SWARM AND EVOLUTIONARY COMPUTING

Hyper-Parameter Optimization: A Review and comparison of Algorithms using Malware Detection

FINAL PROJECT REPORT

Submitted By:

Anukriti (2K17/IT/027)
Anurag Mudgil (2K17/IT/029)

DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

ACKNOWLEDGEMENT

I am very thankful to Ms, Garima Gupta (Professor, IT) and all the faculty members of the Department of Information Technology of DTU. They all provided us with immense support and guidance for the project.

I would also like to express my gratitude to the University for providing us with the laboratories, infrastructure, testing facilities and environment which allowed us to work without any obstructions.

I would also like to appreciate the support provided to us by our lab assistants, seniors and our peer group who aided us with all the knowledge they had regarding various topics.

Date: 28th November, 2020

Place: New Delhi

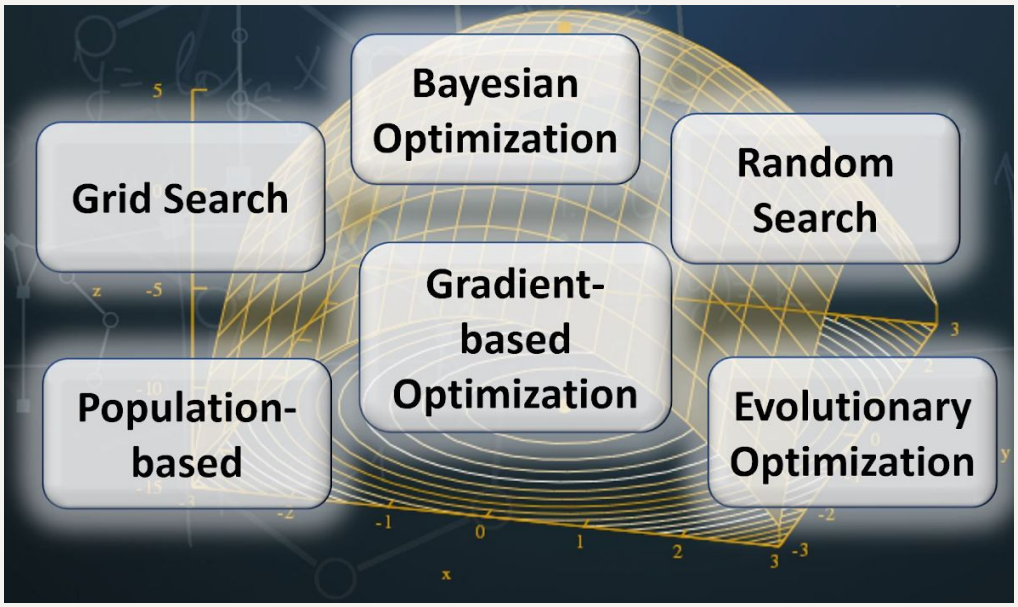
Anukriti (2K17/IT/027)

Anurag Mudgil (2K17/IT/029)

TABLE OF CONTENTS

PROBLEM STATEMENT

Every machine learning system has hyperparameters which determine how our model is structured in the first place. They are extremely important for any machine learning algorithm since they directly control the behaviors of training algorithms and have a significant effect on the performance of machine learning models. Several techniques have been developed and successfully applied for certain application domains. However, this work demands professional knowledge and expert experience. Therefore, if an efficient hyperparameter optimization algorithm can be developed to optimize any given machine learning method, it will greatly improve the efficiency of machine learning.
Machine Learning models tuning is a type of optimization problem. We have a set of hyperparameters and we aim to find the right combination of their values which can help us to find either the minimum (eg. loss) or the maximum (eg. accuracy) of a function.



Through this project, we will be reviewing the main types of algorithms used for hyper-parameter optimization using a machine learning classifier used to classify binaries between legitimate and malicious binaries. We aim to find the set of hyper-parameters leading to the lowest error on the validation set. This can be particularly important when comparing how different Machine Learning models perform on a dataset. In fact, it would be unfair for example to compare an SVM model with the best Hyperparameters against a Random Forest model which has not been optimized.

RELATED WORK

This project can be mainly divided into 2 sub-sections:

a.) Malware Detection from Portable Executable Binary Files:

Various approaches exist in the non-signature based methods such as heuristics, machine learning and hybrid techniques (Treadwell and Zhou, 2009, Firdausi et al., 2010, Qin et al., 2010, Schultz et al., 2001). Malware detection is the process of differentiating malware from benign programs. Malware detection techniques (Signature-based and Non-signature based) by its type of analysis are categorized into static and dynamic. We have used non-signature based techniques which use both kinds of analysis. Shabtai et al. (2009) presented taxonomy for classifying detection methods of malware by machine learning methods based on static features extracted from the executable. They explained various static features, among which PE features are grouped under six categories. Vinod et al. (2011) used three different types of features- mnemonic n-gram, principal instruction opcodes, and PE header entries. They used scatter criterion for feature selection on two datasets created with pairing obfuscated and packed malware with benign samples. Ahmadi et al. (2016) proposed a novel paradigm to group malware variants into different families. In this work, feature extraction and selection methods are given more importance than classification algorithms. Features are extracted from content and the structure of the sample, so the proposed method will even work on packed and obfuscated samples. Bai et al. (2014) built an initial feature set of 197 features, extracted from different PE header fields. They used filter and wrapper features selection methods to reduce the features to 19 and 20 respectively. The selected features were used to train tree based classifiers such as Decision Tree (DT), Random forest (RF), Bagged and Boosted decision tree. Baldangombo et al. (2013) used the three feature set DLL calls, API calls and PE header fields values as both separately and combined. Belaoued and Mazouzi (2015) presented a real-time PE malware detection system, which is based on the analysis of the information stored in the PE-optional header fields. Chi-square and Phi coefficient were used for feature selection and with selected features Rotation forest classifier was trained and tested. Altaher et al. (2012) used API calls of DLL as major feature and Information Gain (IG) to reduce the total number of features to 11. They adopted evolving clustering methods to build the rule-based classifier and compared their results. Liao (2012) selected only five fields of PE headers as the feature and implemented a hand-crafted rule-based algorithm for classification. The malware samples used in previous works are either unavailable due to various reasons or the datasets are private to the company.

b.) Analysis of Hyper-Parameter Optimization Algorithms:

In this we have applied and compared various Hyper Parameter Optimization algorithms like Genetic Algorithm , Random Search etc. on Malware Detection. Previously a similar paper on this Hyper-Parameter Optimization: A Review of Algorithms and Applications where different algorithms were compared on various parameters like Learning Rate , Optimizer etc. An optimization algorithm is a procedure which is executed iteratively by comparing various solutions till an optimum or a satisfactory solution is found . Various Optimisation Algorithms have their own benefits and limitations and different algorithms work better for different problems. Algorithms for Hyper-Parameter Optimization compared only random search and two new greedy sequential methods (Gaussian Process and Parzen WIndow) based on the expected improvement criterion .They applied these models in DBN(Deep Belief Networks) and made two important contributions that Random search is competitive with the manual optimization of DBNs in and Automatic sequential optimization outperforms both manual and random search. Hyperband: Bandit-Based Configuration evaluation for Hyperparameter Optimization is a famous study where the Bayesian Optimisation is compared with a relatively new optimisation algorithm hyperband. Hyperband is a principled early-stopping method that adaptively allocates a predefined resource, e.g., iterations, data samples or number of features, to randomly sampled configurations. Hyperband algorithm was applied by working with the MNIST dataset and optimizing hyperparameters for the LeNet convolutional neural network trained using mini-batch SGD. Comprehensive analysis of gradient-based hyperparameter optimization algorithms - This paper also analyzes hyperparameter optimization problems.It focuses more on the neural network models with large amounts of hyperparameters using gradient-based algorithms (Random Search ,Greedy , HOAG, Dr. Mad) are analyzed. Random Search algorithm was used as a baseline. They applied these algorithms on a dataset for Z class classification problems. There were two model selection criteria: cross-validation and evidence lower bound. A Survey on Hyperparameters Optimization Algorithms of Forecasting Models in Smart Grid - This paper provides literature review of forecasting models and the optimization methods used to tune their hyperparameters using forecasting models in Smart Grid. Analysis of existing optimisation algorithms is also done. Forecasting in SG is done mainly by various algorithms like SVM, BN and ANN so first the hyperparameters for these were found and optimised. Later, a survey on all the surveys is also done.

OUR APPROACH AND SOLUTION

Classification Problem:

We decided to take a binary classification task to classify portable executable binary files as legitimate/benign or malicious.

Various tasks involved:

- ➢ Data Collection
- ➢ Feature Extraction from binaries
- ➢ Feature Selection
- ➢ Classification algorithm Selection
- ➢ Optimization and Evaluation

DATASET CREATION

- **Legitimate Files:** For legitimate files collection, we gathered all the Windows binaries (exe + dll) from installed folders of applications of legitimate software from different categories. They can be downloaded in Apps for Windows. This way, we could gather around 41323 files.
- **Malware Files:** All of the malware files are collected from VirusShare collection by downloading one archive (134th) and malicia-project.com. This way, we could gather around 96724 malicious files.

FEATURE EXTRACTION

Since machine learning algorithms can only take integer or float features as parameters, it was important to extract features from the files obtained and for this, we extracted most of the relevant features of portable executable binary files using a Python module to read and work with PE (Portable Executable) files repo. Most PE parameters obtained were integers such as field size, addresses etc. We also created some additional important features such as the mean, minimum and maximum of entropy for sections and resources from the entropy of section for packer detection.

The final list of 54 features obtained were:

```
{ Name, md5, Machine, SizeOfOptionalHeader, Characteristics, MajorLinkerVersion, MinorLinkerVersion, SizeOfCode, SizeOfInitializedData, SizeOfUninitializedData,
AddressOfEntryPoint, BaseOfCode, BaseOfData, ImageBase, SectionAlignment, FileAlignment, MajorOperatingSystemVersion, MinorOperatingSystemVersion, MajorImageVersion,
MinorImageVersion, MajorSubsystemVersion, MinorSubsystemVersion, SizeOfImage, SizeOfHeaders, CheckSum, Subsystem, DllCharacteristics, SizeOfStackReserve,
SizeOfStackCommit, SizeOfHeapReserve, SizeOfHeapCommit, LoaderFlags, NumberOfRvaAndSizes, SectionsNb, SectionsMeanEntropy, SectionsMinEntropy, SectionsMaxEntropy,
SectionsMeanRawsize, SectionsMinRawsize, SectionMaxRawsize, SectionsMeanVirtualsize, SectionsMinVirtualsize, SectionMaxVirtualsize, ImportsNbDLL, ImportsNb,
ImportsNbOrdinal, ExportNb, ResourcesNb, ResourcesMeanEntropy, ResourcesMinEntropy, ResourcesMaxEntropy, ResourcesMeanSize, ResourcesMinSize, ResourcesMaxSize,
LoadConfigurationSize, VersionInformationSize }
```

FEATURE SELECTION

Our aim was to reduce the 54 features extracted to a smaller set of features which are the most relevant for differentiating legitimate binaries from malware. accordingly. For this, we used Extra Trees Classifier which is a type of ensemble learning technique very similar to a Random Forest Classifier and only differs from it in the manner of construction of the decision trees in the forest. To perform feature selection, each feature is ordered in descending order according to the Gini Importance of each feature and the user selects the top k features.

```
# Feature selection using Trees Classifier
features = []
fsel = ske.ExtraTreesClassifier().fit(X, y)
model = SelectFromModel(fsel, prefit=True)
X_new = model.transform(X)
nb_features = X_new.shape[1]

print('%i features identified as important:' % nb_features)
indices = np.argsort(fsel.feature_importances_)[::-1][:nb_features]
for f in range(nb_features):
    print("%d. feature %s (%f)" % (f + 1, data.columns[2+indices[f]], fsel.feature_importances_[indices[f]]))
```

We extracted the below 13 most relevant features for our classification task:

```
13 features identified as important:
1. feature DllCharacteristics (0.161399)
2. feature Characteristics (0.108048)
3. feature Machine (0.105783)
4. feature VersionInformationSize (0.068582)
5. feature Subsystem (0.059484)
6. feature ImageBase (0.058096)
7. feature SectionsMaxEntropy (0.055294)
8. feature SizeOfOptionalHeader (0.051717)
9. feature ResourcesMaxEntropy (0.037973)
10. feature MajorSubsystemVersion (0.033847)
11. feature ResourcesMinEntropy (0.026483)
12. feature MajorOperatingSystemVersion (0.020831)
13. feature SectionsMinEntropy (0.018710)
```

CLASSIFICATION MODELS IN CONSIDERATION

We performed comparison for our baseline model selection for which we took into consideration the following algorithms and compared their classification scores:

```
#Algorithm comparison
algorithms = {
        "DecisionTree": tree.DecisionTreeClassifier(max_depth=10),
        "RandomForest": ske.RandomForestClassifier(n_estimators=50),
        "GradientBoosting": ske.GradientBoostingClassifier(n_estimators=50),
        "AdaBoost": ske.AdaBoostClassifier(n_estimators=100),
        "GNB": GaussianNB()
    }

results = {}
print("\nNow testing algorithms")
for algo in algorithms:
    clf = algorithms[algo]
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    print("%s : %f %%" % (algo, score*100))
    results[algo] = score
```

- **Random Forest:** Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes or mean/average prediction of the individual trees. In case of Random Forests, **hyperparameters** include the number of decision trees in the forest and number of features considered by each tree when splitting a node.

- **Decision Tree Classifier:** A decision tree is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements. **Hyperparameter of decision trees** is max_features which is the number of features to consider when looking for the best split.

- **Gradient Boosting:** Gradient boosting is a type of machine learning boosting. It relies on the intuition that the best possible next model, when combined with previous models, minimizes the overall prediction error. The key idea is to set the target outcomes for this next model in order to minimize the error. The **hyperparameters of gradient boosting** that can be chosen are learning_rate, max_depth and the n_estimators.

- **AdaBoost:** AdaBoost is short for Adaptive Boosting and is a very popular boosting technique which combines multiple weak classifiers into a single strong classifier. It used to boost the performance of decision trees on binary classification problems. One of the important

hyperparameters for AdaBoost algorithms is the number of decision trees used in the ensemble.

- **Gaussian Naive Bayes:** Gaussian Naive Bayes is a variant of Naive Bayes that follows Gaussian normal distribution and supports continuous data. It's specifically used when the features have continuous values. It's also assumed that all the features are following a gaussian distribution i.e, normal distribution. One of **its hyperparameters** is var_smoothing (float, default=1e-9): Portion of the largest variance of all features that is added to variances for calculation stability.

  After the comparison, **Random Forest Classifier** is seen to provide maximum classification score and hence, chosen as the classifier for our task.

```
Now testing algorithms
DecisionTree : 99.018472 %
RandomForest : 99.413256 %
GradientBoosting : 98.717856 %
AdaBoost : 98.594712 %
GNB : 70.336834 %

Winner algorithm is RandomForest with a 99.413256 % success
```

Optimization Problem and Evaluation:

HYPER-PARAMETERS

These are adjustable parameters that must be tuned in order to obtain a model with optimal performance. Model Hyperparameters are the properties that govern the entire training process. In any machine learning algorithm, these parameters need to be initialized before training a model.

The main parameters used by a Random Forest Classifier are:

- criterion = the function used to evaluate the quality of a split
- max_depth = maximum number of levels allowed in each tree
- max_features = maximum number of features considered when splitting a node
- min_samples_leaf = minimum number of samples which can be stored in a tree leaf
- min_samples_split = minimum number of samples necessary in a node to cause node splitting
- n_estimators = number of trees in the ensemble

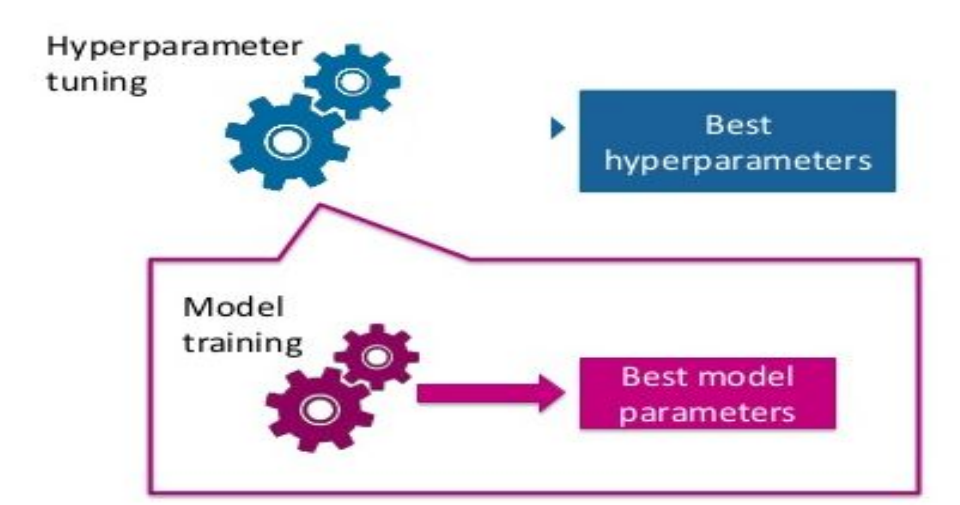
For more information about parameters, refer to scikit-learn Documentation

NEED FOR HYPER-PARAMETER TUNING

Hyperparameters are important because they directly control the behaviour of the training algorithm and have a significant impact on the performance of the model being trained. A good choice of hyperparameters can really make an algorithm shine.

Choosing good hyperparameters gives two benefits:

- Efficiently search the space of possible hyperparameters
- Easy to manage a large set of experiments for hyperparameter tuning



Choosing appropriate hyperparameters plays a crucial role in the success of our neural network architecture. Since it makes a huge impact on the learned model. For example, if the learning rate is too low, the model will miss the important patterns in the data. If it is high, it may have collisions.

TYPES OF OPTIMIZATION ALGORITHMS

The process of finding most optimal hyperparameters in machine learning is called hyperparameter optimization.

Some of them have been implemented in our project:

❖ Manual Search

When using Manual Search, we choose some model hyperparameters based on our judgment/experience. We then train the model, evaluate its accuracy and start the process again. This loop is repeated until a satisfactory accuracy is scored.

Implementation:

```
### Manual Hyperparameter Tuning
model=RandomForestClassifier(n_estimators=300,criterion='entropy',
                             max_features='sqrt',min_samples_leaf=10,random_state=100).fit(X_train,y_train)
```

❖ Grid Search

The search space of each hyper-parameter is discretized, and the total search space is discretized as the Cartesian products of them. Then, the algorithm launches a learning for each of the hyper-parameter configurations, and selects the best at the end.

Implementation:

```
### GridCV
param_grid = {
    'criterion': [rf_randomcv.best_params_['criterion']],
    'max_depth': [rf_randomcv.best_params_['max_depth']],
    'max_features': [rf_randomcv.best_params_['max_features']],
    'min_samples_leaf': [rf_randomcv.best_params_['min_samples_leaf'],
                         rf_randomcv.best_params_['min_samples_leaf']+2,
                         rf_randomcv.best_params_['min_samples_leaf'] + 4],
    'min_samples_split': [rf_randomcv.best_params_['min_samples_split'] - 2,
                          rf_randomcv.best_params_['min_samples_split'] - 1,
                          rf_randomcv.best_params_['min_samples_split'],
                          rf_randomcv.best_params_['min_samples_split'] +1,
                          rf_randomcv.best_params_['min_samples_split'] + 2],
    'n_estimators': [rf_randomcv.best_params_['n_estimators'] - 200, rf_randomcv.best_params_['n_estimators'] - 100,
                     rf_randomcv.best_params_['n_estimators'],
                     rf_randomcv.best_params_['n_estimators'] + 100, rf_randomcv.best_params_['n_estimators'] + 200]
}

#### Fit the grid_search to the data
rf=RandomForestClassifier()
grid_search=GridSearchCV(estimator=rf,param_grid=param_grid,cv=10,n_jobs=-1,verbose=2)
grid_search.fit(X_train,y_train)
```

❖ Random Search

In Random Search, we create a grid of hyperparameters and train/test our model on just some random combination of these hyperparameters. Random search has proven to be particularly effective, especially if the search space is not cubic, i.e. if some hyper-parameters are given a much greater range of variation than others.

Implementation:

```
### Randomized Search Cv
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
max_features = ['auto', 'sqrt','log2']
max_depth = [int(x) for x in np.linspace(10, 1000,10)]
min_samples_split = [2, 5, 10,14]
min_samples_leaf = [1, 2, 4,6,8]
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
              'criterion':['entropy','gini']}

rf=RandomForestClassifier()
rf_randomcv=RandomizedSearchCV(estimator=rf,param_distributions=random_grid,n_iter=50,cv=2,verbose=2,
                               random_state=100,n_jobs=-1)
### fit the randomized model
rf_randomcv.fit(X_train,y_train)
```

❖Sequential Model-Based Optimization

A surrogate model of the validation loss as a function of the hyper-parameters can be fit to the previous tries, and tell where the local minimum might be. These methods are called Sequential Model-Based Optimization (SMBO).

➢ Bayesian Optimization (HyperOpt)

Hyperopt is a Python library for performing automated model tuning through SMBO. Bayesian optimization uses probability to find the minimum of a function. The final aim is to find the input value to a function which can give us the lowest possible output value.It usually performs better than random,grid and manual search providing better performance in the testing phase and reduced optimization time.

In Hyperopt, Bayesian Optimization can be implemented giving 3 three main parameters to the function fmin:

- **Objective Function:** defines the loss function to minimize
- **Domain Space:** defines the range of input values to test (in Bayesian Optimization this space creates a probability distribution for each of the used Hyperparameters)
- **Optimization Algorithm:** defines the search algorithm to use to select the best input values to use in each new iteration

Implementation:

```
def objective(space):
    model = RandomForestClassifier(criterion = space['criterion'], max_depth = space['max_depth'],
                                   max_features = space['max_features'],
                                   min_samples_leaf = space['min_samples_leaf'],
                                   min_samples_split = space['min_samples_split'],
                                   n_estimators = space['n_estimators'],
                                   )

    accuracy = cross_val_score(model, X_train, y_train, cv = 5).mean()

    return {'loss': -accuracy, 'status': STATUS_OK }
```

```
### Bayesian Optimization Algorithm
space = {'criterion': hp.choice('criterion', ['entropy', 'gini']),
        'max_depth': hp.quniform('max_depth', 10, 1200, 10),
        'max_features': hp.choice('max_features', ['auto', 'sqrt','log2', None]),
        'min_samples_leaf': hp.uniform('min_samples_leaf', 0, 0.5),
        'min_samples_split' : hp.uniform ('min_samples_split', 0, 1),
        'n_estimators' : hp.choice('n_estimators', [10, 50, 300, 750, 1200,1300,1500])
    }
```

```
trials = Trials()
best = fmin(fn= objective,
            space= space,
            algo= tpe.suggest,
            max_evals = 80,
            trials= trials)

crit = {0: 'entropy', 1: 'gini'}
feat = {0: 'auto', 1: 'sqrt', 2: 'log2', 3: None}
est = {0: 10, 1: 50, 2: 300, 3: 750, 4: 1200,5:1300,6:1500}


trainedforest = RandomForestClassifier(criterion = crit[best['criterion']], max_depth = best['max_depth'],
                                       max_features = feat[best['max_features']],
                                       min_samples_leaf = best['min_samples_leaf'],
                                       min_samples_split = best['min_samples_split'],
                                       n_estimators = est[best['n_estimators']]).fit(X_train,y_train)
```

❖ Automated Hyper Parameter Tuning (using Optuna)

The hyperparameters of the above algorithm are n_estimators and max_depth for which we can try different values to see if the model accuracy can be improved. The objective function is modified to accept a trial object. This trial has several methods for sampling hyperparameters. We create a study to run the hyperparameter optimization and finally read the best hyperparameters.

Implementation:

```
### Optuna Automated Hyper Parameter Tuning
def objective1(trial):
    classifier = trial.suggest_categorical('classifier', ['RandomForest'])

    if classifier == 'RandomForest':
        n_estimators = trial.suggest_int('n_estimators', 200, 2000,10)
        max_depth = int(trial.suggest_float('max_depth', 10, 100, log=True))

        clf = ske.RandomForestClassifier(
            n_estimators=n_estimators, max_depth=max_depth)

    return sklearn.model_selection.cross_val_score(clf,X_train,y_train, n_jobs=-1, cv=3).mean()
```

```
study = optuna.create_study(direction='maximize')
study.optimize(objective1, n_trials=100)

trial = study.best_trial
study.best_params

rf=RandomForestClassifier(n_estimators=330,max_depth=53)
rf.fit(X_train,y_train)
```

❖ **Hyperband**

The underlying principle of this algorithm is that if a hyperparameter configuration is destined to be the best after a large number of iterations, it is more likely to perform in the top half of configurations after a small number of iterations.

Below is a step-by-step implementation of Hyperband:
Randomly sample n number of hyperparameter sets in the search space. After k iterations evaluate the validation loss of these hyperparameters. Discard the half of lowest performing hyperparameters. Run the good ones for k iterations more and evaluate and discard the bottom half. Repeat until we have only one model of hyperparameter left.

❖ **Genetic Algorithms (TPOT Classifier)**

Genetic Algorithms tries to apply natural selection mechanisms to Machine Learning contexts. Let's imagine we create a population of N Machine Learning models with some predefined Hyperparameters. We can then calculate the accuracy of each model and decide to keep just half of the models (the ones that perform best). We can now

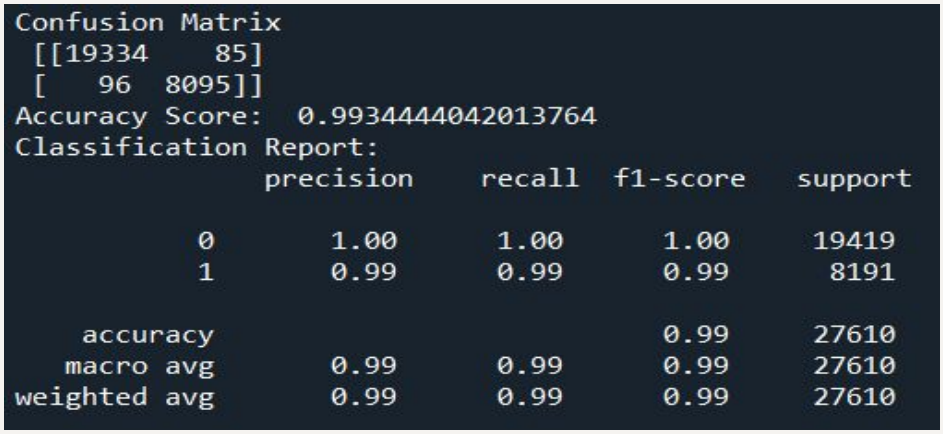
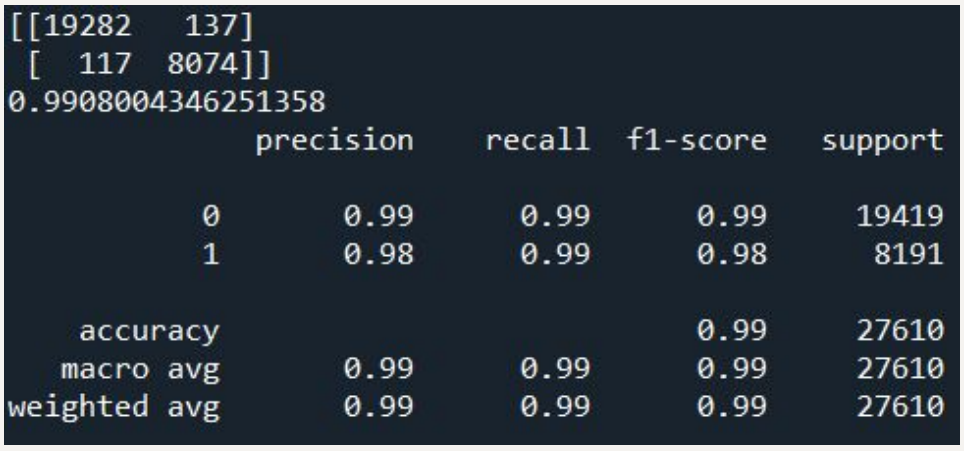
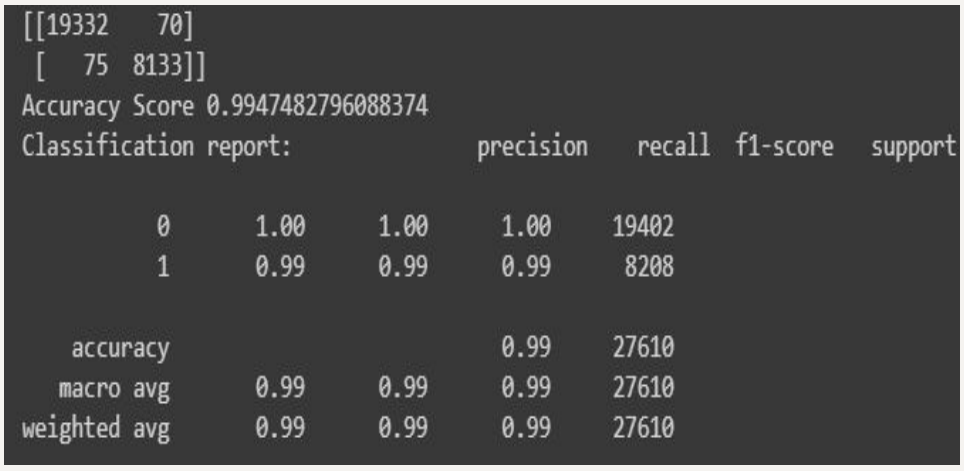
generate some offspring having similar Hyperparameters to the ones of the best models so that we get again a population of N models. At this point we can again calculate the accuracy of each model and repeat the cycle for a defined number of generations. In this way, just the best models will survive at the end of the process.

```
### Genetic Algorithm --> TPOT Classifier
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
max_features = ['auto', 'sqrt','log2']
max_depth = [int(x) for x in np.linspace(10, 1000,10)]
min_samples_split = [2, 5, 10,14]
min_samples_leaf = [1, 2, 4,6,8]

param = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
              'criterion':['entropy','gini']}

tpot_classifier = TPOTClassifier(generations= 5, population_size= 24, offspring_size= 12,
                                 verbosity= 2, early_stop= 12,
                                 config_dict={'sklearn.ensemble.RandomForestClassifier': param},
                                 cv = 4, scoring = 'accuracy')
tpot_classifier.fit(X_train,y_train)
```

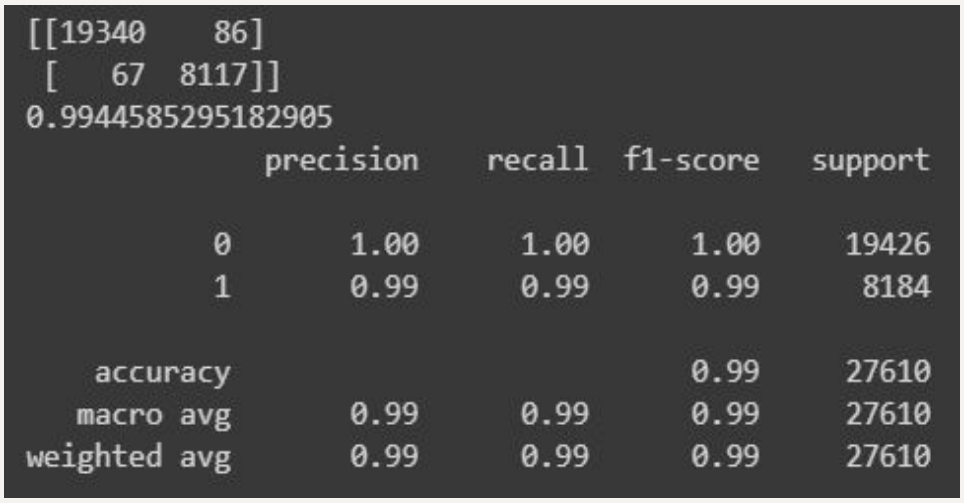
EVALUATION AND RESULTS

Evaluation metrics for various optimization algorithms:

```
## Evaluation (No tuning) ------------> (1)
prediction=rf_classifier.predict(X_test)
print('Confusion Matrix\n', confusion_matrix(y_test,prediction))
print('Accuracy Score: ', accuracy_score(y_test,prediction))
print('Classification Report:\n',classification_report(y_test,prediction))
```

1. Without Hyper-Parameter Tuning	
2. Manual Hyper-Parameter Tuning	
3. Randomized Search Algorithm	

4. Grid Search Algorithm	[[19333 69] [76 8132]] Accuracy Score 0.9947482796088374 Classification report: precision recall f1-score support 0 1.00 1.00 1.00 19402 1 0.99 0.99 0.99 8208 accuracy 0.99 27610 macro avg 0.99 0.99 0.99 27610 weighted avg 0.99 0.99 0.99 27610
5. Bayesian Optimization using HyperOpt	[[19126 226] [131 8127]] 0.9870699022093444 precision recall f1-score support 0 0.99 0.99 0.99 19352 1 0.97 0.98 0.98 8258 accuracy 0.99 27610 macro avg 0.98 0.99 0.98 27610 weighted avg 0.99 0.99 0.99 27610
6. Automated optimization with Optuna	[[19138 89] [67 8316]] 0.9943498732343354 precision recall f1-score support 0 1.00 1.00 1.00 19227 1 0.99 0.99 0.99 8383 accuracy 0.99 27610 macro avg 0.99 0.99 0.99 27610 weighted avg 0.99 0.99 0.99 27610

7. **Genetic ALgorithm (TPOT Classifier)**	

```
[[19340    86]
 [   67  8117]]
0.9944585295182905
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     19426
           1       0.99      0.99      0.99      8184

    accuracy                           0.99     27610
   macro avg       0.99      0.99      0.99     27610
weighted avg       0.99      0.99      0.99     27610
```

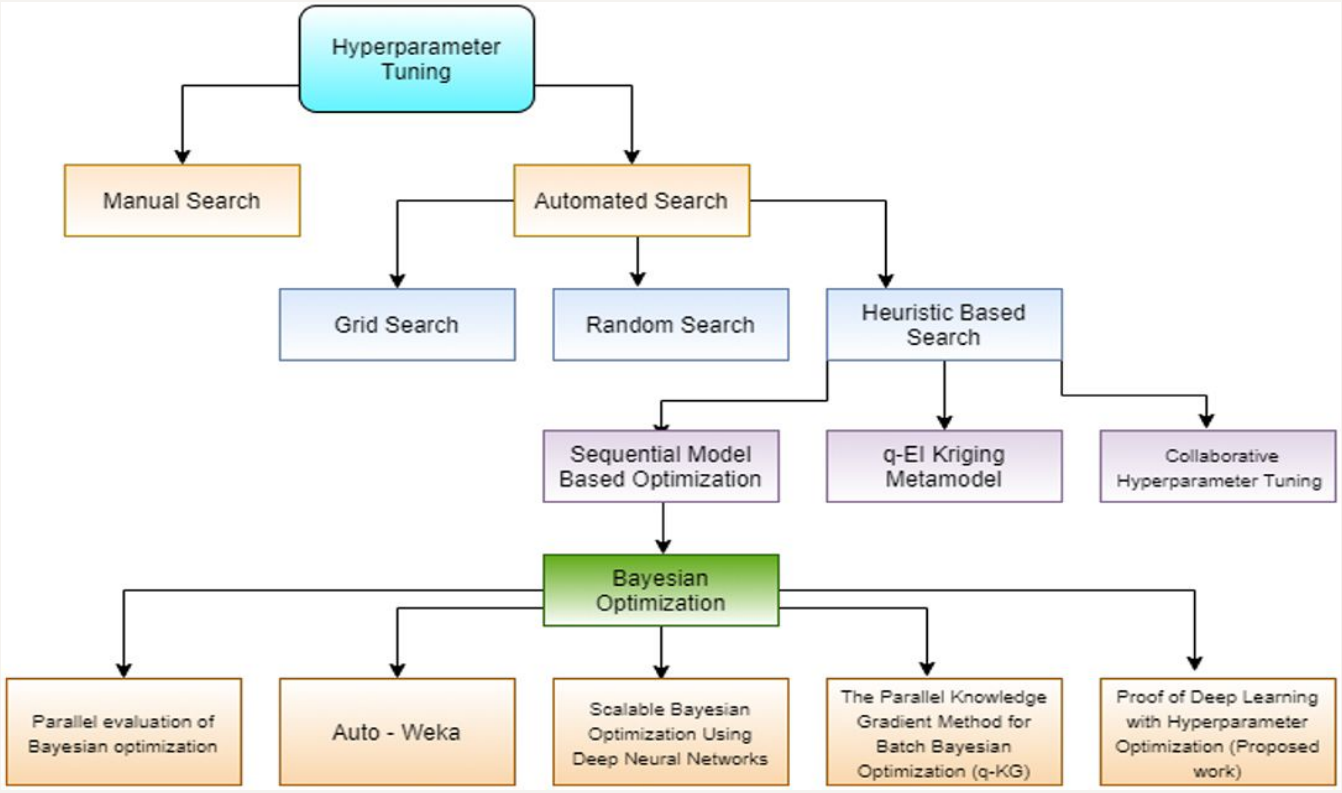
Code Repository:

The code snippets have already been provided in the report. For the entire code, refer to SwarmProject Repo. Steps to run the code have been updated in readme.

ANALYSIS OF HYPER-PARAMETER OPTIMIZATION ALGORITHMS

Brief summary about the pros and cons of each optimization algorithms are presented below which have found out on application of these algorithms on our problem:

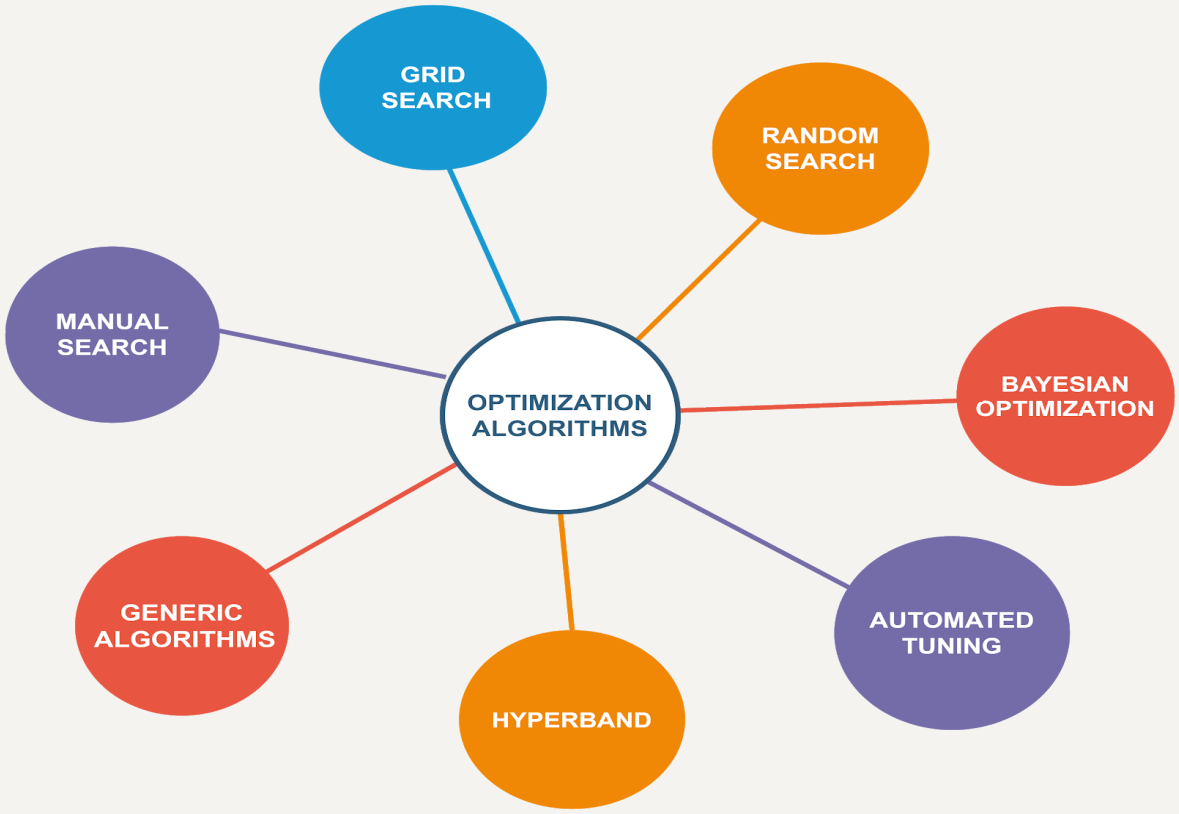
- **Manual Search:** It helps to learn more in depth about the behaviour of hyperparameters but Manual works are required and one may overthink about the unexpected movement of the score without trying many and checking if it was a generalized movement.



- **Random Search:** As the number of parameter searches are controlled so the run time is also controlled. There should be some compromise that the finally selected hyperparameter set might not be the true best out of the ranges one puts in the search. Depending on the number of searches and how large the parameter space is, some parameters might not be explored enough.

- **Grid Search:** All possible prospective sets of parameters are covered. So we get the most viable solution using this solution. The run time of the whole parameter sets can be huge, and therefore the number of parameters to explore has practical limitations.

- **Sequential Model Based Optimization (Bayesian Optimization):** It offers the appealing prospects of interpolating performance between observed parameter settings and of extrapolating to previously unseen regions of parameter space. Initial Experimental designs are very costly. It only supports numerical parameters and only optimizes target algorithm performance for single instances.



- **Automated Hyperparameter Tuning:** Manual work is negligible as the whole process is automated. Time taken is considerably less compared to other algorithms. The results from this are not that satisfactory as compared to other algorithms.

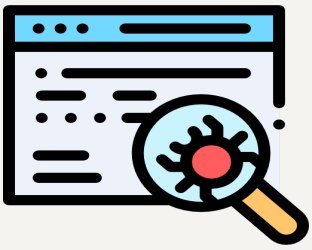
- **Hyperband:** Deciding when to stop the training of models depends on how strongly the training data affects the score. Hyperband balances these two extremes hence it has significant parallelism. The most obvious opportunity involves job prioritization. Hyperband fits many models in parallel. If the number of samples is large some good performing hyperparameter sets which require some time to converge may be discarded early in the optimization.

- **Genetic Algorithms:** It searches from a population of points not a single point. It is good for noisy environments as it is robust to maxima and minima. The computational time is very large. Designing an objective function and getting the operators and representation right is difficult.

EXPECTED CONTRIBUTION (RESULTS)

This project can be mainly divided into 2 sub-sections:

a.) Malware Detection from Portable Executable Binary Files:



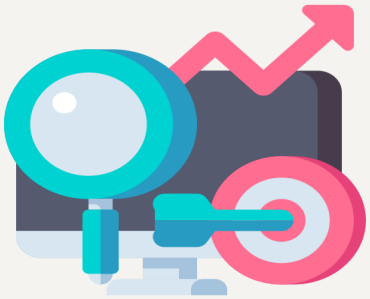
The exponential increase in the creation of new malware in the last five years made malware detection as a challenging research issue. Detection of malicious PE files from benign is very important as PE file format is the highly used file format, used in Windows OS.

Contributions of the proposed work are as listed below:

- Our work has provided a novel derived feature engineering technique that improves the performance of machine learning based classifiers for malicious PE file detection.
- We used static analysis techniques to extract the features which have lower time and resource requirements than dynamic analysis.
- An integrated feature set is created, which has raw and derived features, based on different header fields' values of PE file. The proposed integrated feature set improves the classification accuracy of machine learning classifiers.
- It also provides empirical evidence for usability of different header's fields' value of PE file as useful discriminative features.

We have also compared the performance of different machine learning algorithms and found that the performance of classifiers improved significantly with the new integrated feature set. 13 extracted and processed raw & integrated feature sets are also available for comparing future work with the proposed work. The dataset will help future researchers of this field to take research on the further level by having such dataset in advance.

b.) Analysis of Hyper-Parameter Optimization Algorithms:



It is important to determine how our model is structured in the first place. Hence, hyperparameters are extremely important for any machine learning algorithm since they directly control the behaviors of training algorithms and have a significant effect on the performance of machine learning models.

In this project, we have studied several optimization algorithms that have been developed and then, successfully applied them to a real-life application domain.

Major advantages of performing this analysis:

- This will improve the performance of machine learning algorithms (by tailoring them to the problem at hand).
- We considered building the relationship between the performance of the machine learning models and their hyperparameters.
- Our work provides a review of the most essential topics on HPO. The first one being the introduction of key hyper-parameters related to model training and structure, and then discussion regarding their importance and methods to define the value range.
- This will improve the reproducibility and fairness of scientific studies. It facilitates fair comparisons since different methods can only be compared fairly if they all receive the same level of tuning for the problem at hand.
- Our work also implemented automated tuning algorithms, comparing their support for state-of-the-art searching algorithms.
- We also provided a comparison between optimization algorithms, and prominent approaches for model evaluation with limited computational resources.

Through this project, we aimed at improving the performance of Machine Learning models by providing a fair comparison between different optimization algorithms and their effect on evaluation metrics. This work can in turn help the future developers to understand which one of them will be best for their application domain.

REFERENCES

Hyperparameter Optimization: Automated Machine Learning

(PDF) Algorithms for Hyper-Parameter Optimization

[1912.06059] Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS

Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization

[2003.05689] Hyper-Parameter Optimization: A Review of Algorithms and Applications

Malware Detection PE-Based Analysis Using Deep Learning Algorithm Dataset

https://virusshare.com/

http://malicia-project.com/

https://github.com/fmfn/BayesianOptimization

https://github.com/hyperopt/hyperopt

https://www.jeremyjordan.me/hyperparameter-tuning/

https://download.cnet.com/windows/

https://www.virustotal.com/

https://optuna.org/

https://scikit-optimize.github.io/stable/auto_examples/hyperparameter-optimization.html