MILESTONE 3

CSE 351

Haein Park
Pyungkang Hong
Wha Suk Lee

TRAINING AND TEST DATA

The chart below shows how large our dataset is,

	NAME	AMOUNT (# of files)
1	Ramnit	1541
2	Lollipop	2478
3	Kelihos_ver3	2942
4	Vundo	475
5	Simda	42
6	Tracur	751
7	Kelihos_ver1	398
8	Obfuscator.ACY	1228
9	Gatak	1013

Due to the limited size of the storage of our machines, we cannot train nor test models on the full dataset. Therefore, we set up two scenarios:

Scenario \ Type	Training (total)	Test (total)
Small	315 (35 * 9)	63 (7 * 9)
Large	5334	1332

The first scenario is using a **small subset of data. For each class of malware, 35 files are used for training and 7 files for testing. The other scenario is using a relatively **large** subset of data. If

a class has more than one thousand files, we fix the size of data to be one thousand. 80% (800 files) is used for training and 20% (200 files) for testing. If a class has less than one thousand files, use 80% of them for training and the remaining for testing.

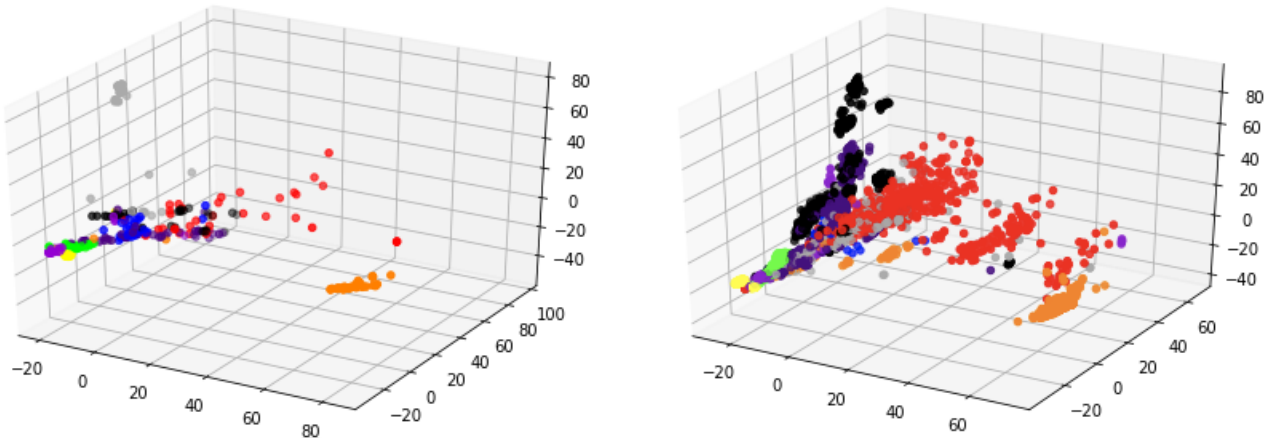
Feature	Summary
train_data_750.csv	4gram
train_dll.csv	Library dependencies
train_frequency.csv	How often each two bytes appears
train_instr_frequency.csv	How often each mnemonic appears
train_asm_image.csv	A gray-scale image representation

MACHINE LEARNING TECHNIQUES

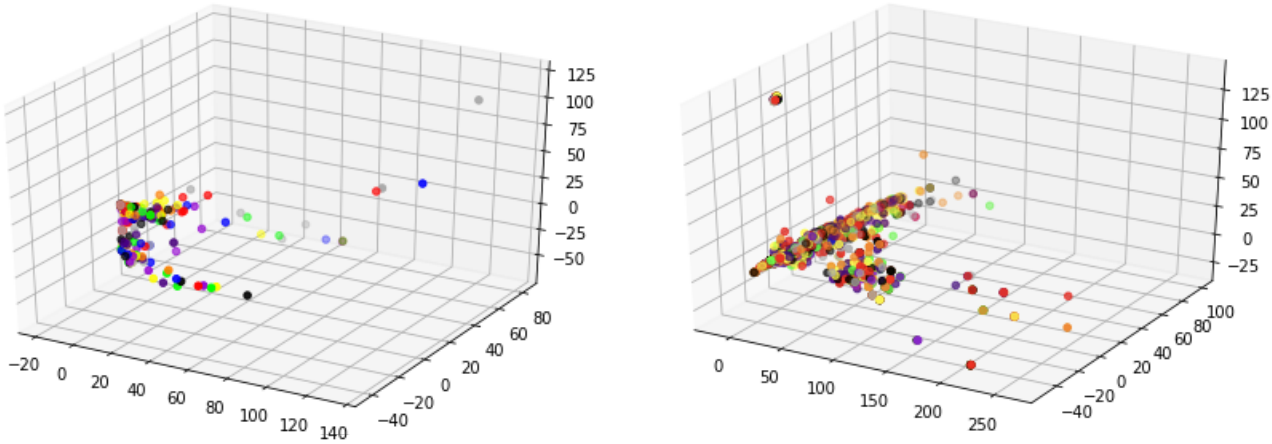
We decided to begin from basic classification techniques to advance.

1. KNN

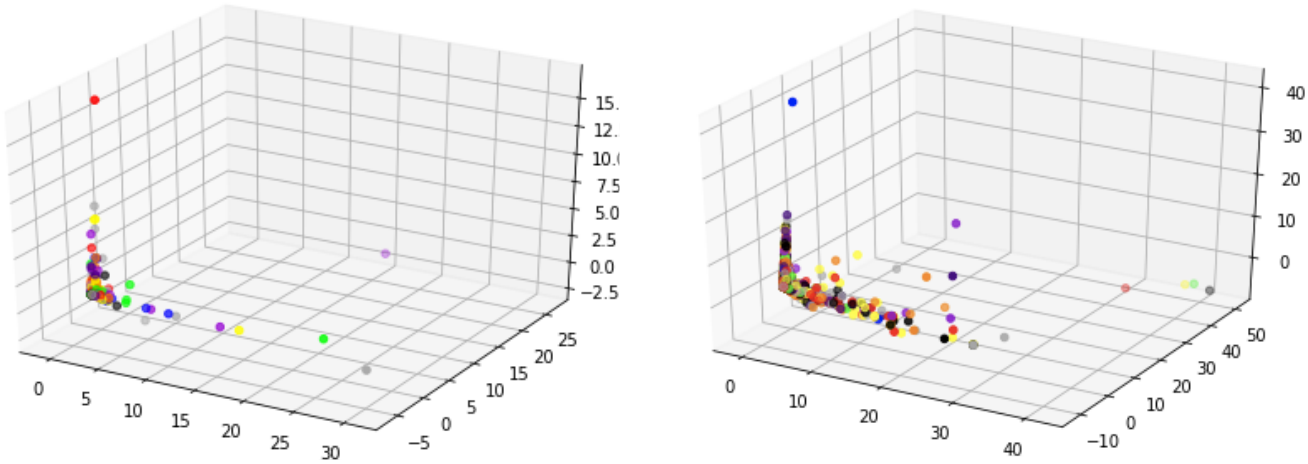
   Perhaps the most naive approach would be the k-nearest neighbor algorithm, KNN in short. The name already explains that an input data's class is determined by the K nearest neighbors. **Left plots are from the small subset of data, whereas right plots are the large subset of data**.



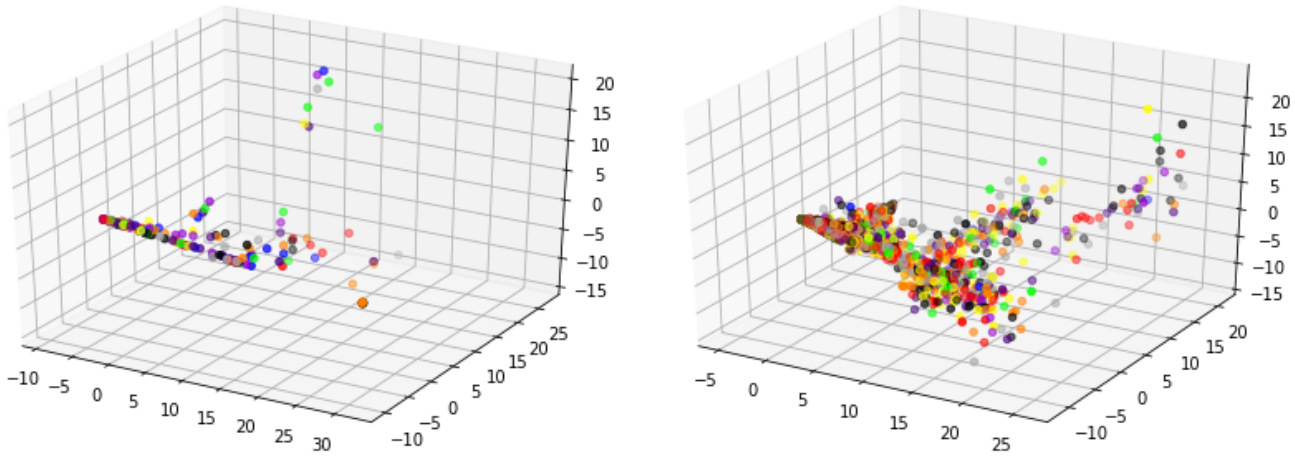
Visualization of 4gram using PCA (n=3)



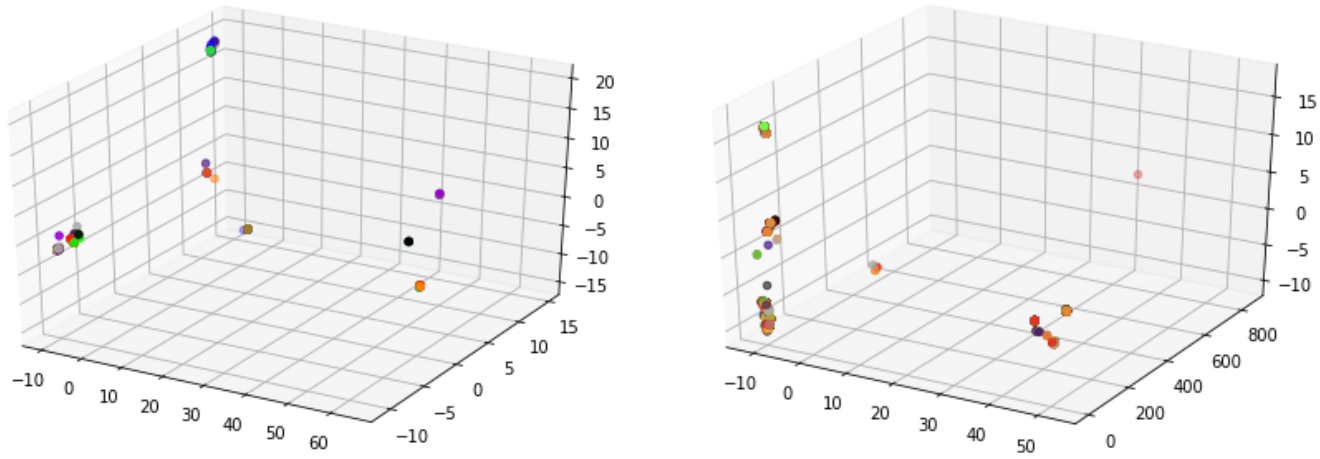
Visualization of DLL using PCA (n=3)



Visualization of Instruction Frequency using PCA (n=3)



Visualization of Frequency using PCA (n=3)



Visualization of a gray-scale image PCA (n=3)

Although no classification is applied yet, we can find an interesting observation in the visualization of 4gram. Red and orange dots are easily separable while the remaining classes are mostly tangled up. The orange refers to a malware class ‘Lollipop’. The red dots, ‘Ramnit’ class, also look promising.

**Please note that PCA is only for visualization. When we actually train and test the model, no dimensionality reduction is performed because we’ve found that it deteriorates the accuracy significantly. (For example, when the model was trained and tested with PCA-ed data with n = 2,

the highest accuracy we could achieve was 20%) Meanwhile, normalization step is taken with the help of “StandardScaler” from “sklearn.preprocessing”

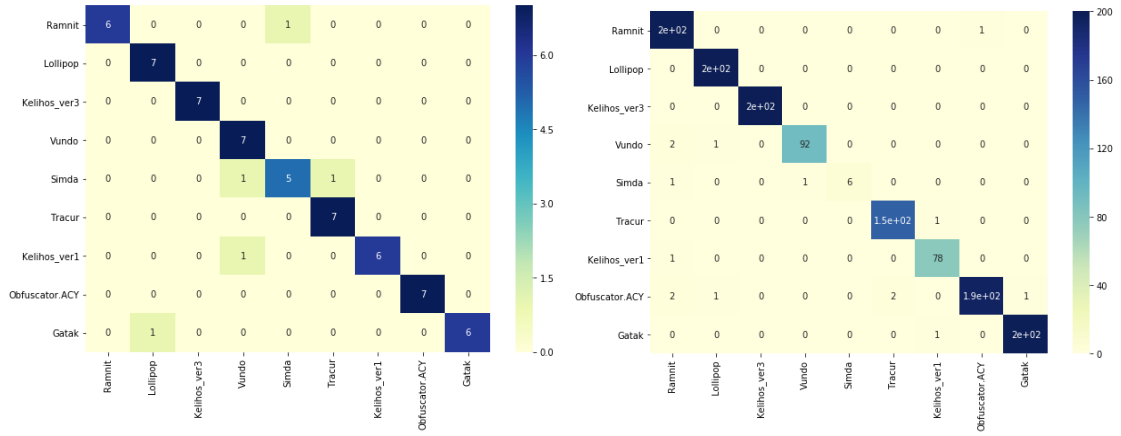
Id	FF85C074	00008378	0000E8A4	00800001	5E33C05B	F8FFFF8D	00840
0S7z4qxYTHPUDO8fglyA	0	1	0	1	0	0	
03nJaQV6K2ObICUmyWoR	0	0	0	0	0	0	
bGPHZFpAL3N957064wzj	0	0	0	0	0	0	
3WhAuJ8OlUsk2XT0ImiK	0	0	0	0	0	0	
0aU7XWsr8RtN94jvo3lG	0	0	0	0	0	0	

(Before the normalization)

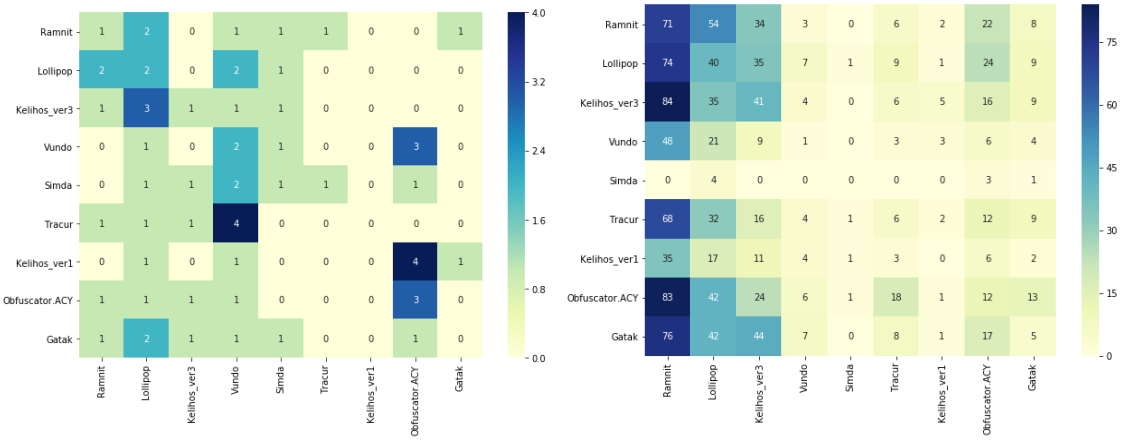
```
array([[-0.66704882,  2.48175158, -0.47003216, ..., -0.35355339,
        -0.26726124, -0.28065481],
       [-0.66704882, -0.40294122, -0.47003216, ..., -0.35355339,
        -0.26726124, -0.28065481],
       [-0.66704882, -0.40294122, -0.47003216, ..., -0.35355339,
        -0.26726124, -0.28065481],
       ...,
       [ 1.49914065, -0.40294122,  2.12751399, ..., -0.35355339,
        -0.26726124, -0.28065481],
       [ 1.49914065, -0.40294122, -0.47003216, ..., -0.35355339,
        -0.26726124, -0.28065481],
       [-0.66704882, -0.40294122, -0.47003216, ..., -0.35355339,
        -0.26726124, -0.28065481]])
```

(After the normalization)

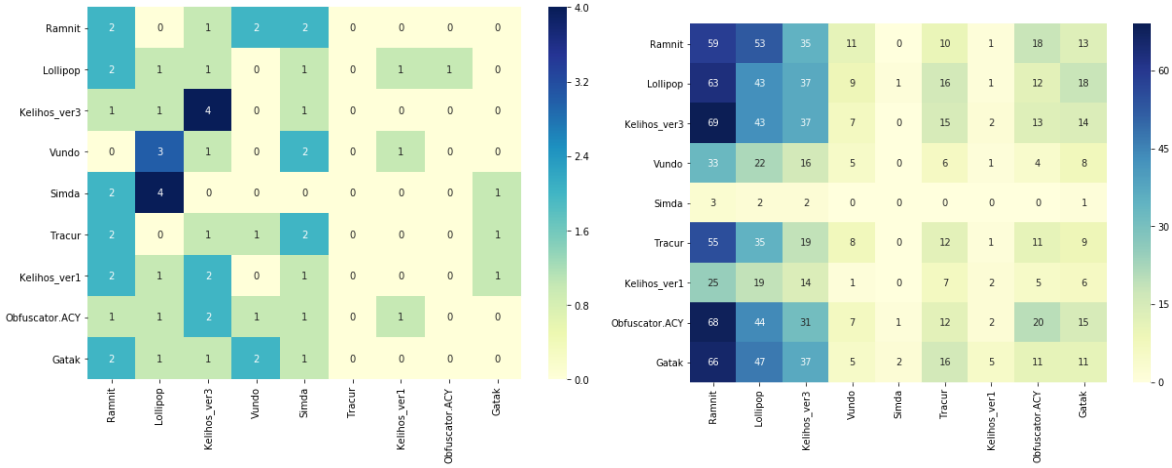
We then trained and tested KNN models, using a single feature, to see how accurate a single feature can achieve.



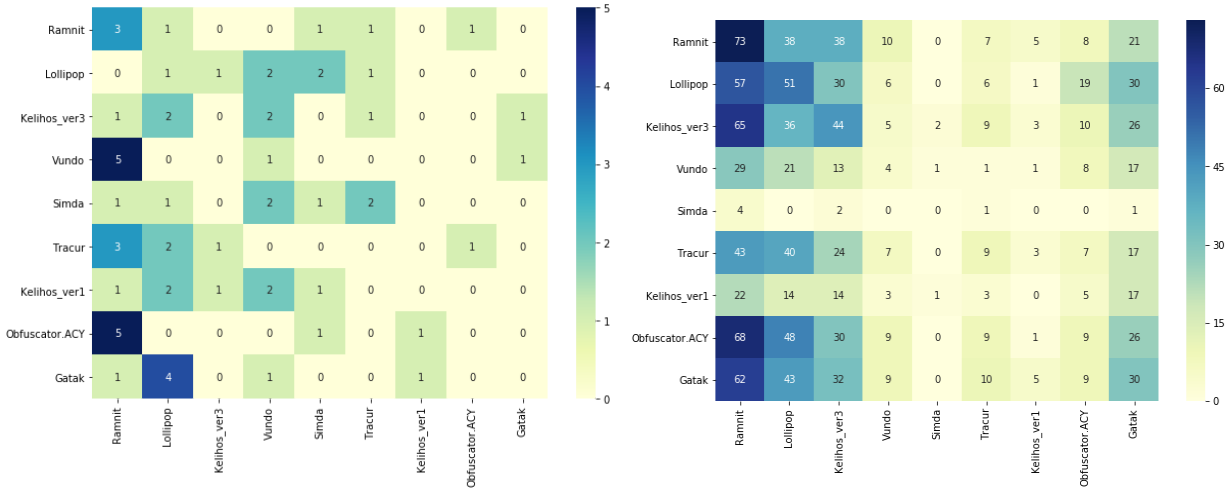
Confusion matrices for KNN trained with 4gram



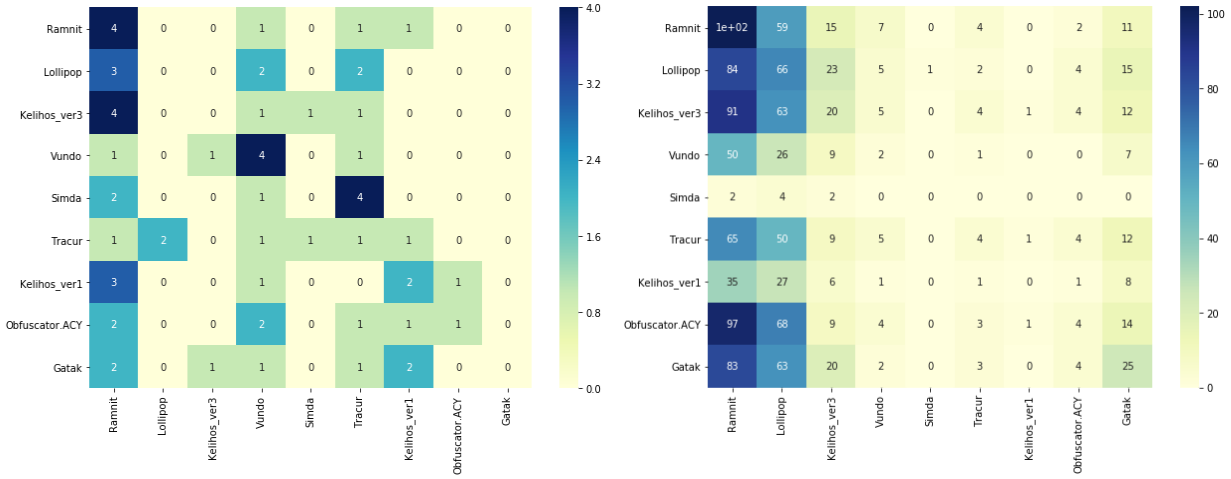
Confusion matrices for KNN trained with dll



Confusion matrices for KNN trained with Frequency



Confusion matrices for KNN trained with Instruction Frequency



Confusion matrices for KNN trained with a gray-scale image

Only ‘4gram’ has achieved high performance when used alone, its accuracy improved from 0.92 to 0.99. For other features, you can notice that instead of the diagonal of their confusion matrix getting darker, the whole left side of their matrix is getting darker. It means their chance of false-negative has increased.

	Small	Large
4gram	0.92063	0.988739
dll	0.15873	0.132132
Frequency	0.111111	0.141982
Instruction frequency	0.09523	0.165165
A gray-scale image	0.19476	0.167417

4gram Precision & Recall

```
pd.DataFrame([precision[0], recall[0]],
             columns = CLASSES,
             index=['Precision','Recall'])
```

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	1.000000	0.875	1.0	0.777778	0.833333	0.875	1.000000	1.0	1.000000
Recall	0.857143	1.000	1.0	1.000000	0.714286	1.000	0.857143	1.0	0.857143

4gram Precision & Recall

```
pd.DataFrame([precision[0], recall[0]],
             columns = CLASSES,
             index=['Precision','Recall'])
```

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.970732	0.990099	1.0	0.989247	1.00	0.986755	0.975000	0.994872	0.995
Recall	0.995000	1.000000	1.0	0.968421	0.75	0.993333	0.987342	0.970000	0.995

Precision and recall of 4gram-trained KNN (Up: Small dataset. Down: Large dataset)

**Precision and recall for other features are not included intentionally since their accuracies are below 0.20. You can understand them visually via their confusion matrices.

**Combining features are attempted to check if any combination helps to improve accuracy, but the accuracy either stays or decreases.

2. Neural Network

We then came up with a neural network, a state-of-the-art ML technique. A neural network is used to experiment if Ngram's performance is consistent in other ML algorithms. Furthermore, we know CNN has a tremendous capability in classifying images. Therefore, a gray-scale image will be tested as well.TensorFlow 2 is used to build a neural network. Again, no dimensionality reduction is applied due to the aforementioned problem. Two neural networks are built: a simple neural network with ngram and a convoluted neural network with a gray-scale image.

```
[9]  model = tf.keras.models.Sequential()
     model.add(tf.keras.layers.Dense(128,activation='relu',input_shape=(scaled_data.shape[1],)))
     model.add(tf.keras.layers.Dense(64))
     model.add(tf.keras.layers.Dense(32))
     model.add(tf.keras.layers.Dense(10, activation='softmax'))

[10] model.compile(optimizer='adam',
                   loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                   metrics=['accuracy'])
```

(Simple Neural Network Architecture)

```
[150] model = tf.keras.models.Sequential()
      model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 1)))
      model.add(tf.keras.layers.MaxPooling2D((2, 2)))
      model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
      model.add(tf.keras.layers.MaxPooling2D((2, 2)))
      model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))

      model.add(tf.keras.layers.Flatten())
      model.add(tf.keras.layers.Dense(64, activation='relu'))
      model.add(tf.keras.layers.Dense(10))

[ ]   model.compile(optimizer='adam',
                    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                    metrics=['accuracy'])
```

(Simple Convolutional Neural Network Architecture)

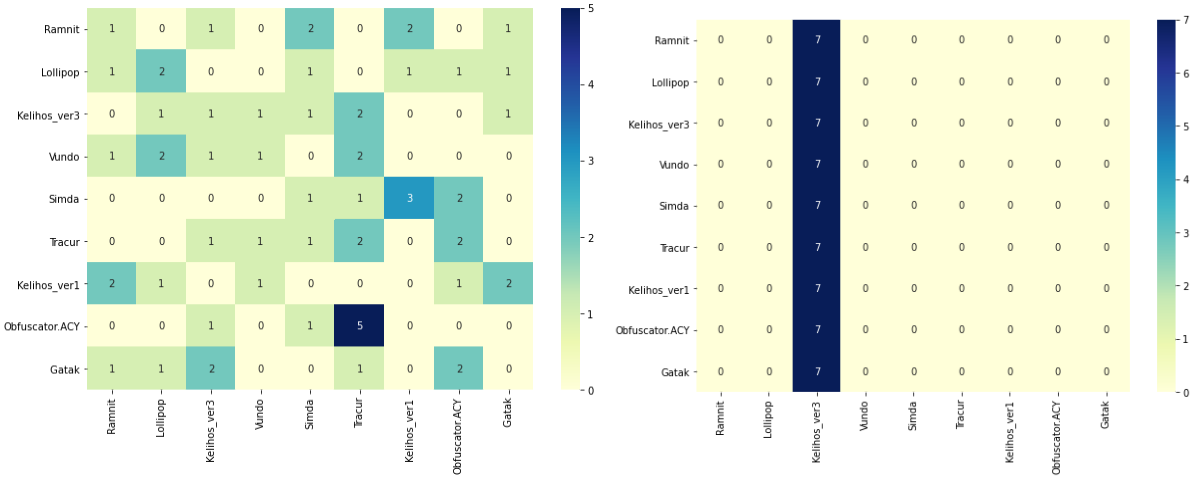
2-1. Neural Network with Small Dataset

```
[12]  1 model.evaluate(df_t,GT)

      2/2 [==============================] - 0s 3ms/step - loss: 2.3265 - accuracy: 0.1270
      [2.326510429382324, 0.1269841343164444]
```

```
[153]  1 model.evaluate(x_test,  y_test)

       2/2 [==============================] - 0s 3ms/step - loss: 2.3500 - accuracy: 0.1111
       [2.350039005279541, 0.1111111119389534]
```

(Small dataset. Prediction scores for both networks)

When using the small dataset, training accuracies were 0.9016 and 0.7206 for simple network and CNN, respectively. However, when tested, 0.1270 and 0.1111 were returned. Therefore, we can say they are highly overfitted.



(Confusion matrices for simple network and CNN)

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.222222	0.285714	0.142857	0.333333	0.166667	0.222222	0.0	0.0	0.0
Recall	0.285714	0.285714	0.142857	0.285714	0.142857	0.285714	0.0	0.0	0.0

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.15015
Recall	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.00000

Precision and recall of simple network and CNN (Small dataset)
(Nan is treated as 0)

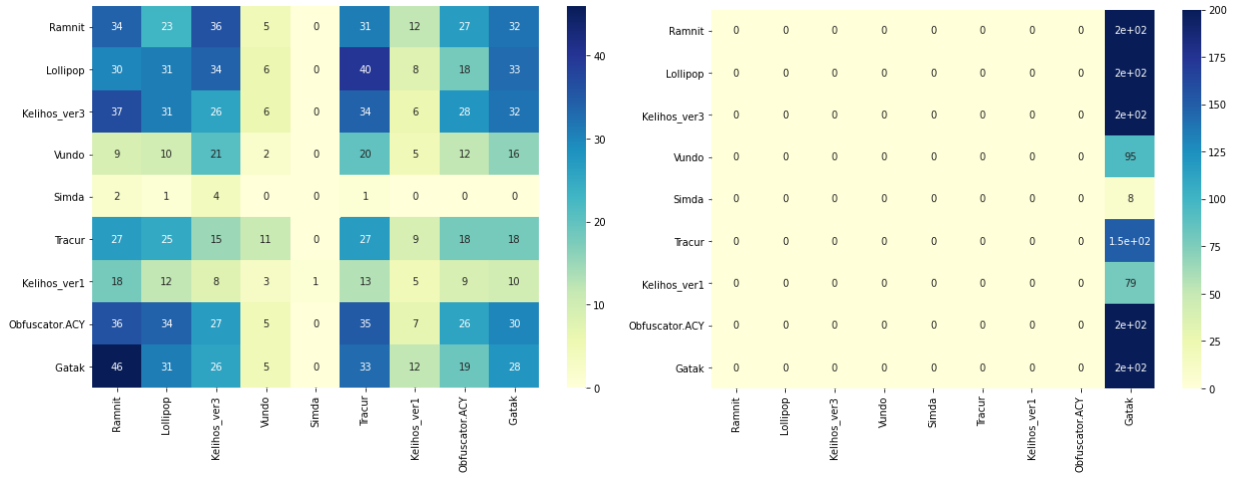
2-2. Neural Network with Large Dataset

```
[16]  1 model.evaluate(df_t,GT)

      42/42 [==============================] - 0s 3ms/step - loss: 2.3287 - accuracy: 0.1329
      [2.3286550045013428, 0.13288287818431854]
```

```
  1 model.evaluate(x_test,y_test)

  42/42 [==============================] - 0s 11ms/step - loss: 25.1044 - accuracy: 0.1502
  [25.104360580444336, 0.1501501500606537]
```

(Large dataset. Prediction scores for both networks)

Now, we used the same model architectures and trained with the large dataset. The training accuracies were 0.9636 and 0.7593 for simple network and CNN, respectively. Again, when tested, the accuracies decreased significantly, 0.1329 and 0.150150, respectively.



(Confusion matrices for simple network and CNN)

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.145	0.165094	0.13198	0.048780	NaN	0.114286	0.067568	0.161616	0.134021
Recall	0.145	0.175000	0.13000	0.042105	0.0	0.133333	0.063291	0.160000	0.130000

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.15015
Recall	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.00000

Precision and recall of simple network and CNN (Large dataset)

3. XGBoost

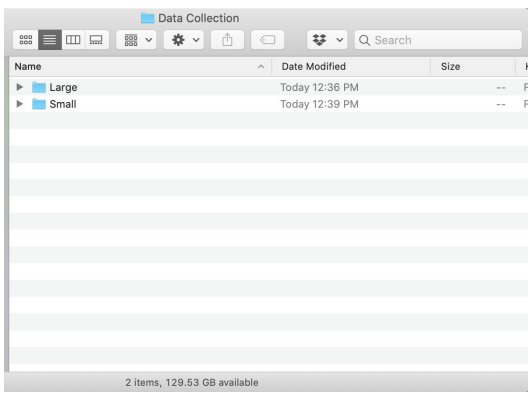
Finally, We tested XGBoost for our datasets.

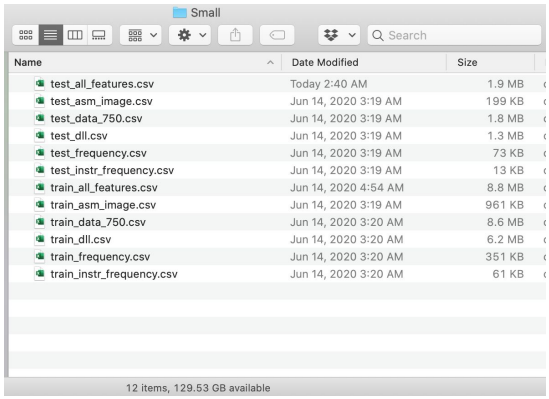
We chose to use XGBoost for this project because of two specific reasons. First, XGBoost is one of the most popular and powerful ML techniques that currently exist, which led many teams to win a variety of Kaggle competitions. Second and perhaps the most crucial reason, the first team of this particular Kaggle challenge, “Saynotooverfitting,” proposed that this technique is the key feature of their project, which contributed to their win for the most.

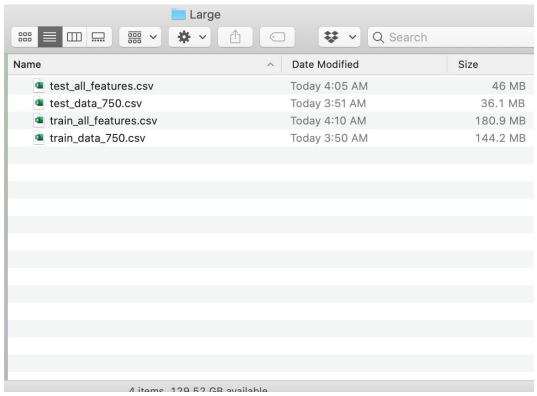
To train and use XGBoost, few hyperparameters should be provided to the model.

- **Max_depth:** this is the depth of the tree used by XGBoost. If the depth is too small, the model would not be delicate enough to classify accurately, and if the depth is too large, the model would end up overfitted.
- **Eta:** This is similar to the learning rate. This hyperparameter adjusts the weight on each step. 0.3 is the default value for most of the time, and often, lowering the value a little would improve the performance.
- **Objective:** This hyperparameter is to tell the model what type of result is expected. Setting it as **multi: softmax** would return the best of many classifications. Setting it as **multi: softprob** will return respective probabilities for each classification.
- **Num_class:** This hyperparameter tells the model how many target classes are there.

We followed to use the default values for the hyperparameter and observed the result. As mentioned above, there are two scenarios and several differences in datasets, in accordance with the scenarios. Like the other machine learning techniques, we applied XGBoost for the following train and test datasets: small_all_features, small_asm_image, small_data_750, small_dll, small_frequency, small_instr_frequency, large_all_features, and large_data_750.

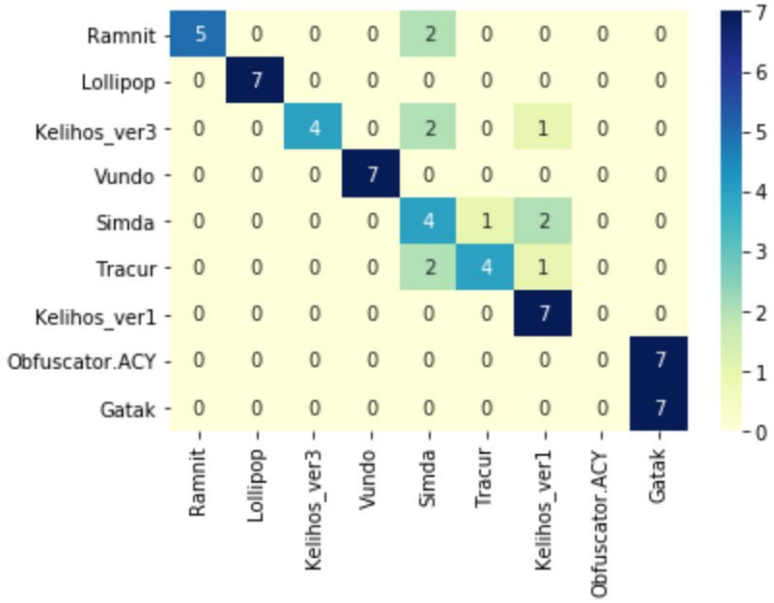




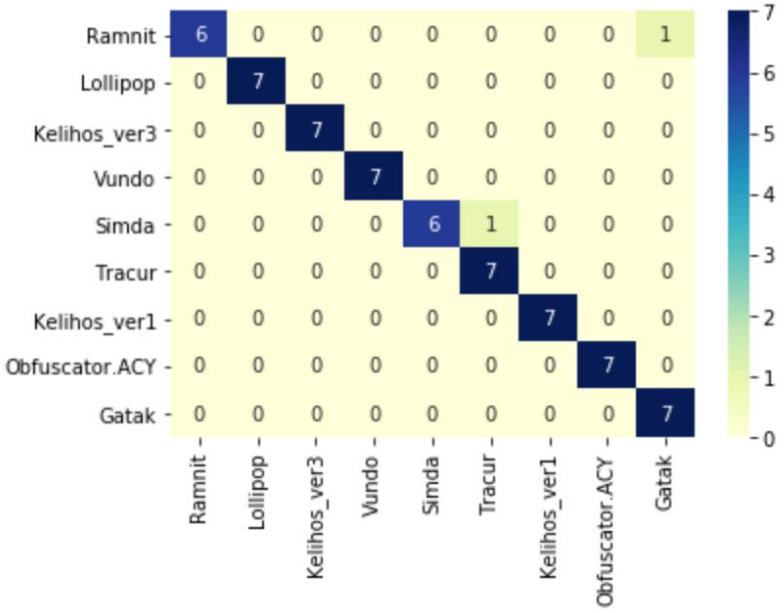


Here are our results of applying XGBoost to the datasets.

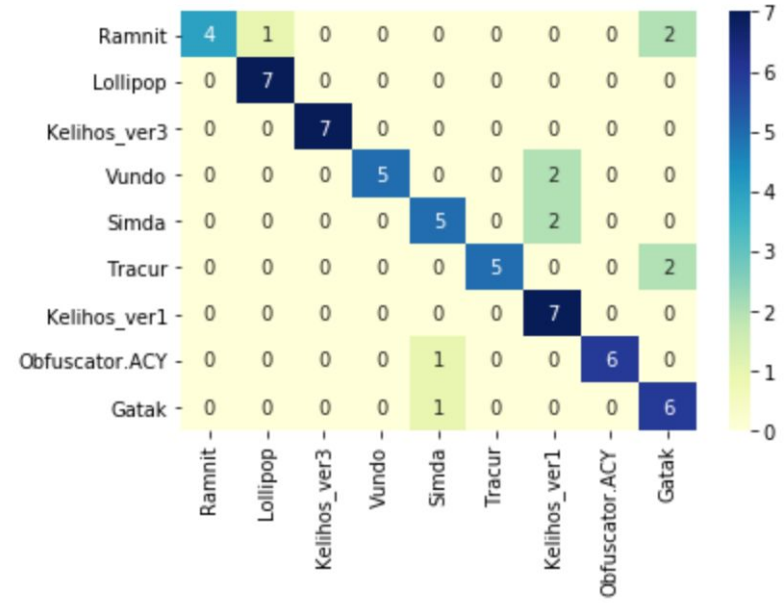
Small Datasets



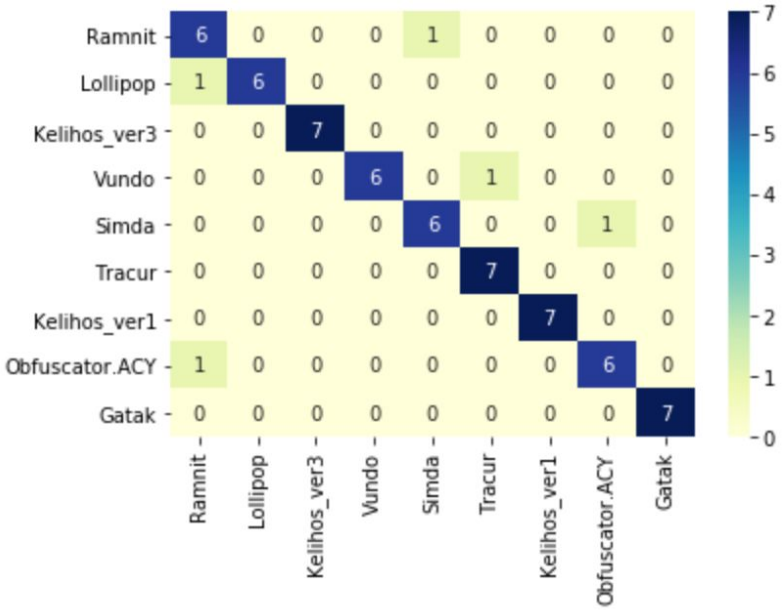
Confusion Matrix for XGBoost with a small gray-scale image datasets



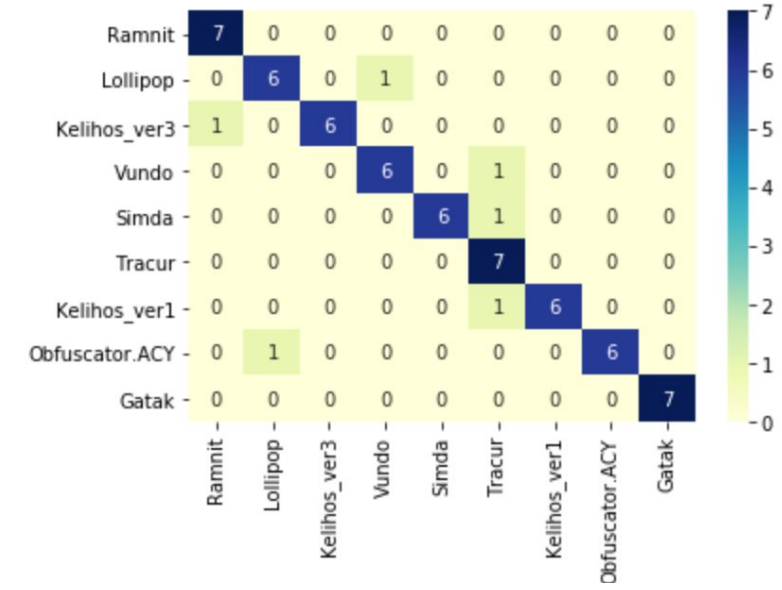
Confusion matrix for XGBoost with small 4grams datasets



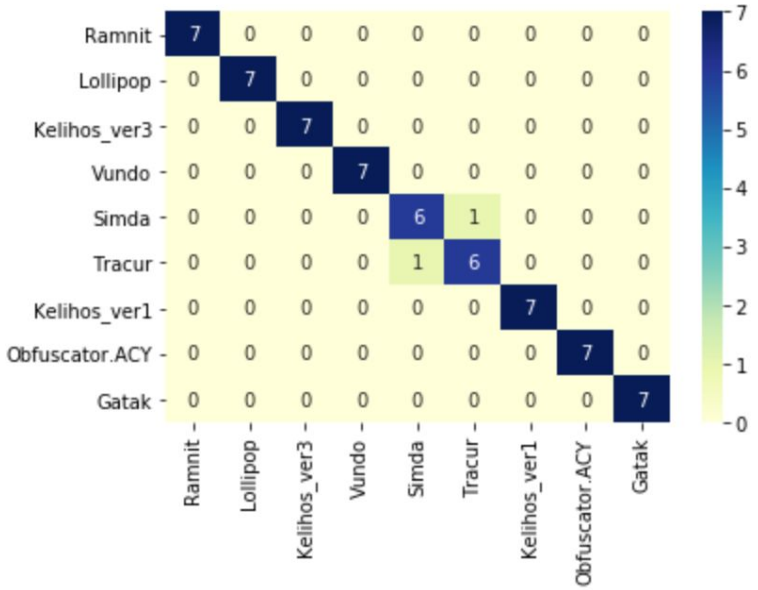
Confusion matrix for XGBoost with small dll datasets



Confusion matrix for XGBoost with small two bytes frequency datasets (train_frequency.csv and test_frequency.csv)

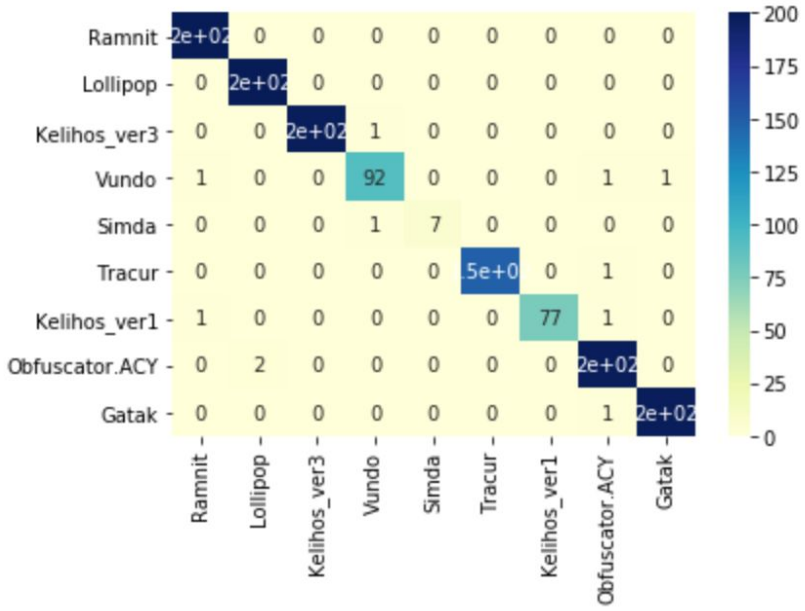


Confusion matrix for XGBoost with small instruction frequency datasets

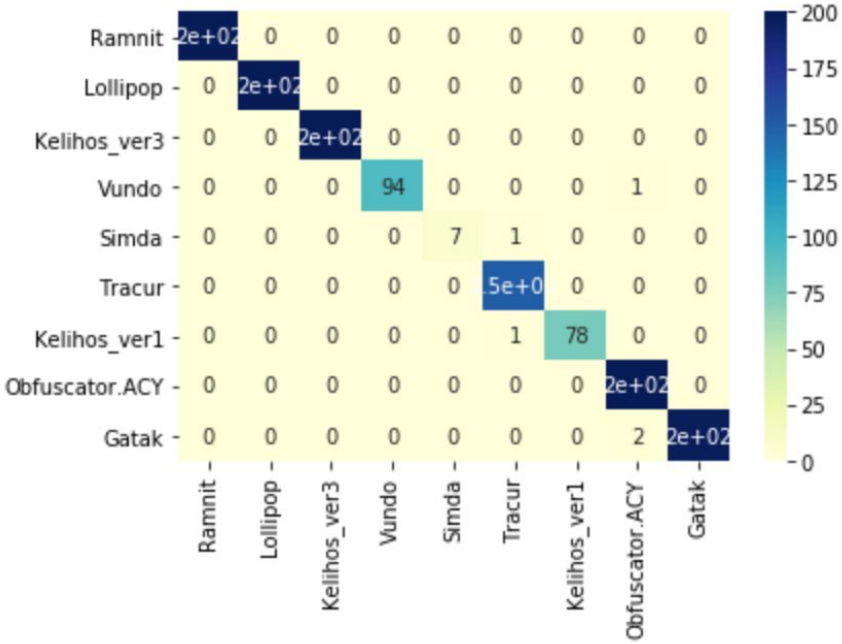


Confusion matrix for XGBoost with small all features datasets

Large Datasets



Confusion matrix for XGBoost with large 4gram datasets



Confusion matrix for XGBoost with large all features datasets

To sum up the results of XGBoost, compared to the other algorithms, XGBoost showed relatively stronger performance.

	Accuracy
small_asm_image	0.71429
small_4gram	0.96825
small_dll	0.82539
small_frequency	0.92063
small_instr_frequency	0.90476
small_all_features	0.96825
large_4gram	0.99174
large_all_features	0.99625

As you can see, the **large_all_features** dataset has recorded the highest accuracy and the **small_asm_image** has recorded the lowest accuracy for XGBoost.

Below are the precision and recall tables of small_4gram and large_4gram.

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	1.000000	1.0	1.0	1.0	1.000000	0.875	1.0	1.0	0.875
Recall	0.857143	1.0	1.0	1.0	0.857143	1.000	1.0	1.0	1.000

Small_4gram

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.990099	0.990099	1.000	0.978723	1.000	1.000000	1.000000	0.980198	0.995
Recall	1.000000	1.000000	0.995	0.968421	0.875	0.993333	0.974684	0.990000	0.995

Large_4gram

Since we are going to compare the performance of all algorithms we used by the 4gram features in the conclusion section, we only included the tables for 4gram features.

CONCLUSION

To sum all the results obtained from Milestone 3, we have used three algorithms to tackle the malware classification challenge. Out of KNN, Convolutional Neural Network, and XGBoost, XGBoost showed the best performance in general.

It's also interesting that CNN has performed really bad in this particular project. In other words, a gray-scale image representation of .bytes files is not useful.

We chose the 4gram feature to be the standard of comparison among the algorithms. Intuitively, we thought that the 4gram feature would explain many things about the dataset. For example, suppose you're reading a book. A sentence explains more about its context than a single word. We thought this intuition can be applied in 4gram as well. Therefore, rather than using other criteria, we decided to use the 4gram feature to be our determinant of performance.

Using 4gram as the key feature, we decided to compare the performance of the three algorithms based on their precision, recall, and accuracy.

Small dataset: Precision, Recall, and Accuracy

KNN

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	1.000000	0.875	1.0	0.777778	0.833333	0.875	1.000000	1.0	1.000000
Recall	0.857143	1.000	1.0	1.000000	0.714286	1.000	0.857143	1.0	0.857143

Neural Network

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.222222	0.285714	0.142857	0.333333	0.166667	0.222222	0.0	0.0	0.0
Recall	0.285714	0.285714	0.142857	0.285714	0.142857	0.285714	0.0	0.0	0.0

XGBoost

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	1.000000	1.0	1.0	1.0	1.000000	0.875	1.0	1.0	0.875
Recall	0.857143	1.0	1.0	1.0	0.857143	1.000	1.0	1.0	1.000

	Accuracy
KNN	0.92063
Neural Network	0.1270
XGBoost	0.96825

Large Dataset: Precision, Recall, and Accuracy

KNN

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.970732	0.990099	1.0	0.989247	1.00	0.986755	0.975000	0.994872	0.995
Recall	0.995000	1.000000	1.0	0.968421	0.75	0.993333	0.987342	0.970000	0.995

Neural Network

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.145	0.165094	0.13198	0.048780	NaN	0.114286	0.067568	0.161616	0.134021
Recall	0.145	0.175000	0.13000	0.042105	0.0	0.133333	0.063291	0.160000	0.130000

XGBoost

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Precision	0.990099	0.990099	1.000	0.978723	1.000	1.000000	1.000000	0.980198	0.995
Recall	1.000000	1.000000	0.995	0.968421	0.875	0.993333	0.974684	0.990000	0.995

	Accuracy
KNN	0.988739
Neural Network	0.1329
XGBoost	0.99174

FUTURE WORK

- Check if the accuracy changes if N changes in the Ngram.
- Can this algorithm be applied to sentiment analysis as well?
- Would this algorithm be applicable to modern programming languages as well?
    - Ex. Java, Python, C, etc
- If this project was done in unsupervised learning, would we be able to find out other classes?

GitHub