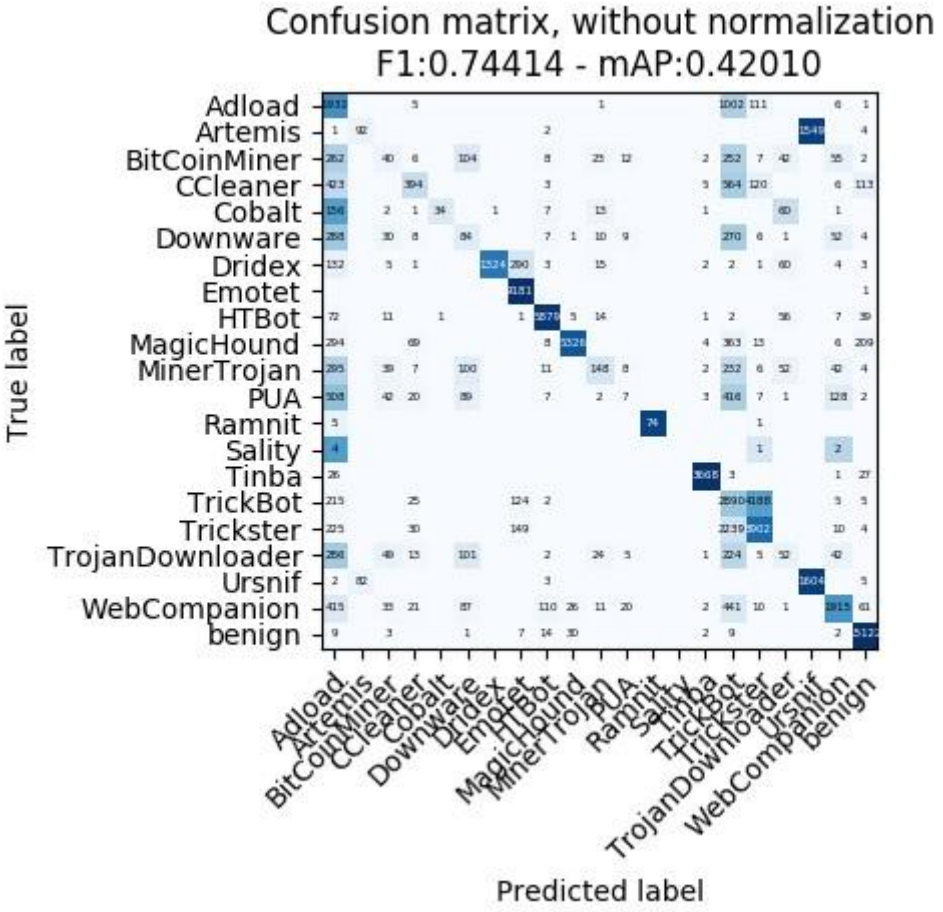
NetML Competition - Task 2 Team IANDJ

Itay Meiri and Jessica Fliker

We started off exploring the data and the confusion matrix of the baseline results of all challenges. The process described below was similar to all attempts on every dataset.

Consider the fine-grained confusion matrix of the NetML dataset:



From the data above, we can tell that the model confuses TrickBot and Trickster classification, as well as Adload and TrickBot. Furthermore, there is a large amount of error when classifying Artemis and Ursnif.

In order to improve the model, we have attempted different approaches, that were unsuccessful:

1) We have attempted to train TrickBot and Trickster classification independently
2) We have attempted to train Adload and Trickster/TrickBot classification independently

```
df_a['label'] = ytrain # label the data # separate # #
trickster from trickbot
dfa = df_a[ (df_a['label'] == 15) | (df_a['label'] == 16)]
```

This yielded no significant improvement for all classifications.
Instead, we attempted to cross-reference the data between Adload/Trickster classification and Adload/Trickbot and classification, which was much better.

We did by manually combining the results from the original model, with the ones we trained.

For instance the following:

```
# atr Adload/Trickster, atb Adload/TrickBot, sub TrickBot/Trickster
for i in res:
   if res[i] == 'Adload':
       if atr[i] != 'Adload' and atb[i] != 'Adload':
           res[i] = sub[i]

# Both atr, atb models think that the classification isn't an Adload
# So instead, we change it to Trickster/Trickbot based on the sub model
```

In the sample code above, we iterate over the results of the original submission data and replace the old classification with the new one, in an attempt to improve the overall accuracy.

This process yielded very small improvements, and in some cases lowered the rate from the base model. We have noticed that the manual cross reference was too lenient in changing data, which led to an *overcorrection.* Furthermore, the classifier for TrickBot/Trickster wasn't good enough to distinguish between Trickbot and Trickster.

In order to make the model more robust, we have decided to use multiple classifiers and cross references all of them.

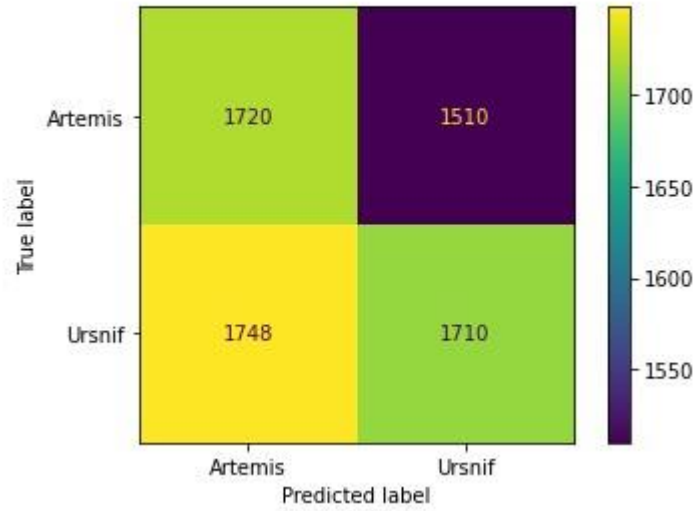
```
# atr Adload/Trickster, atb Adload/TrickBot, sub TrickBot/Trickster
# we have 3 difference models for Adload/TrickBot/Trickster
# Random Forest, Decision Tree, Extra Tree
for i in res:
   if res[i] == 'Adload':
       if atrRF[i]!='Adload' and atbRF[i]!='Adload' and atrDT!='Adload' and atbDT!='Adload':
           if subRF[i] == subDT[i] and subRF[i] == subET[i]:
              res[i] = subRF[i]
# All models combined are certain that the classification isn't Adload
# At the same time, all models agree that it is either TrickBot or Trickster
```

In the example above, we trained several different classifiers and cross-referenced them with themselves, before also cross-referencing the different classifications. This process did increase our performance, but not by much.

Instead of focusing on TrickBot and Trickster, we have decided to focus on another misclassification. Namely, we tried to distinguish Artemis and Ursnif malware

Using the methods described above, we haven't improved our model, and the results of the confusion matrix have shown we had a very significant overfitting problem:

```
Training Score:          0.99940
Validation Score:        0.50852
```



As we can see from above, the training score is very high - meaning that the model fit the training data. However, the validation score is very low.

In order to combat the overfitting, we have decided to use a few tools(independently and in conjunction with one another):

1) We have reduced the number of estimators, iterations in order to reduce the power of the classifier.
2) We have used **KFold** to split the data into k consecutive folds.
3) We have used **KBest** and other feature selection methods to select features and improve performance.

Examples:

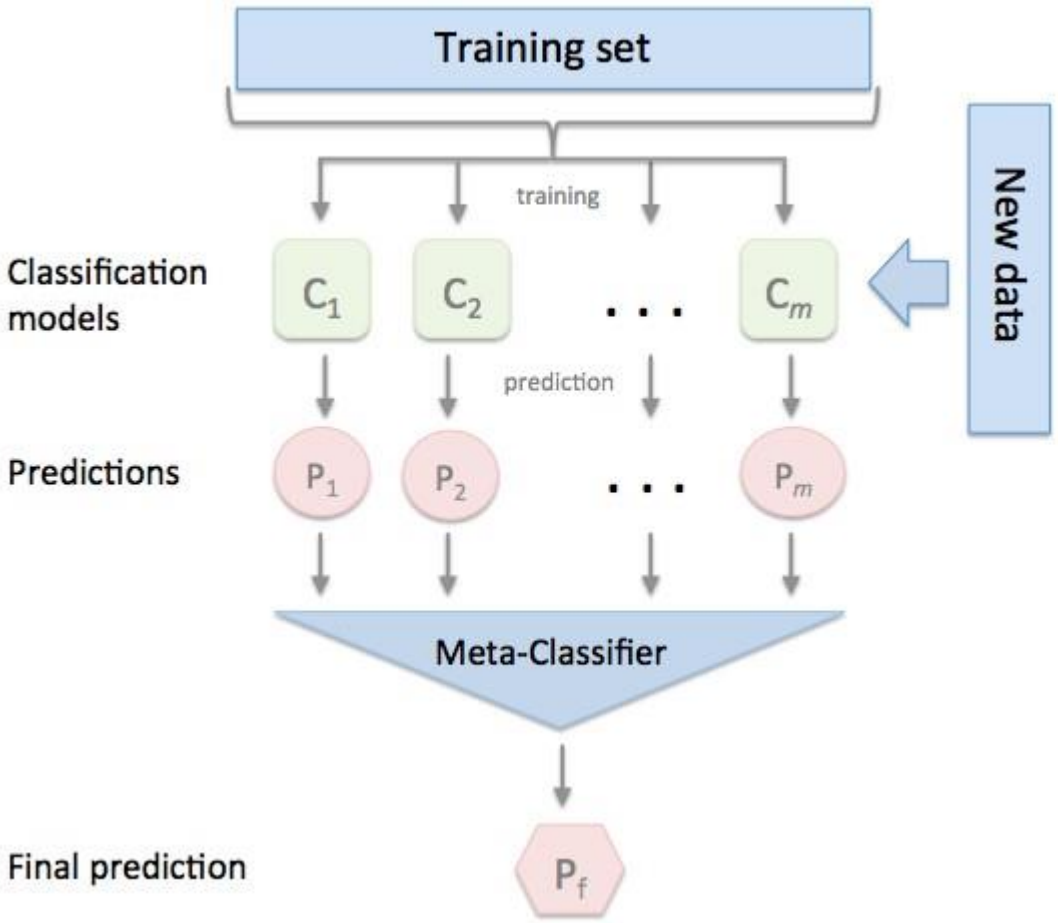
```
# Splitting into 10, using mutual_info_classif to select the K "best"
# features kf = KFold(n_splits=10, random_state=None,
    shuffle=False)
    X_new2 = SelectKBest(mutual_info_classif,k=100).fit_transform(Xtrain2, ytrain2)
```

Using the steps above, we have improved the classification to around 60%. The steps above haven't improved Trickster/TrickBot significantly.

However, in order to increase the model further, we have attempted to look for alternative methods.

Our best performing method:

We have discovered that stacking the classifiers is the optimal way to improve overall performance.



First, we decide on the ideal classification models by running them side by side and tinkering with their parameters:

```
clf = RandomForestClassifier(n_estimators=100, random_state=42,n_jobs=-1,
max_features="auto") clf.fit(X_train_scaled, y_train) print("RandomForestClassifier
Complete") print("Training Score: \t{:.5f}".format(clf.score(X_train_scaled, y_train)))
print("Validation Score: \t{:.5f}".format(clf.score(X_val_scaled, y_val)))

clf = DecisionTreeClassifier(random_state = 42) clf.fit(X_train_scaled,
y_train) print("DecisionTreeClassifier Complete") print("Training Score:
\t{:.5f}".format(clf.score(X_train_scaled, y_train))) print("Validation
Score: \t{:.5f}".format(clf.score(X_val_scaled, y_val)))


clf = ExtraTreesClassifier(n_estimators=100, random_state=0, n_jobs = -
1) clf.fit(X_train_scaled, y_train) print("ExtraTreesClassifier
Complete")
print("Training Score: \t{:.5f}".format(clf.score(X_train_scaled, y_train)))
print("Validation Score: \t{:.5f}".format(clf.score(X_val_scaled, y_val)))

clf = MLPClassifier(random_state=42, max_iter=350, early_stopping=True)
clf.fit(X_train_scaled, y_train) print("MLPClassifier Complete")
print("Training Score: \t{:.5f}".format(clf.score(X_train_scaled,
y_train))) print("Validation Score:
\t{:.5f}".format(clf.score(X_val_scaled, y_val)))

# etc
```

Then, after deciding on the ideal candidates we choose two or three of them to stack:

```
eclf = StackingClassifier(estimators=[('RFC', RFC), ('MLP', MLP)],final_estimator=ETC,n_jobs=-
1) eclf.fit(X_train_scaled, y_train)
```

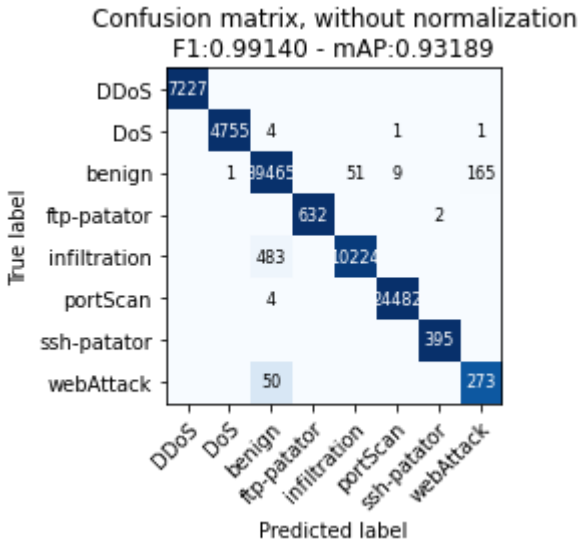
Choosing which classifiers to stack and which to use for the final estimator is important, which is why we experimented based on their scores, F1 and mAP rates, as well as combining classifiers which are conceptually different in order to use the strengths of both models.

The final estimator can benefit or impede based on whether it is trained on the predictions of the estimators alone, or on the original training data as well.

The results for each challenge are as seen below:

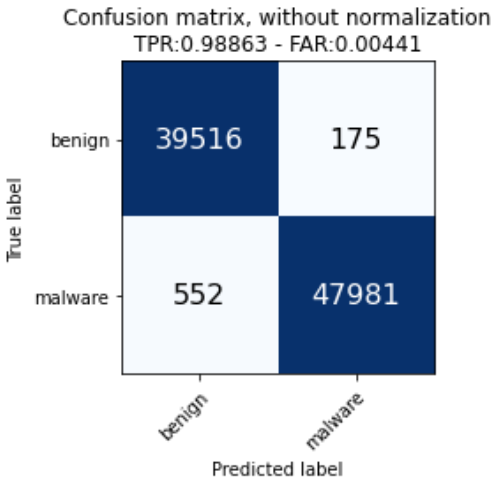
2017 FINE

F1: 0.99140 mAP: 0.93189



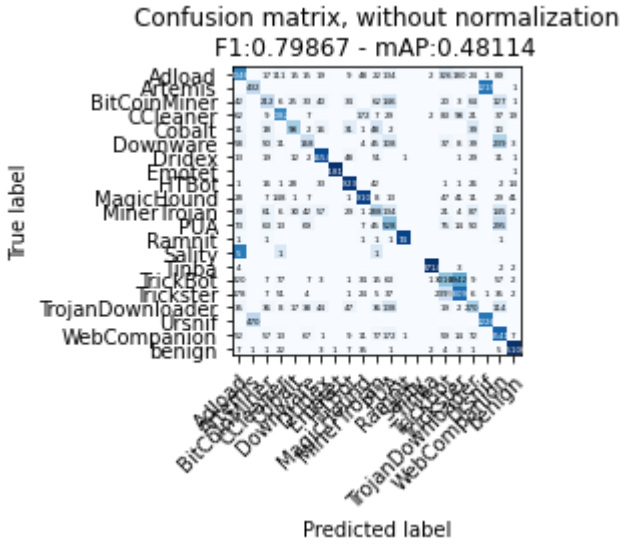
2017 TOP

TPR: 0.98863
FAR: 0.00441



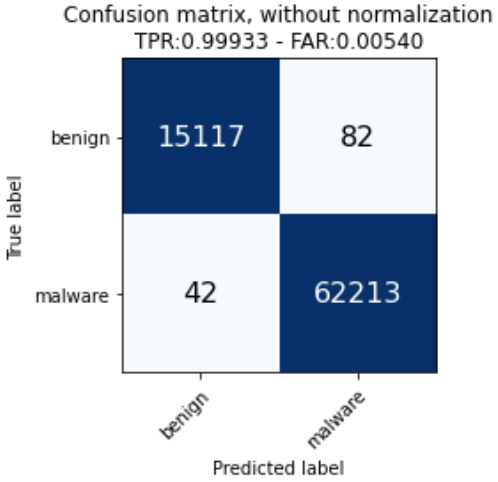
NETML FINE

F1: 0.79867
mAP: 0.48114



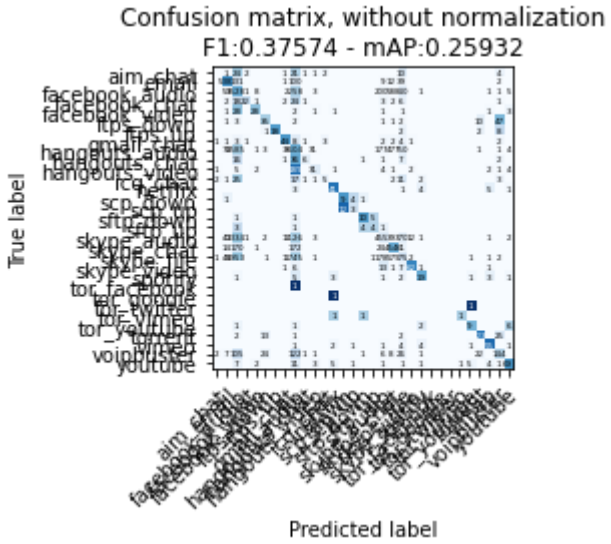
NETML TOP

TPR:	0.99933
FAR:	0.00540



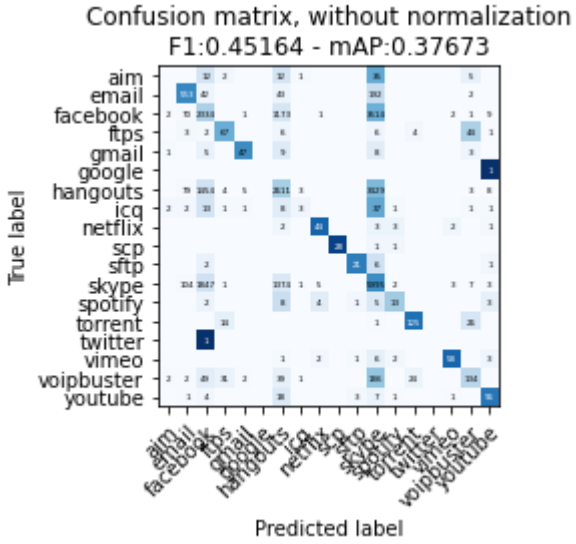
NONVPN FINE

F1: 0.37574 mAP: 0.25932



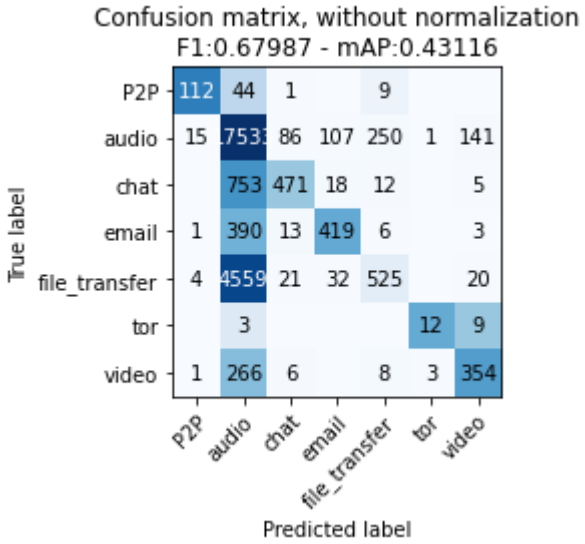
NONVPN MID

F1: 0.45164 mAP: 0.37673



NONVPN TOP

F1: 0.67987 mAP: 0.43116



Eval-AI scores:
DEV

cicids2017 fine

F1	mAP	Overall
0.99181	0.93232	0.92468

2017 Top

F1	mAP	Overall
0.98839	0.00367	0.98477

Nonvpn fine

F1	mAP	Overall
0.36645	0.27502	0.10078

Nonvpn mid

F1	mAP	Overall
0.45406	0.36613	0.16624

Nonvpn top

F1	mAP	Overall
0.68028	0.39676	0.26991

NetML fine

F1	mAP	Overall
0.83309	0.52822	0.44005

NetML top

F1	mAP	Overall
0.99928	0.00579	0.99349