*. First, the families are counted to see which contain more than 20 samples. Then, the index of the families that contain more than 20 samples are saved and appended to a new list to save the names of those samples.



```
### Locate the families that have less than 20 examples
fam_add = []
fam_del = []
count_fam = Counter(fam_data)
for key, value in count_fam.items():
    if value >= 20:
        fam_add.append(key) # list of names of all families to be add


index_add = []
fam_list = []
name_list = []

### Find index of families with more than 20 examples
for index, family in enumerate(fam_data):
    for fam_name in fam_add:
        if family == fam_name:
            index_add.append(index) # append index number to list


### Remove from fam_list and name_list all instances that is less than
for index in index_add:
    fam_list.append(fam_data[index])
    name_list.append(name_data[index])
```

Figure 1: Script to locate and remove the families that have less than 20 samples.

Doing this found 175 families that contained less than 20 samples. This consisted of 775 samples in total. Therefore, from a total of 5,560 initial malicious samples only 4,785 samples remained.

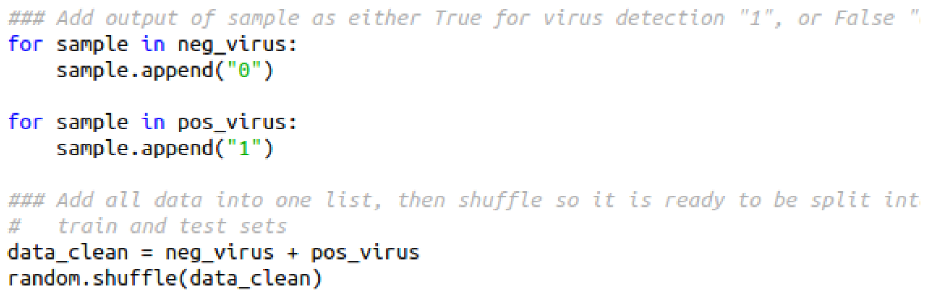


```
### Load all samples
SAMPLES_DIR = "feature_vectors"
temp_pos = []
pos_virus = []


### If list of positive virus features is already created, load. Otherwise
#   run for the first time by running the script in this section
if 'pos_virus_list' in  os.listdir(directory):
    pos_virus = pickle.load(open('pos_virus_list', 'rb'))
    print("Done loading saved positive virus feature file.")
else:
    ## Create a list of a list of vectors for each POSITIVE instance of a viru
#     for name in name_list:
    for file_n in os.listdir(SAMPLES_DIR):
        if file_n in name_list: # if it is a virus
            with codecs.open(os.path.join(SAMPLES_DIR, file_n), "r") as file:
                for line in file.readlines():
                    # Remplaces :: tp !! characters after call of main element
                    # To make sure there aren't more :: that get split in the
                    # main info feature call
                    line = re.sub('::', "!!", line)
                    split = line.split('!!')
                    # add the feature in a list with [name, feature]
                    temp_pos += [split]
            pos_virus += [temp_pos]
            temp_pos = []
    print("Done with adding positive virus samples!")
    pickle.dump(pos_virus, open("pos_virus_list", 'wb'))
    print("Done with saving file of positive virus samples!")
```

Figure 2: Script to get all the features for the malicious samples.

The next step was to extract the features of all the samples. The process as shown in *Figure 2* is to replace '::' from the string that contains all the features with the set: {name_set::features} to '!!' since some portions of certain strings contain also '::' within the features; this could otherwise create future issues. It then splits the string of one sample at this location to create a list: [name_set, feature] and then a list of all these features together. This is repeated for each sample, therefore created a list of 4,785 malicious samples and 123,453 benign samples each with lists of lists containing all the features.

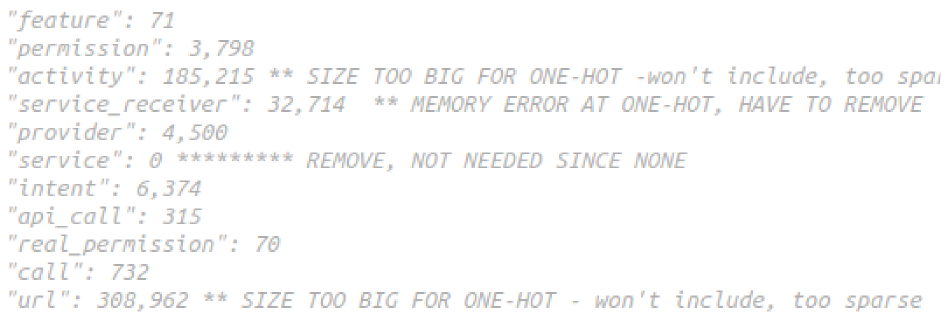
Next step was to clean the data, as seen in *Figure 3*. To do so, every sample was appended the correct value corresponding to their output, meaning malicious '1' or benign '0'. The two different type of samples were then added into one list and shuffled so they would be ready to be later split into training or test sets.



```
### Add output of sample as either True for virus detection "1", or False "
for sample in neg_virus:
    sample.append("0")

for sample in pos_virus:
    sample.append("1")

### Add all data into one list, then shuffle so it is ready to be split int
#   train and test sets
data_clean = neg_virus + pos_virus
random.shuffle(data_clean)
```

Figure 3: Script that adds its output value to each sample, then adds all the samples into one list that is later shuffled.

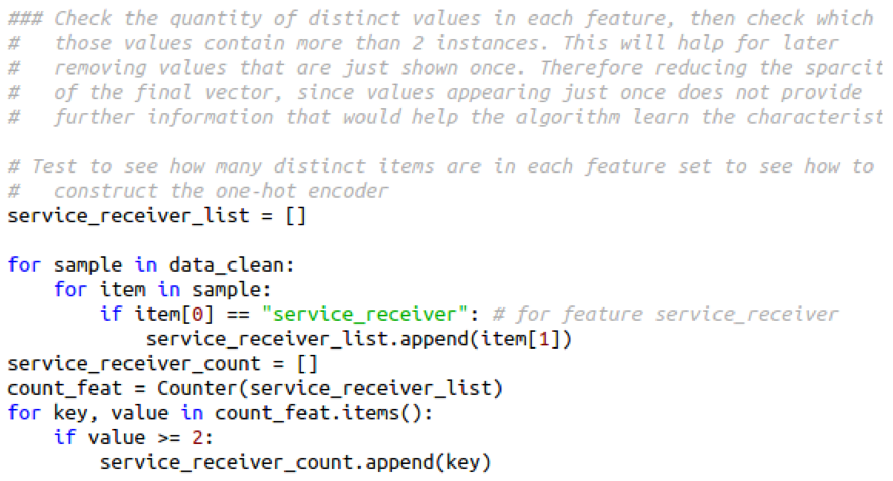
Initially, due to memory restrictions three of the feature sets were not included into my vector. While constructing the separate feature set as a dataframe that would later be binarized, my system gave memory errors whenever the amount of values would be greater than 5,000. Therefore, I studied each feature set to see which ones would be included into my vector. As seen in in *Figure 4*, the features that were initially removed were 'activity', 'service_receiver', and 'url'. To minimize the memory issue, features that did not occur more



```
"feature": 71
"permission": 3,798
"activity": 185,215 ** SIZE TOO BIG FOR ONE-HOT -won't include, too spa
"service_receiver": 32,714  ** MEMORY ERROR AT ONE-HOT, HAVE TO REMOVE
"provider": 4,500
"service": 0 ********* REMOVE, NOT NEEDED SINCE NONE
"intent": 6,374
"api_call": 315
"real_permission": 70
"call": 732
"url": 308,962 ** SIZE TOO BIG FOR ONE-HOT - won't include, too sparse
```

Figure 4: Information of the quantity of non-repeated values for each feature set. Due to memory error, some of these features were initially removed.

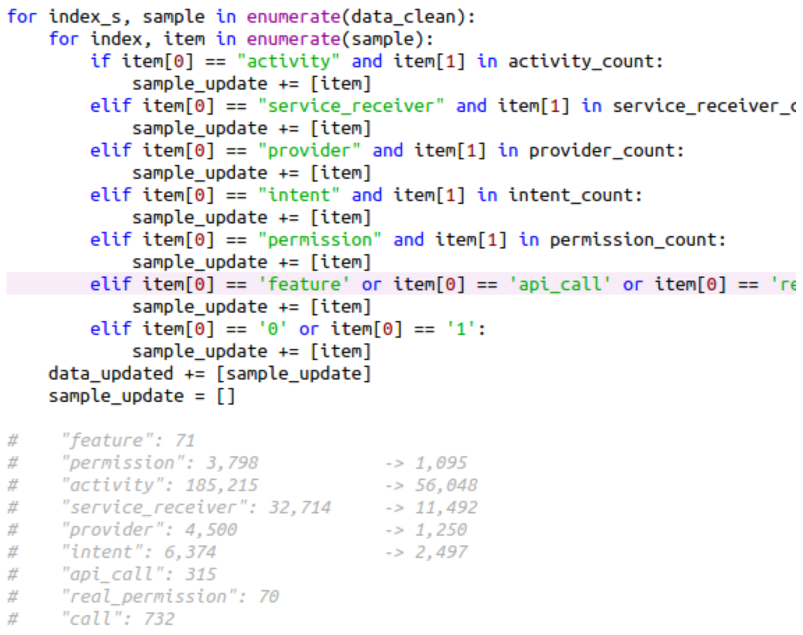
than once was removed. This would not only help with the memory, but also reduce sparsity of the final vector. This would increase the speed of the calculation and would not remove any special information since features that appear just once do not provide further information that would help the algorithm. As shown in *Figure 5*, this was executed by counting the amount times each feature appeared and then appending to a list all features that occur more than once. This process was repeated for some of the features that contained many features, namely: ‘permission’, ‘activity’, ‘service_receiver’, ‘provider’, and ‘intent’. The feature ‘url’ was completely removed since it contained 300k features and was analyzed that it would probably not contain information that could determine the learning of malware.



```
### Check the quantity of distinct values in each feature, then check which
#   those values contain more than 2 instances. This will halp for later
#   removing values that are just shown once. Therefore reducing the sparcit
#   of the final vector, since values appearing just once does not provide
#   further information that would help the algorithm learn the characterist

# Test to see how many distinct items are in each feature set to see how to
#   construct the one-hot encoder
service_receiver_list = []

for sample in data_clean:
    for item in sample:
        if item[0] == "service_receiver": # for feature service_receiver
            service_receiver_list.append(item[1])
service_receiver_count = []
count_feat = Counter(service_receiver_list)
for key, value in count_feat.items():
    if value >= 2:
        service_receiver_count.append(key)
```

Figure 5: Check for unique features for each feature set. Features that would occur more than once were added to a list for later use.

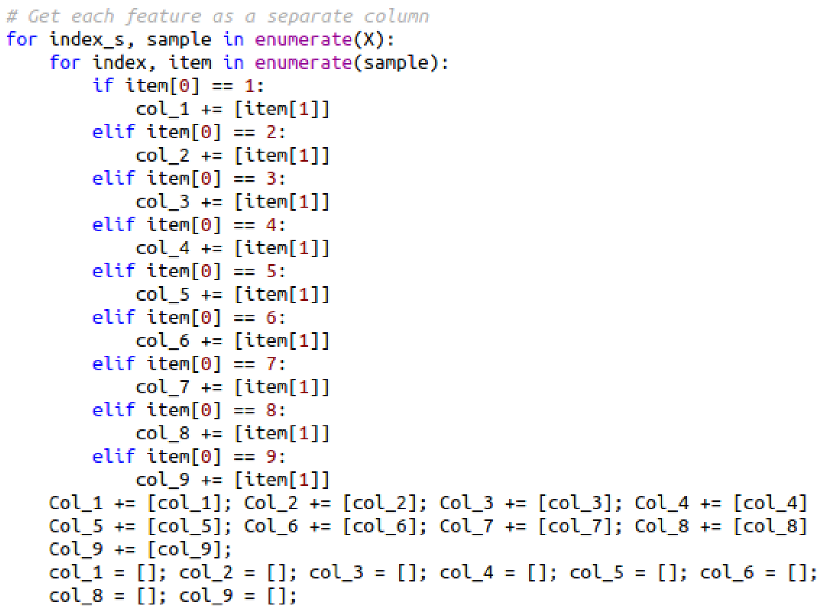
With this information, the data is cleaned as shown in *Figure 6* by removing features that include just one instance, and the feature set ‘url’. The result of the number of features in each feature set is shown below the script.



```
for index_s, sample in enumerate(data_clean):
    for index, item in enumerate(sample):
        if item[0] == "activity" and item[1] in activity_count:
            sample_update += [item]
        elif item[0] == "service_receiver" and item[1] in service_receiver_c
            sample_update += [item]
        elif item[0] == "provider" and item[1] in provider_count:
            sample_update += [item]
        elif item[0] == "intent" and item[1] in intent_count:
            sample_update += [item]
        elif item[0] == "permission" and item[1] in permission_count:
            sample_update += [item]
        elif item[0] == 'feature' or item[0] == 'api_call' or item[0] == 're
            sample_update += [item]
        elif item[0] == '0' or item[0] == '1':
            sample_update += [item]
    data_updated += [sample_update]
    sample_update = []

#    "feature": 71
#    "permission": 3,798              -> 1,095
#    "activity": 185,215              -> 56,048
#    "service_receiver": 32,714       -> 11,492
#    "provider": 4,500                -> 1,250
#    "intent": 6,374                  -> 2,497
#    "api_call": 315
#    "real_permission": 70
#    "call": 732
```

Figure 6: Script to clean data of features that occurs less than twice and 'url' feature set. The resulting number of features for each feature set is shown below.

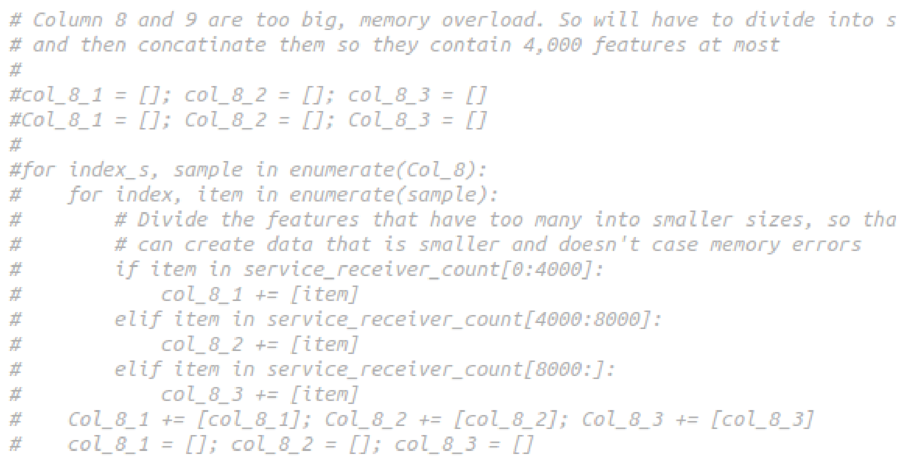
The next step was to split the data into X and y, referring to the feature and output (0 or 1) respectively. *Figure 7* shows the process to this by appending to a new list all the features and to another list only the last element which included the value of the output.



```
# Get each feature as a separate column
for index_s, sample in enumerate(X):
    for index, item in enumerate(sample):
        if item[0] == 1:
            col_1 += [item[1]]
        elif item[0] == 2:
            col_2 += [item[1]]
        elif item[0] == 3:
            col_3 += [item[1]]
        elif item[0] == 4:
            col_4 += [item[1]]
        elif item[0] == 5:
            col_5 += [item[1]]
        elif item[0] == 6:
            col_6 += [item[1]]
        elif item[0] == 7:
            col_7 += [item[1]]
        elif item[0] == 8:
            col_8 += [item[1]]
        elif item[0] == 9:
            col_9 += [item[1]]
    Col_1 += [col_1]; Col_2 += [col_2]; Col_3 += [col_3]; Col_4 += [col_4]
    Col_5 += [col_5]; Col_6 += [col_6]; Col_7 += [col_7]; Col_8 += [col_8]
    Col_9 += [col_9];
    col_1 = []; col_2 = []; col_3 = []; col_4 = []; col_5 = []; col_6 = [];
    col_8 = []; col_9 = [];
```

Figure 7: Script to separate each feature in an array, with each feature having a column.

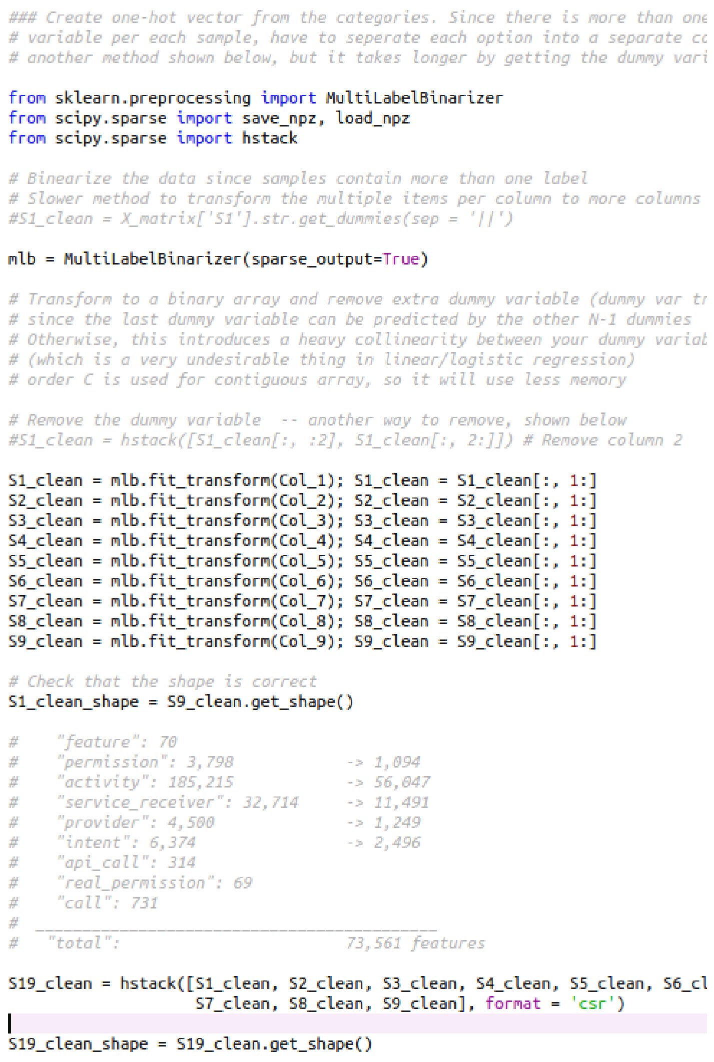
e a one-hot vector of the features for each sample. Since there was a memory issue to deal with, I expanded the features S1-S7 to S1-S9 so to distribute the data into separate bins that could be saved and later loaded to create the entire vector. The names of the feature sets were replaced with their appropriate number of 1-9. Each feature number was then separated from the dataset to be set one column each, since previously the data was set in a list and an array is needed to run the algorithm. This process is shown in *Figure 8*, as a loop of each sample and within each sample each feature number. For each number, the item was appended to its appropriate column number.



```
# Column 8 and 9 are too big, memory overload. So will have to divide into s
# and then concatinate them so they contain 4,000 features at most
#
#col_8_1 = []; col_8_2 = []; col_8_3 = []
#Col_8_1 = []; Col_8_2 = []; Col_8_3 = []
#
#for index_s, sample in enumerate(Col_8):
#    for index, item in enumerate(sample):
#        # Divide the features that have too many into smaller sizes, so tha
#        # can create data that is smaller and doesn't case memory errors
#        if item in service_receiver_count[0:4000]:
#            col_8_1 += [item]
#        elif item in service_receiver_count[4000:8000]:
#            col_8_2 += [item]
#        elif item in service_receiver_count[8000:]:
#            col_8_3 += [item]
#    Col_8_1 += [col_8_1]; Col_8_2 += [col_8_2]; Col_8_3 += [col_8_3]
#    col_8_1 = []; col_8_2 = []; col_8_3 = []
```

Figure 8: Script of implementation to resolve memory issue, that is no longer used since it was resolved otherwise.

Due to the initial memory issue, another step was previously taken that was now removed. This was to split feature 8 and 9 into smaller arrays that could handle the memory. See *Figure 9* that shows this implementation for feature 8, but that was commented out since it was not used.

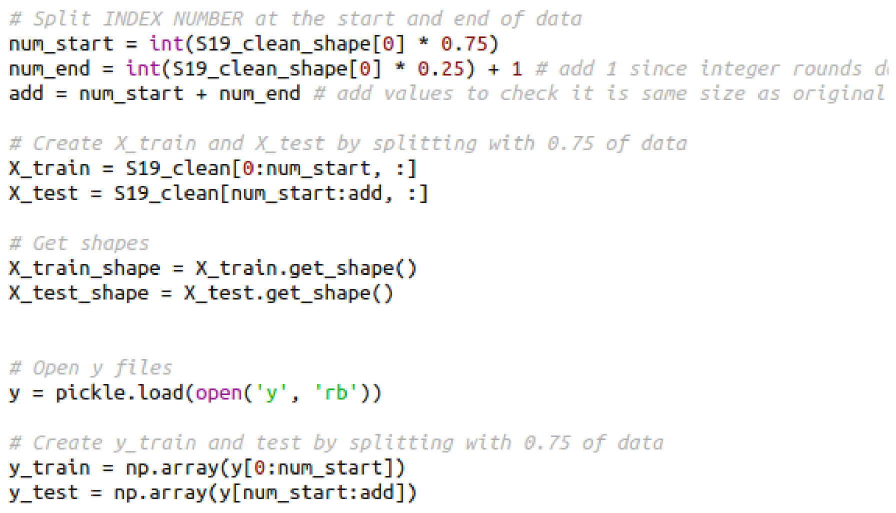


```
### Create one-hot vector from the categories. Since there is more than one
# variable per each sample, have to seperate each option into a separate cc
# another method shown below, but it takes longer by getting the dummy vari

from sklearn.preprocessing import MultiLabelBinarizer
from scipy.sparse import save_npz, load_npz
from scipy.sparse import hstack

# Binearize the data since samples contain more than one label
# Slower method to transform the multiple items per column to more columns
#S1_clean = X_matrix['S1'].str.get_dummies(sep = '||')

mlb = MultiLabelBinarizer(sparse_output=True)

# Transform to a binary array and remove extra dummy variable (dummy var tr
# since the last dummy variable can be predicted by the other N-1 dummies
# Otherwise, this introduces a heavy collinearity between your dummy variat
# (which is a very undesirable thing in linear/logistic regression)
# order C is used for contiguous array, so it will use less memory

# Remove the dummy variable  -- another way to remove, shown below
#S1_clean = hstack([S1_clean[:, :2], S1_clean[:, 2:]]) # Remove column 2

S1_clean = mlb.fit_transform(Col_1); S1_clean = S1_clean[:, 1:]
S2_clean = mlb.fit_transform(Col_2); S2_clean = S2_clean[:, 1:]
S3_clean = mlb.fit_transform(Col_3); S3_clean = S3_clean[:, 1:]
S4_clean = mlb.fit_transform(Col_4); S4_clean = S4_clean[:, 1:]
S5_clean = mlb.fit_transform(Col_5); S5_clean = S5_clean[:, 1:]
S6_clean = mlb.fit_transform(Col_6); S6_clean = S6_clean[:, 1:]
S7_clean = mlb.fit_transform(Col_7); S7_clean = S7_clean[:, 1:]
S8_clean = mlb.fit_transform(Col_8); S8_clean = S8_clean[:, 1:]
S9_clean = mlb.fit_transform(Col_9); S9_clean = S9_clean[:, 1:]

# Check that the shape is correct
S1_clean_shape = S9_clean.get_shape()

#    "feature": 70
#    "permission": 3,798            -> 1,094
#    "activity": 185,215            -> 56,047
#    "service_receiver": 32,714     -> 11,491
#    "provider": 4,500              -> 1,249
#    "intent": 6,374                -> 2,496
#    "api_call": 314
#    "real_permission": 69
#    "call": 731
#  ________________________________________
#   "total":                        73,561 features

S19_clean = hstack([S1_clean, S2_clean, S3_clean, S4_clean, S5_clean, S6_cl
                    S7_clean, S8_clean, S9_clean], format = 'csr')

S19_clean_shape = S19_clean.get_shape()
```

Figure 9: Script to create a vector of binarized features

To create the one-hot vector of the features, each feature set should be added its separate column so that each sample could contain a 0 or 1 to show that it does not or does, respectively, include that specific feature. The issue with this process was that each sample contains more than one instance per feature set, meaning a normal label encoding and one-hot encoding would not be enough. Therefore, the MultiLabelBinarizer (MLB) was used from the Sklearn Preprocessing library to do the task. Since for many of the feature sets there are thousands of features, this process created a big burden in the memory. Creating multiple ~130k by ~100k vectors arrays damped this process. As explained earlier, multiple options were tried to solve this memory issue. However, it was only when the output of this MLB was set to a sparse matrix did this memory error was eliminated.

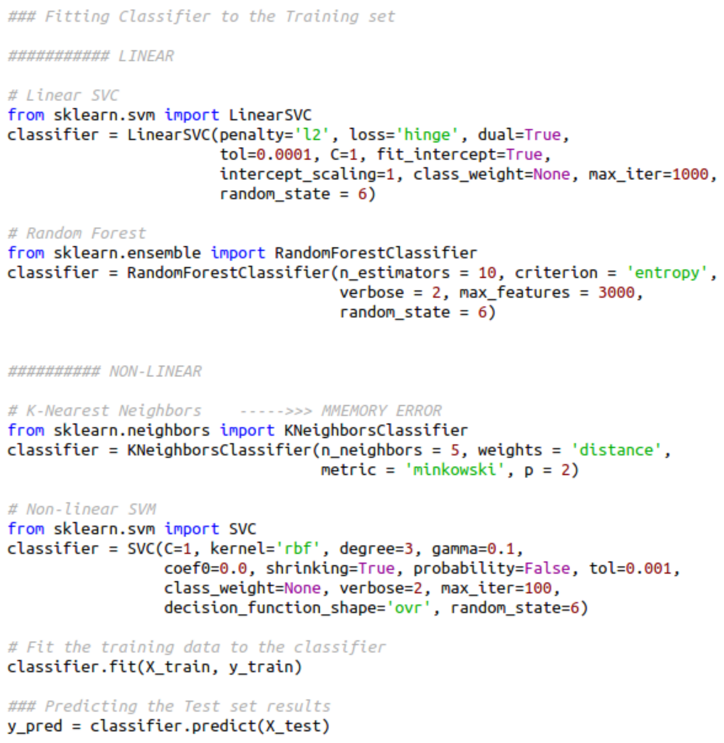
Figure 10 shows this implementation done for each column feature set. From the resulting clean feature set, one column was removed which is the extra dummy variable and avoid the dummy variable trap. The removal of this extra dummy variable is important because otherwise it introduces a heavy collinearity between the dummy variables, which is very undesirable for this linear implementation. The shape of each of these new arrays were checked to be the size that is expected (N-1) as seen in the bottom of *Figure 10*. All these column feature sets were then stacked horizontally with hstack which does so for sparse matrices and output another sparse matrix of type CSR. The shape of the resulting feature vector for all the data is 128,238 x 73,561 (total samples x total features).



```
# Split INDEX NUMBER at the start and end of data
num_start = int(S19_clean_shape[0] * 0.75)
num_end = int(S19_clean_shape[0] * 0.25) + 1 # add 1 since integer rounds d
add = num_start + num_end # add values to check it is same size as original

# Create X_train and X_test by splitting with 0.75 of data
X_train = S19_clean[0:num_start, :]
X_test = S19_clean[num_start:add, :]

# Get shapes
X_train_shape = X_train.get_shape()
X_test_shape = X_test.get_shape()


# Open y files
y = pickle.load(open('y', 'rb'))

# Create y_train and test by splitting with 0.75 of data
y_train = np.array(y[0:num_start])
y_test = np.array(y[num_start:add])
```

Figure 10: Script to split the data between train and test sets

The last step in the method to create the clean data that would be inputted into the algorithms is the split of the dataset into testing and training. The training set was given 75% of the data, which results in 96,178 samples with the same 73,561 features. The test set which is used to check the accuracy and other performance parameters of the trained model contained the rest of the data set 25% which corresponds to 32,060 samples with the same 73,561 features. See *Figure 11* for this implementation.



```
### Fitting Classifier to the Training set

########### LINEAR

# Linear SVC
from sklearn.svm import LinearSVC
classifier = LinearSVC(penalty='l2', loss='hinge', dual=True,
                       tol=0.0001, C=1, fit_intercept=True,
                       intercept_scaling=1, class_weight=None, max_iter=1000,
                       random_state = 6)

# Random Forest
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy',
                                    verbose = 2, max_features = 3000,
                                    random_state = 6)


########## NON-LINEAR

# K-Nearest Neighbors    ----->>> MMEMORY ERROR
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 5, weights = 'distance',
                                  metric = 'minkowski', p = 2)

# Non-linear SVM
from sklearn.svm import SVC
classifier = SVC(C=1, kernel='rbf', degree=3, gamma=0.1,
                 coef0=0.0, shrinking=True, probability=False, tol=0.001,
                 class_weight=None, verbose=2, max_iter=100,
                 decision_function_shape='ovr', random_state=6)

# Fit the training data to the classifier
classifier.fit(X_train, y_train)

### Predicting the Test set results
y_pred = classifier.predict(X_test)
```

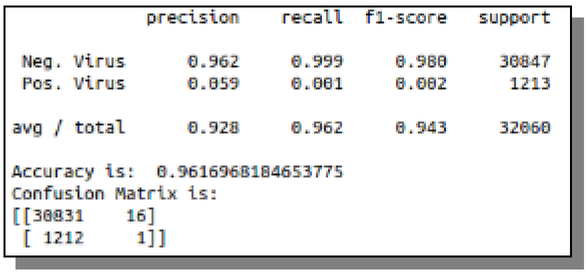
Figure 11: Script of the implementation of Linear SVC, Random Forest, and Kernel SVM algorithms to the dataset

<u>5. Algorithm:</u>

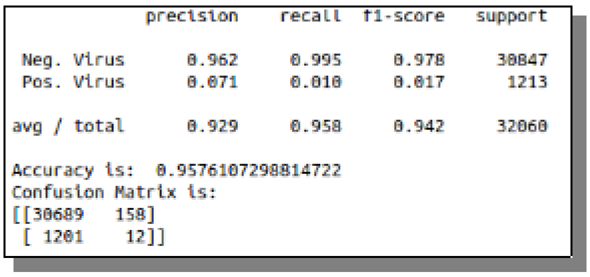
Three different algorithms were implemented to the dataset. All the algorithms implemented were part of the Sklearn library. For a linear classification, the LinearSVC and Random Forest were implemented. Instead to test if the dataset was not linear, a non-linear algorithm was tested which was the Kernel SVM. Note that the K-Nearest Neighbor was also tested, however due to memory constraints the algorithm could not finalize running. See *Figure 12* to see the scripts of these algorithms.

The first of the classification algorithms implemented was the Linear SVC from the Support Vector Machine family (it uses a simpler method that runs faster than the regular SVC that has any kernel including the linear kernel. This algorithm finds the Maximum Margin Hyperplane which is the best decision boundary that will separate space into classes. This line is drawn equidistant to the 'support vectors' and has to be maximized. It uses the items that are the most like construct the algorithm.

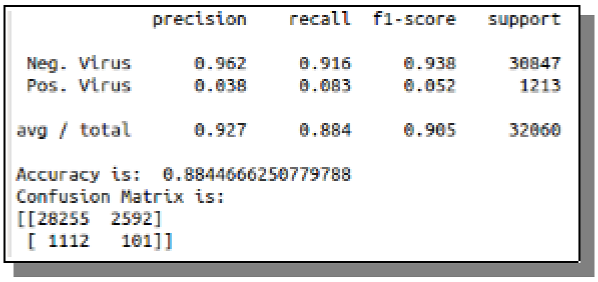
The other classification algorithm that was tested was the non-linear SVM. This was used mainly as a test, since initially the results from the linear algorithms were not given good results. The Kernel SVM works similar to the linear SVC but uses a kernel system. These kernels are used as the decision boundary to decide



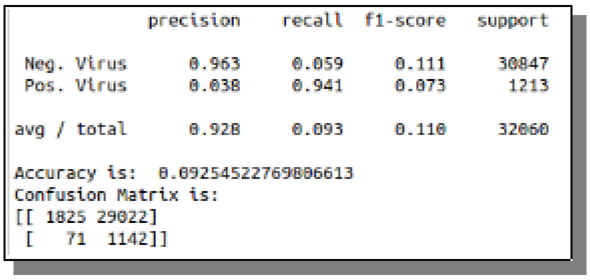
```
              precision    recall  f1-score   support

 Neg. Virus       0.962     0.999     0.980     30847
 Pos. Virus       0.059     0.001     0.002      1213

avg / total       0.928     0.962     0.943     32060

Accuracy is:  0.9616968184653775
Confusion Matrix is:
[[30831     16]
 [ 1212      1]]
```



```
              precision    recall  f1-score   support

 Neg. Virus       0.962     0.995     0.978     30847
 Pos. Virus       0.071     0.010     0.017      1213

avg / total       0.929     0.958     0.942     32060

Accuracy is:  0.9576107298814722
Confusion Matrix is:
[[30689    158]
 [ 1201     12]]
```



```
              precision    recall  f1-score   support

 Neg. Virus       0.962     0.916     0.938     30847
 Pos. Virus       0.038     0.083     0.052      1213

avg / total       0.927     0.884     0.905     32060

Accuracy is:  0.8844666250779788
Confusion Matrix is:
[[28255  2592]
 [ 1112   101]]
```



```
              precision    recall  f1-score   support

 Neg. Virus       0.963     0.059     0.111     30847
 Pos. Virus       0.038     0.941     0.073      1213

avg / total       0.928     0.093     0.110     32060

Accuracy is:  0.09254522769806613
Confusion Matrix is:
[[ 1825 29022]
 [   71  1142]]
```

Figure 12: Results of various algorithms using less features due to memory constraints. Top Left: Linear SVC; Top Right: Random Forest; Bottom Left: Linear SVC plus Bagging Classifier; Bottom Right: Naive Bayes

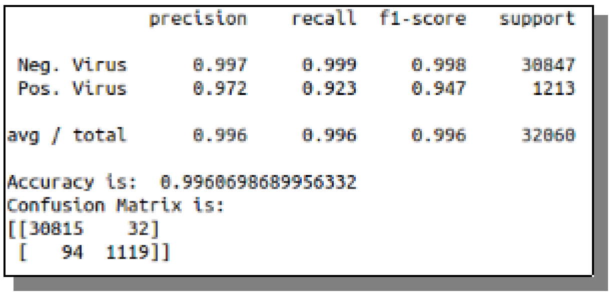
what is set in one or another class. It can use different kernels such as RBF, Sigmoid, and Polynomial. For example, the RBF kernel uses a normal gaussian to make its decisions; if the distance between a point to the landmark is large, the kernel is close to zero. Otherwise if it is close, the kernel converges to 1. This decision 1 or 0 determines on which side the sample lies.

The last classification algorithm that was implemented was the Random Forest. This algorithm takes multiple decision tress to make one single more powerful algorithm. It works by taking several random trees to predict the output value of the samples. This sample is then assigned the average across all predicted output values.

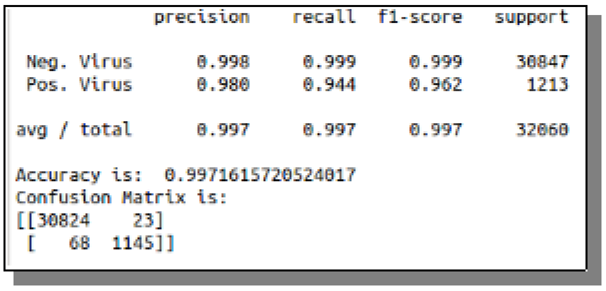
<u>6. Initial Results:</u>

Initially due to the memory issue, the feature vector only included: "feature", "permission", "provider", "intent", "api_call", "real_permission", "call". When running various algorithms, the results were very poor. *Figure 13* shows the performance of the various algorithms. The benign samples were classified for the most part correctly, but the problem was that even with every combination of parameters the malicious samples were not being classified correctly.

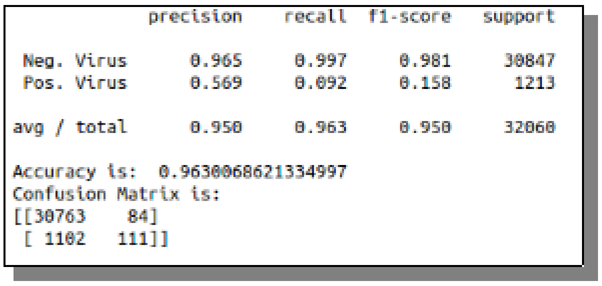
Since no combination of parameters or algorithms gave good results, further possibilities were investigated. After re-reading the original DREBIN paper, it was concluded that important features were left out. These were ‘service_receiver’ and ‘activity’. Both of these were added back to the feature vector and with the use of



```
             precision    recall  f1-score   support

 Neg. Virus       0.997     0.999     0.998     30847
 Pos. Virus       0.972     0.923     0.947      1213

avg / total       0.996     0.996     0.996     32060

Accuracy is:  0.9960698689956332
Confusion Matrix is:
[[30815    32]
 [   94  1119]]
```

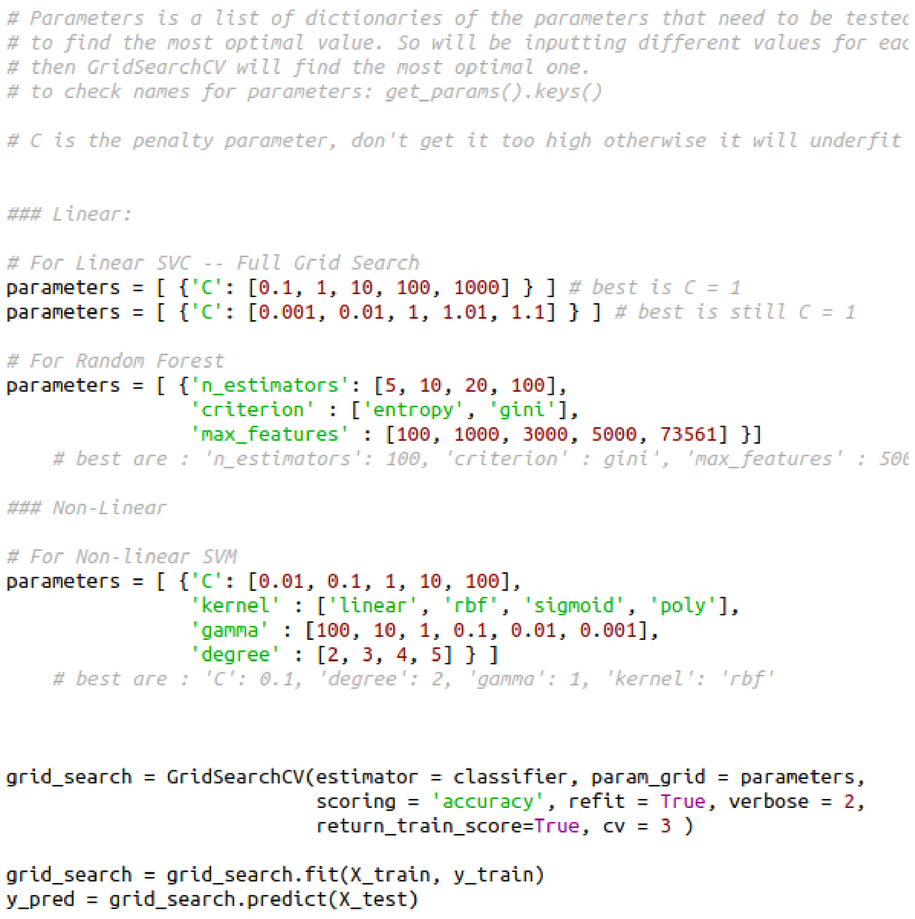


```
             precision    recall  f1-score   support

 Neg. Virus       0.998     0.999     0.999     30847
 Pos. Virus       0.980     0.944     0.962      1213

avg / total       0.997     0.997     0.997     32060

Accuracy is:  0.9971615720524017
Confusion Matrix is:
[[30824    23]
 [   68  1145]]
```



```
             precision    recall  f1-score   support

 Neg. Virus       0.965     0.997     0.981     30847
 Pos. Virus       0.569     0.092     0.158      1213

avg / total       0.950     0.963     0.950     32060

Accuracy is:  0.9630068621334997
Confusion Matrix is:
[[30763    84]
 [ 1102   111]]
```

Figure 14: Results of algorithms after included all but one ('url') features. Top Left: Linear SVC; Top Right: Random Forest; Bottom: Kernel SVM



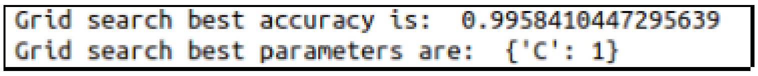
```
# Parameters is a list of dictionaries of the parameters that need to be testec
# to find the most optimal value. So will be inputting different values for eac
# then GridSearchCV will find the most optimal one.
# to check names for parameters: get_params().keys()

# C is the penalty parameter, don't get it too high otherwise it will underfit


### Linear:

# For Linear SVC -- Full Grid Search
parameters = [ {'C': [0.1, 1, 10, 100, 1000] } ] # best is C = 1
parameters = [ {'C': [0.001, 0.01, 1, 1.01, 1.1] } ] # best is still C = 1

# For Random Forest
parameters = [ {'n_estimators': [5, 10, 20, 100],
                'criterion' : ['entropy', 'gini'],
                'max_features' : [100, 1000, 3000, 5000, 73561] }]
    # best are : 'n_estimators': 100, 'criterion' : gini', 'max_features' : 50(

### Non-Linear

# For Non-linear SVM
parameters = [ {'C': [0.01, 0.1, 1, 10, 100],
                'kernel' : ['linear', 'rbf', 'sigmoid', 'poly'],
                'gamma' : [100, 10, 1, 0.1, 0.01, 0.001],
                'degree' : [2, 3, 4, 5] } ]
    # best are : 'C': 0.1, 'degree': 2, 'gamma': 1, 'kernel': 'rbf'



grid_search = GridSearchCV(estimator = classifier, param_grid = parameters,
                           scoring = 'accuracy', refit = True, verbose = 2,
                           return_train_score=True, cv = 3 )

grid_search = grid_search.fit(X_train, y_train)
y_pred = grid_search.predict(X_test)
```

Figure 13: Script to do a Grid Search to find the most optimal parameters for the Linear SVC, Random Forest, and Kernel SVM algorithms, respectively. Accuracy is used as the scoring parameter for the decisions.

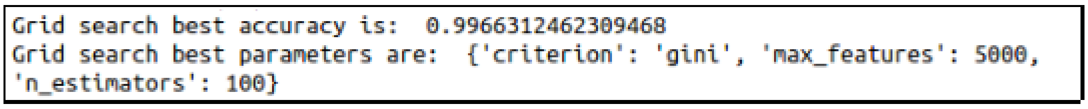
sparse matrices, the memory constraint stop being a problem. *Figure 14* shows the results of these changes using those algorithms.

7. Parameter Optimization:

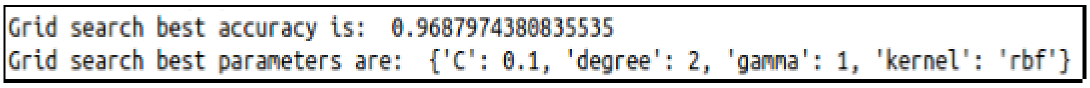
To find the best parameters for the algorithms above, a Grid Search was implemented. As shown in *Figure 15* to implement this, certain set of parameters have to be checked through each algorithm. Each set of options from each possible parameter is checked through the algorithm three times to confirm that those results are the most optimal.



```
Grid search best accuracy is:  0.9958410447295639
Grid search best parameters are:  {'C': 1}
```



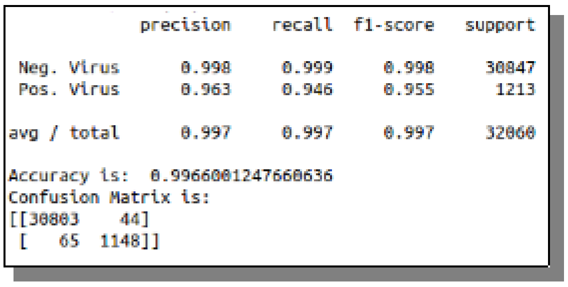
```
Grid search best accuracy is:  0.9966312462309468
Grid search best parameters are:  {'criterion': 'gini', 'max_features': 5000,
'n_estimators': 100}
```



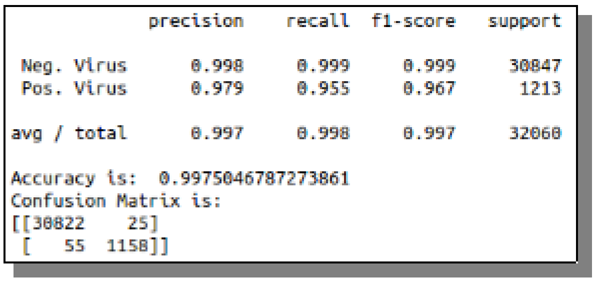
```
Grid search best accuracy is:  0.9687974380835535
Grid search best parameters are:  {'C': 0.1, 'degree': 2, 'gamma': 1, 'kernel': 'rbf'}
```

Figure 15: Results of running Grid Search. Top: Linear SVC; Middle: Random Forest; Bottom: Kernel SVM.

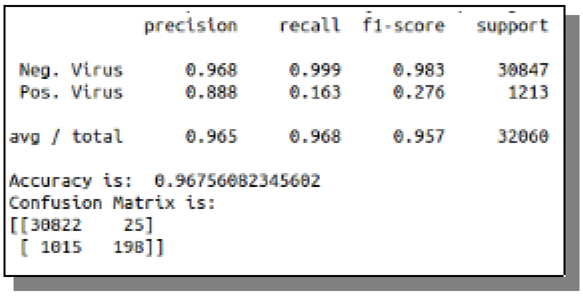
The results from the Grid Search is shown on *Figure 16*. With the best parameters for Linear SVC to be C=1, for Random Forest to be criterion = gini, max_features = 5,000, and n_estimators = 100, and lastly for Kernel SVM to be C = 0.1, kernel = rbf, and gamma = 1.



```
             precision    recall  f1-score   support

 Neg. Virus      0.998     0.999     0.998     30847
 Pos. Virus      0.963     0.946     0.955      1213

avg / total      0.997     0.997     0.997     32060

Accuracy is:  0.9966001247660636
Confusion Matrix is:
[[30803    44]
 [   65  1148]]
```



```
             precision    recall  f1-score   support

 Neg. Virus      0.998     0.999     0.999     30847
 Pos. Virus      0.979     0.955     0.967      1213

avg / total      0.997     0.998     0.997     32060

Accuracy is:  0.9975046787273861
Confusion Matrix is:
[[30822    25]
 [   55  1158]]
```

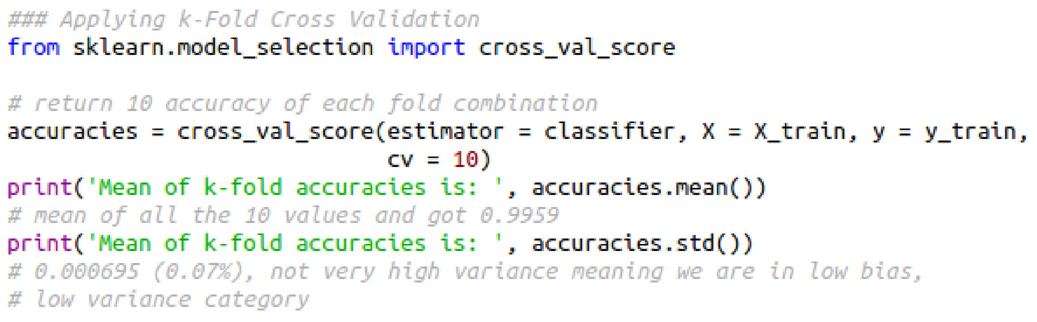


```
             precision    recall  f1-score   support

 Neg. Virus      0.968     0.999     0.983     30847
 Pos. Virus      0.888     0.163     0.276      1213

avg / total      0.965     0.968     0.957     32060

Accuracy is:  0.96756082345602
Confusion Matrix is:
[[30822    25]
 [ 1015   198]]
```

Figure 16: Results after Grid Search of the most optimal parameters based on accuracy. Top Left: Linear SVC; Top Right: Random Forest; Bottom: Kernel SVM

8. Final Results:

The results for the Grid Search are shown in *Figure 17*. Even though Random Forest has a better result for accuracy and F1 score, the algorithm takes 7 minutes to run compared to 2 seconds for the Linear SVC. During the prediction of the test, however, the system can classify in a few seconds for both. This means that for this implementation for Android users that need to scan applications as fast as possible, either Linear SVC or Random Forest would work with very high results.



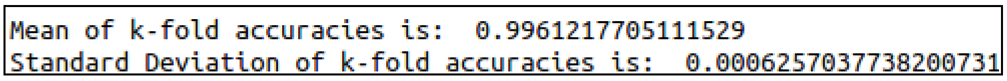
```python
### Applying k-Fold Cross Validation
from sklearn.model_selection import cross_val_score

# return 10 accuracy of each fold combination
accuracies = cross_val_score(estimator = classifier, X = X_train, y = y_train,
                             cv = 10)
print('Mean of k-fold accuracies is: ', accuracies.mean())
# mean of all the 10 values and got 0.9959
print('Mean of k-fold accuracies is: ', accuracies.std())
# 0.000695 (0.07%), not very high variance meaning we are in low bias,
# low variance category
```

Figure 17: Script to run the k-fold cross validation on the model.

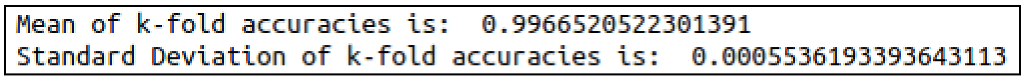
On the other hand, the Kernel SVM did not perform as well as the other two algorithms even with optimal parameters. This means that the classification of this data is linear. For this algorithm, the recall of the malicious samples was very low with 0.16 meaning that it found too many false negatives (the system did not detect the malware). The F-1 score is therefore also low with 0.27, since it looks at the weighted average if precision and recall. This is therefore not the best algorithm for this use.

On the other hand, the results for Linear SVC has better results under all measurements. Precision Recall and F-1 Score for both benign and malicious samples show a measurement of 0.94 or higher. The F-1 score is also almost 1, meaning that the weighted average is proportional and therefore the model can be used to predict future samples with great metrics.

A validation of this results was also performed using k-fold validation. This is used to make sure that the trained model is balanced in variance and bias. That it is not favoring the trained model by underfitting (high bias) or favoring too much the model by overfitting (high variance). In order to check that the model is not under or overfitting, a validation set is needed. The k-fold cross validation splits the training set into k-number of folds. The system is then trained on k-1 of those folds and then tested on the last k fold. *Figure 18* shows the script that runs this evaluation of the model.



```
Mean of k-fold accuracies is:  0.9961217705111529
Standard Deviation of k-fold accuracies is:  0.0006257037738200731
```



```
Mean of k-fold accuracies is:  0.9966520522301391
Standard Deviation of k-fold accuracies is:  0.0005536193393643113
```

Figure 18: Results from k-cross validation of Top: Linear SVC and Bottom: Random Forest

The result of the k-fold cross validation for the Linear SVC can be seen on *Figure 19.* The mean of the accuracies for the 10 folds is 0.996 and the standard deviation is 0.06%. Meaning that the model is at a low bias and low variance category, which is what we want. The same pattern in seen in the Random Forest k-fold validation.

It is important to note that the two features that were initially removed 'service_receiver' and 'activity' have demonstrated to be some of the most important ones since adding them in the model gave the most favorable results. The final features added to the vector space for training the model were therefore the following: "feature", "permission", "provider", "intent", "api_call", "real_permission", "call", "service_receiver", "activity".

The conclusion of this implementation is that the fastest and most favorable model is the Linear SVC with a weighted accuracy of 97.2% and a weighted F-1 score of 0.976. The next model that reproduces similar results but runs much slower in comparison is the Random Forest with a weighted accuracy of 97.7% and a F-1 score of 0.983. Due to the success of its metrics and the speed of the algorithm, the Linear SVC ends up being the best algorithm from the ones tested in this implementation.