Data Mining Techniques on Algorithm Analysis for Parallelized Compliance Graph Generation

Final Project Report by:
Noah L. Schrick

On Behalf of:
Dr. Ismail Abdulrashid

Collins College of Business
The University of Tulsa
Tulsa, OK, USA

Abstract (150 words)
Large-scale attack and compliance graphs can be used for detecting, preventing, and correcting cybersecurity or compliance violations with a system or set of systems. However, as modern-day networks expand in size, and as the number of possible exploits and regulation mandates increase, large-scale attack and compliance graphs can seldom be feasibly generated through serial means. Recent works present a parallelized generation process that leverages Message-Passing Interface (MPI) for distributed computing. The approach was deployed on a High-Performance Computing (HPC) system where a large amount of performance data was collected to capture and conduct a comprehensive analysis on the approach. This work analyzes the data through a series of testing. This work provides insight on parameter impact to runtime, contributions of parameters, and insight into the effectiveness of the algorithm. Additional insight is gathered regarding future contributions to be made for algorithmic improvements based on the successes identified with this approach.

Table of Contents

List of Figures and Tables

Section I: Introduction

Compliance graphs are generated graphs (or networks) that represent systems' compliance or regulation standings at present, with expected changes, or both. These graphs are generated as directed acyclic graphs (DAGs), and can be used to identify possible correction or mitigation schemes for environments necessitating compliance to mandates or regulations.

Compliance graphs are an appealing approach since they are often designed to be exhaustive: all system properties are represented at its initial state, all violation options are fully enumerated, all permutations are examined, and all changes to a system are encoded into their own independent states, where these states are then individually analyzed through the process. Despite their advantages, compliance graphs do suffer from their exhaustiveness as well. As the authors of [1] examine, even very small networks with only 10 hosts and 5 vulnerabilities yield graphs with 10 million edges. When scaling compliance graphs to analyze the modern, interconnected state of large networks comprising of a multitude of hosts, and utilizing the entries located in the National Vulnerability Database and any custom vulnerability or regulation testing, graph generation quickly becomes infeasible. Similar difficulties arise in related fields, where social networks, bioinformatics, and neural network representations result in graphs with millions of states [2]. This state space explosion is a natural by-product of the graph generation process, and removing or avoiding it entirely undermines the overall goal of compliance graphs.

Related research works such as those seen by the authors of [2], [4], [5], [6], and [7] extend the generation process to function on distributed computing environments to take advantage of the increased computing power using message-passing. The approach utilized for this work is novel, and makes use of a task parallelism approach for the generation process, and examines the effect of various parameter tuning and algorithm design choices and their effects on speedup and efficiency.

This work aims to analyze the timing data obtained through the deployment of the new MPI tasking algorithm. By analyzing this data, an answer will be sought out for the following questions:

1. How does each parameter contribute to the overall runtime?
2. Which parameters have the largest effect on overall runtime?
3. Can an optimal set or subset of parameters with corresponding values be identified?
4. What is the best way to visualize and represent the data and findings?

Analyzing this data will provide further insight into the effectiveness of the algorithm, and allow future contributions to be made for algorithmic improvements based on the successes identified with this approach.

Section II: Background and Related Works

There have been numerous approaches at generation improvement specific to attack graphs. As a means of improving scalability of attack graphs, the authors of [1] present a new representation scheme. Traditional attack graphs encode the entire network at each state, but the representation presented by the authors uses logical statements to represent a portion of the network at each node. This is called a logical attack graph. This approach led to the reduction of the generation process to quadratic time and

reduced the number of nodes in the resulting graph to O(n2). However, this approach does require more analysis for identifying attack vectors.
Another approach presented by the authors of [8] represents a description of systems and their qualities and topologies as a state, with a queue of unexplored states.
This work was continued by the authors of [9] by implementing a hash table among other features. Each of these works demonstrates an improvement in scalability through refining the desirable information output.

Another approach for generation improvement is through parallelization. The authors of [9] leverage OpenMP to parallelize the exploration of a FIFO queue. This parallelization also includes the utilization of OpenMP's dynamic scheduling. In this approach, each thread receives a state to explore, where a critical section is employed to handle the atomic functions of merging new state information while avoiding collisions, race conditions, or stale data usage. The authors measured a 10x speedup over the serial algorithm.
The authors of [6] present a parallel generation approach using CUDA, where speedup is obtained through a large number of CUDA cores.
For a distributed approach, the authors of [7] present a technique for utilizing reachability hyper-graph partitioning and a virtual shared memory abstraction to prevent duplicate work by multiple nodes. This work had promising results in terms of speedup and in limiting the state-space explosion as the number of network hosts increases.

Section III: Data Analysis Approach

Section III.A: Data Description

The algorithmic approach is to break the generation process into six main tasks. Dependent variables of the data are time per task and overall runtime. To measure the speedup and efficiency, data is collected for each task based on parameter tuning, as well as data for the overall process. Reducing the dependent variables, it can be shown that there is one dependent variable (overall runtime) that is comprised of 6 sub-dependent variables (time per task).

Across the six main tasks and throughout the generation process, a total of 4 additional independent variables are used. These independent variables are: the number of compute nodes used, the number of exploits, the number of applicable exploits, and the memory load before database operations are performed. Each variable is changed individually across a given range to capture a full-spectrum image of their effects on time.

The number of nodes ranges from 2 through 12, incrementing by 1.
The number of exploits ranges from 6 to 49152, incrementing by a factor of x2.
The number of applicable exploits ranges from 0% to 100%, incrementing by 25%.
The database load ranges from 0% to 100%, incrementing by 25%.

The dataset therefore has 2,464 entries, with each entry spread across 11 columns, leading to a total of 27,104 individual pieces of data. The data will be collected in an automated fashion. The experiment will be deployed on the University's supercomputer cluster, with custom scripting to automatically run each subsequent test, and collect the data into a raw, unfiltered CSV.

Since this approach is novel, this data is not public, and has not been analyzed by any other individuals or groups.

Section III.B: Exploratory Data Analysis

Exploratory Data Analysis (EDA) was first performed to obtain preliminary information regarding the data. In order to determine viable analysis techniques, the Variance Inflation Factor (VIF) was computed for the four main parameters (nodes, exploits, applicability of exploits, and database load). This resulted in the following results, shown in Table 1:

Table 1: Feature VIF

Feature	**VIF**
Number of Nodes	4.095
Number of Exploits	1.102
Applicability of exploits	2.482
Database Load	5.401

The results of the VIF computations reveal that though the Database Load is moderately high at 5.4, there is not great concern for muilticolinearity. As a result, no additional work is required to address or correct multicolinearity or Type II errors.

Prior to normalizing or scaling data, each feature was plotted with respect to runtime. By purposefully not normalizing or scaling, it allows for insight into the underlying shape of the data. The following four figures depict the plots of the raw data.

Figure 1: Non-Normalized and Non-Scaled Plot of Node Feature vs Runtime

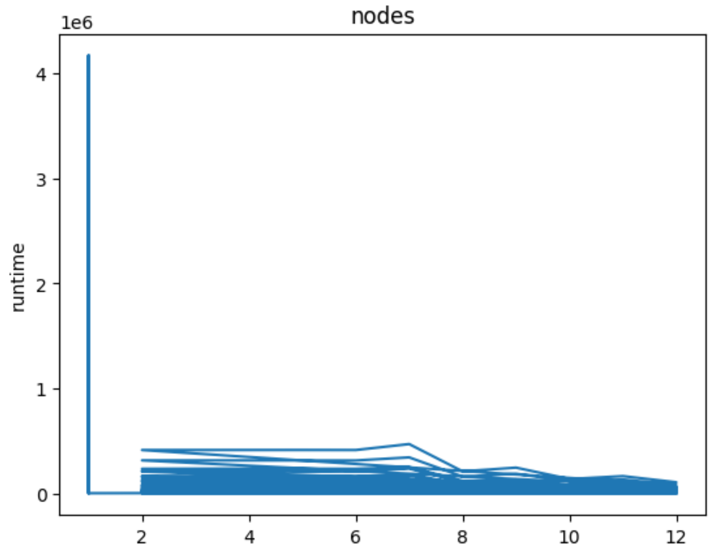


Figure 2: Non-Normalized and Non-Scaled Plot of Exploit Feature vs Runtime

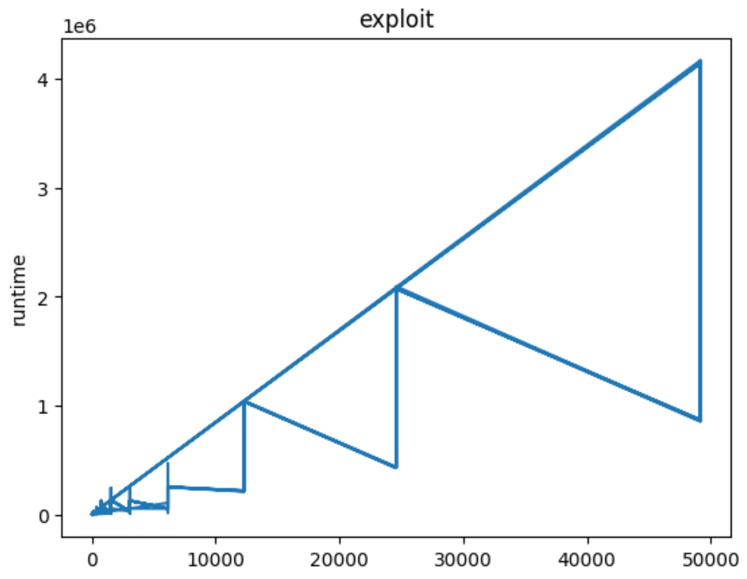


Figure 3: Non-Normalized and Non-Scaled Plot of Applicability Feature vs Runtime

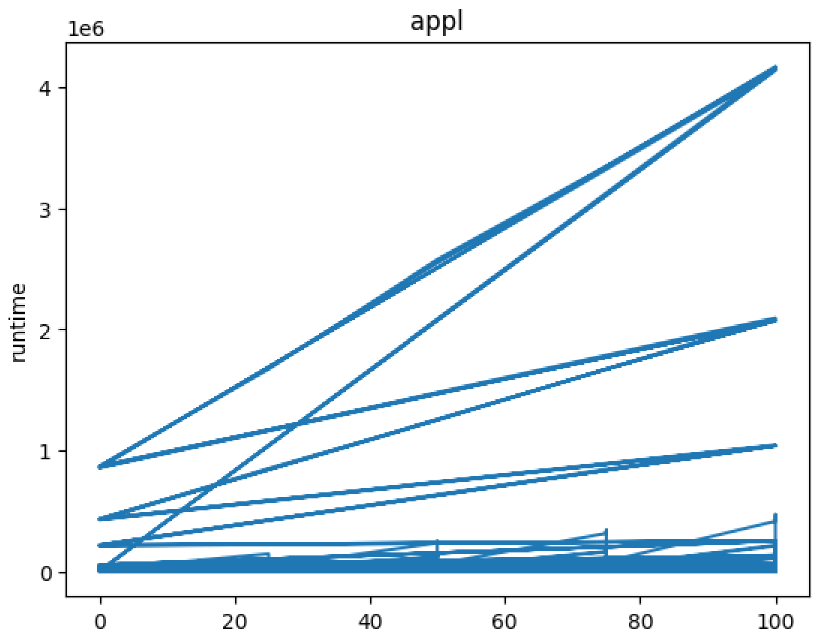
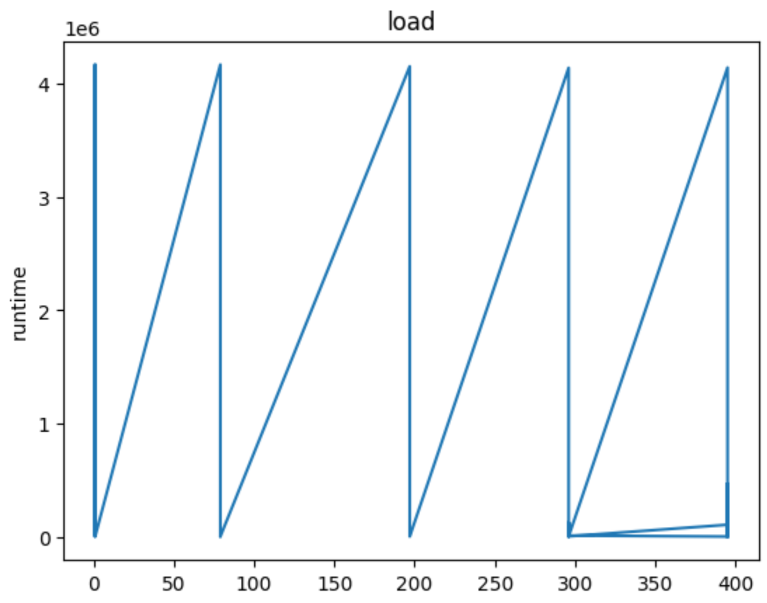


Figure 4: Non-Normalized and Non-Scaled Plot of Database Load Feature vs Runtime



These plots highlight the nonlinear and unpredictable trends of the data. Due to this, normalization or scaling will be necessary. Applying a learning technique or a regression technique will not reveal promising outcomes, since these plots indicate that the shape of the data will be hard to match or predict. As a result, normalization was applied, and the following four plots depict the normalized features versus the runtime.

Figure 5: Normalized Plot of Nodes Feature vs Runtime

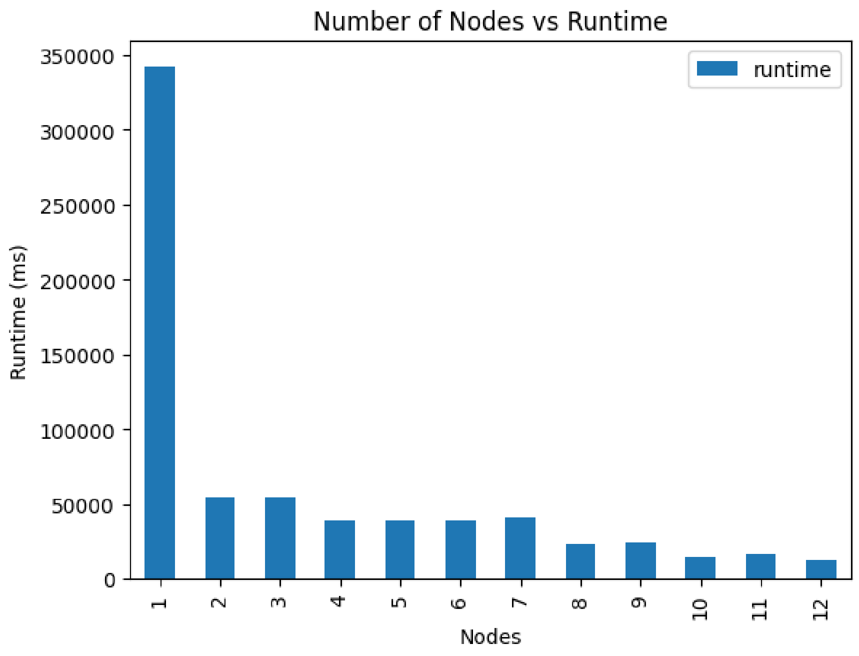


Figure 6: Normalized Plot of Exploits Feature vs Runtime

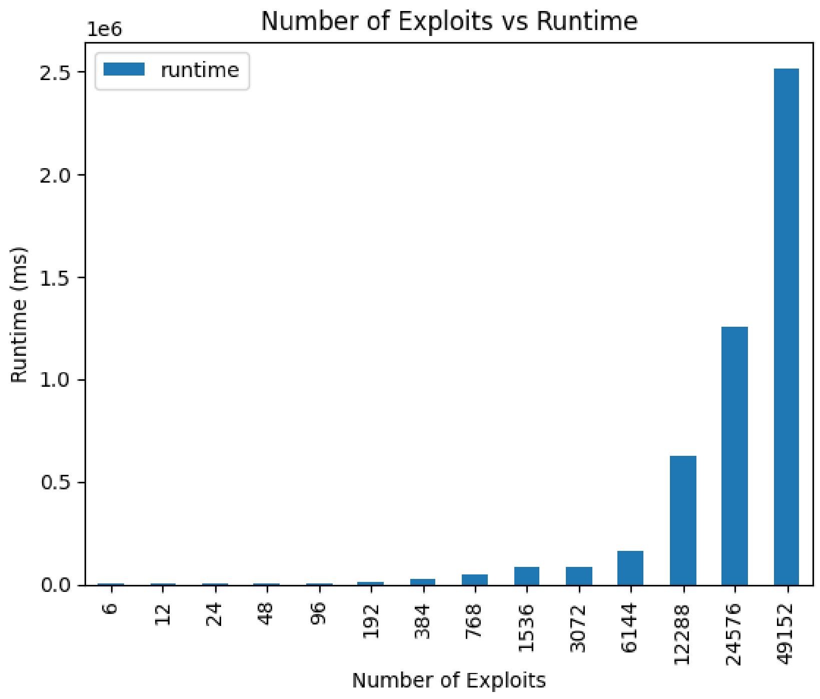


Figure 7: Normalized Plot of Applicability Feature vs Runtime

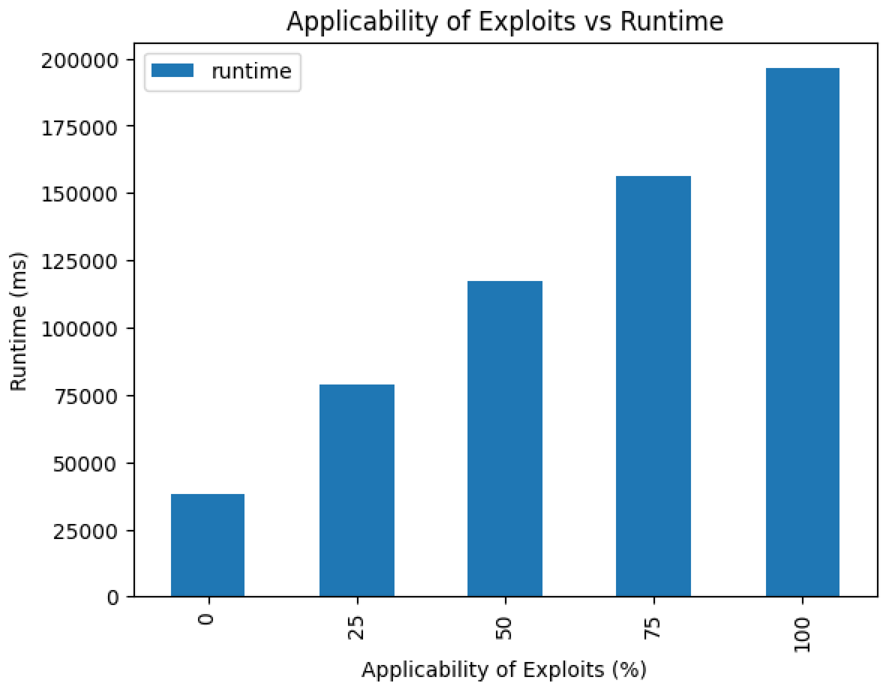
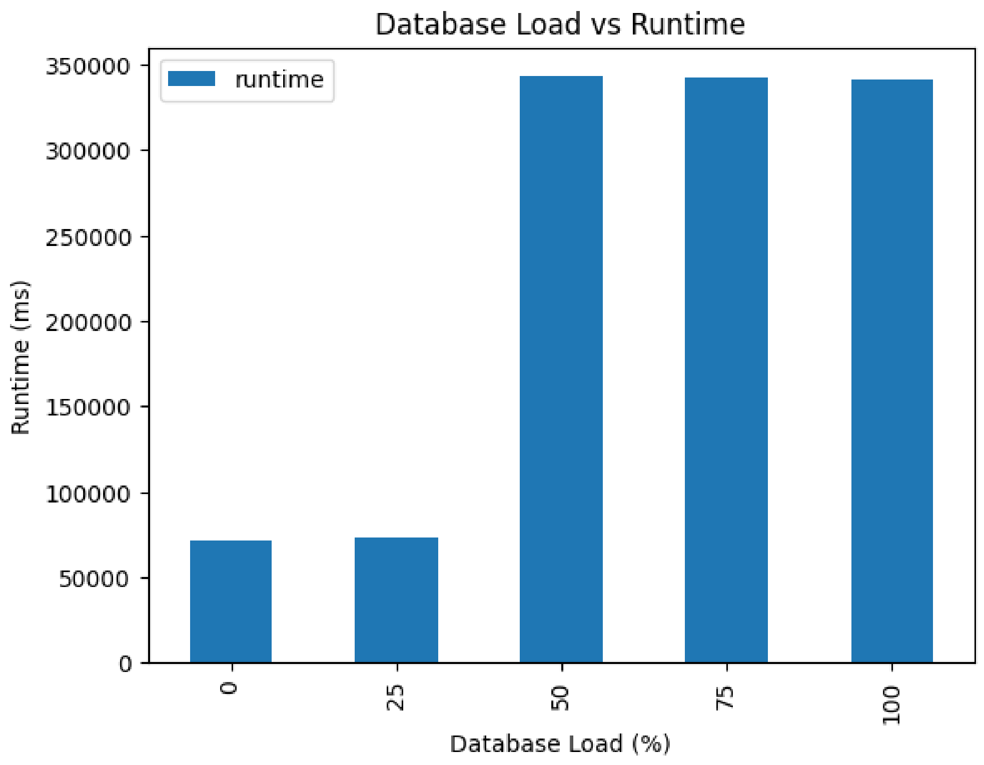


Figure 8: Normalized Plot of Database Load Feature vs Runtime



These four figures, compared to Figures 1-4, represent much simpler data trends. When normalizing the data, it becomes much more likely to apply a learning or regression technique to match and predict future data. For the following sections, normalized data will be used.

Section III.C: Speedup and Efficiency

One metric in which the data was analyzed was through speedup and efficiency. Speedup measures the performance increase obtained by parallelization compared to the sequential runtime. Speedup, as defined by Amdahl, is as follows:

$$Speedup\ on\ N\ nodes = \frac{sequential\ execution\ time}{execution\ time\ on\ N\ nodes} \qquad (1)$$

Efficiency measures how well the additional nodes were used. Efficiency is a value between 0 and 1. The ideal parallelization is when each node is able to fully reduce the runtime by half. Efficiency is defined as follows:

$$Efficiency = \frac{speedup}{N} \qquad (2)$$

To measure the algorithm in terms of speedup and efficiency, new columns were added to the data. The values in each column were filled by looping through the data frame, and dividing each row by the equivalent row where all parameters were equal to the loop row, except the node parameter was 1. Efficiency computations were performed similarly, except the speedup column was used. After the columns were filled, pivot tables were used to create plots for the minimum, maximum, and mean speedups and efficiencies across the node feature. The following two figures display the results.

Figure 9: Speedups Across the Node Feature

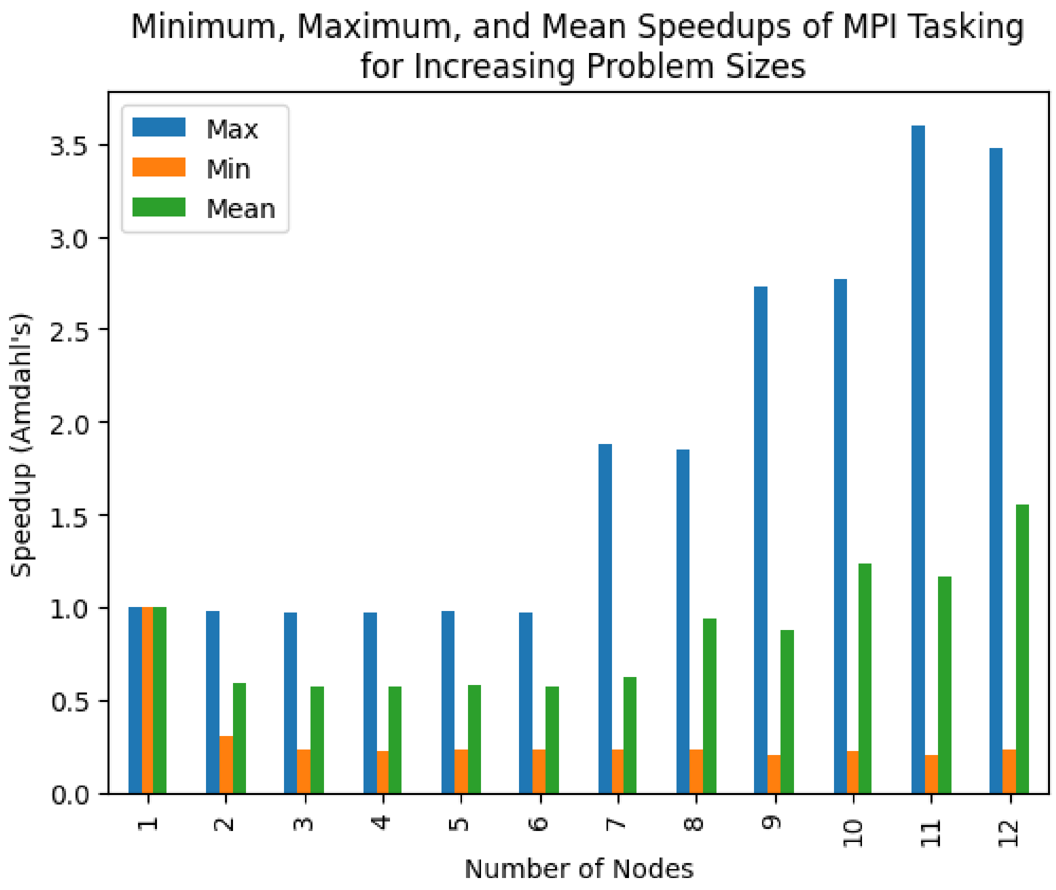
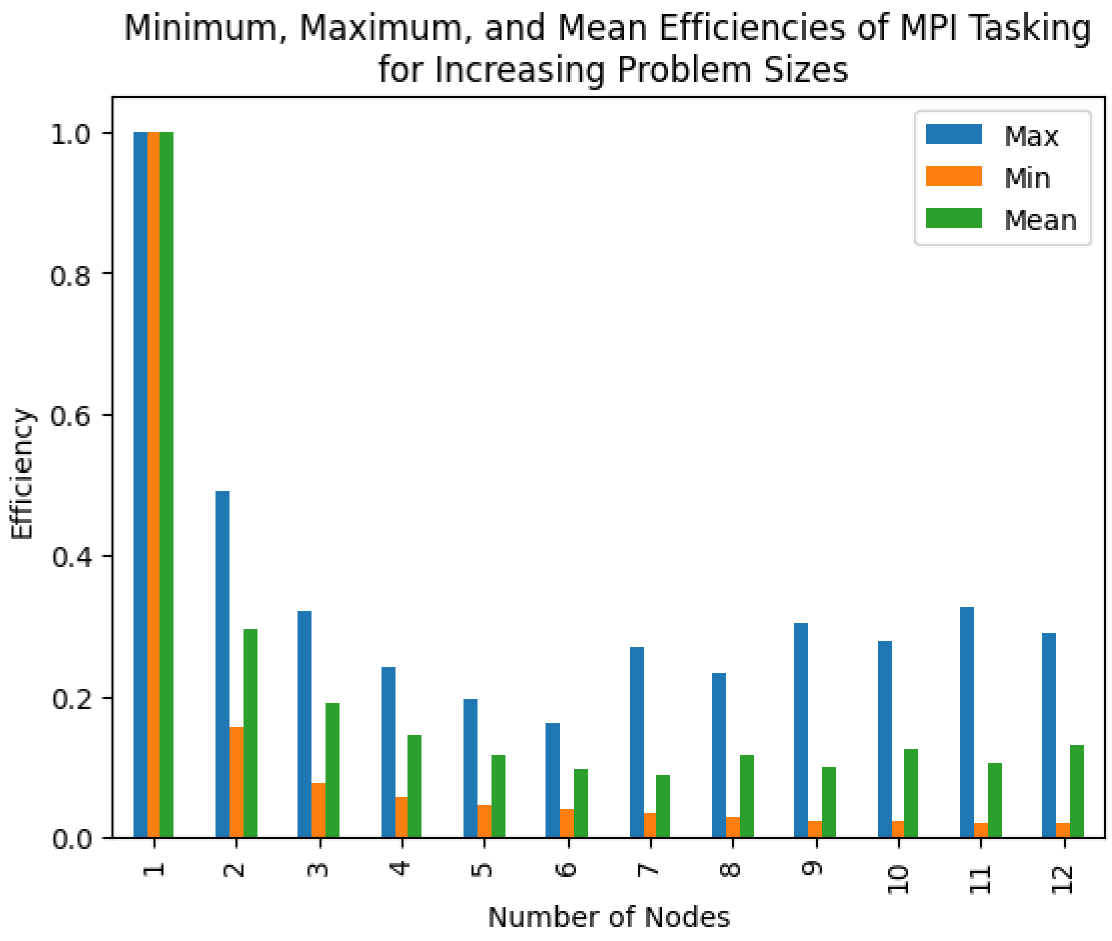


Figure 10: Efficiencies Across the Node Feature



The results from Figures 9 and 10 yield promising information on the performance of the new algorithm.

Section III.D: Linear Regression

1) Per Feature

Linear regression was performed on the normalized data frame for each feature. For each feature, the data was partitioned with a train-test split of 40% test, 60% train. A linear regression model was fit to the training data, and coefficients were obtained. Regression summaries were also obtained for each feature. The results are shown in Table 2 below.

Table 2: Linear Regression Statistics per Feature

Feature / **Regression Statistics**	**Nodes**	**Exploit**	**Applicability**	**Database Load**
Intercept	0.00142	-0.00668	0.00013	-0.00708
Coefficient	-0.255	0.876	0.123	-0.183
Mean Error	0.000	0.000	0.000	0.000
RMSE	0.946	0.437	0.972	0.962
MAE	0.400	0.145	0.376	0.381

2) Multivariate Linear Regression

In addition to performing linear regression on each feature individually, multivariate linear regression analysis was conducted. In this approach, all parameters were set as predictors in the linear regression model. The model was fit to the 60% train data, and results were obtained and are shown below in Tables 3 and 4.

Table 3: Feature Coefficients for Multivariate Linear Regression

Predictor	**Coefficient**
Nodes	-0.025
Exploits	0.877
Applicability	0.129
Database Load	0.031

Table 4: Regression Statistics for Multivariate Linear Regression

Statistic	**Value**
Mean Error	0.000
RMSE	0.417
MAE	0.185

Comparing Tables 2 and 3, it is worth noting that the coefficients obtained from multivariate linear regression vary in similarity to the coefficients obtained from the per-feature linear regression. Table 5 shown below displays the percent difference between the two approaches. The coefficients for the exploit and applicability feature have a low percent difference, whereas the percent differences between node coefficients and database load coefficients are much higher.

Table 5: Percent Differences of Multivariate and Per-Feature Linear Regression Coefficients

Feature	**Per-Feature Coefficient**	**Multivariate Coefficient**	**Percent Difference**
Node	-0.255	-0.025	164.29%
Exploit	0.876	0.877	0.11%
Applicability	0.123	0.129	4.76%
Database Load	-0.183	0.031	200.00%

Section III.E: Nonlinear Regression

In addition to linear regression techniques, nonlinear approaches were performed. By utilizing nonlinear approaches, it is possible to see how they handle the nonlinear aspects of the data. For instance, the database load parameter appears to be well-suited for a nonlinear technique. While some parameters, such as the node parameter, may be better suited for a linear approach, it is considered worthwhile to examine the results obtained from a nonlinear approach. In addition, there may be value from a multivariate nonlinear technique.

1) Random Forest

Random forest regression was performed on a per-feature basis. Figures 11, 12, 13, and 14 display the results obtained from a random forest fit for the node parameter, exploit parameter, applicability parameter, and database load parameter, respectively.

Figure 11: Random Forest Regression for the Node Parameter

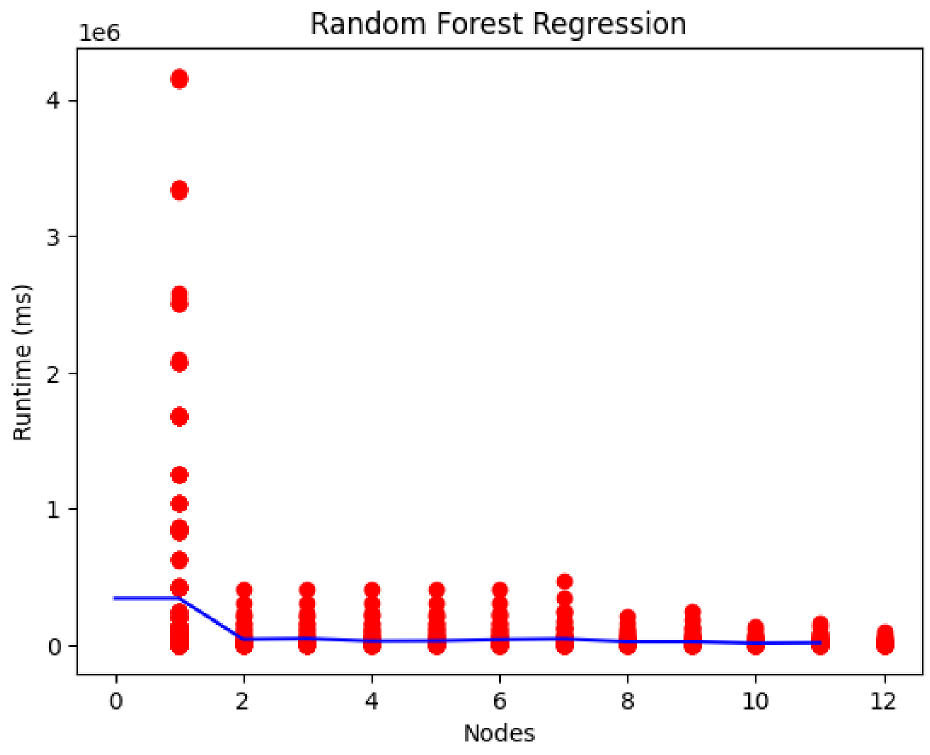


Figure 12: Random Forest Regression for the Exploit Parameter

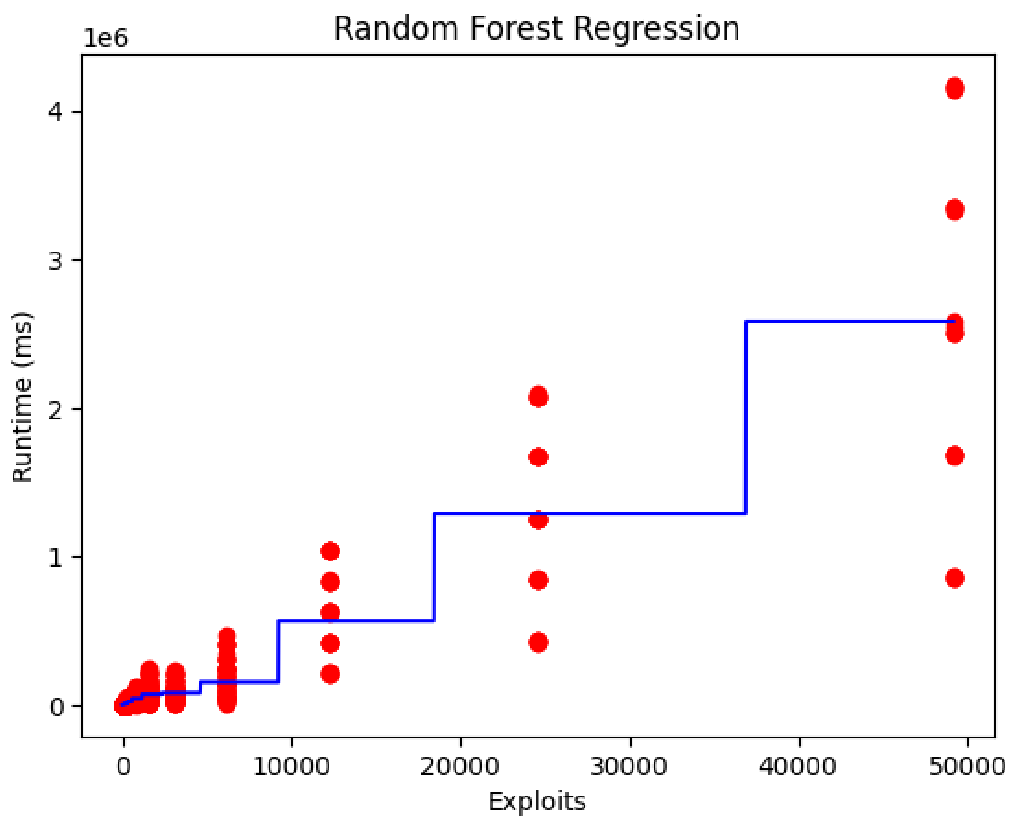


Figure 13: Random Forest Regression for the Applicability Parameter

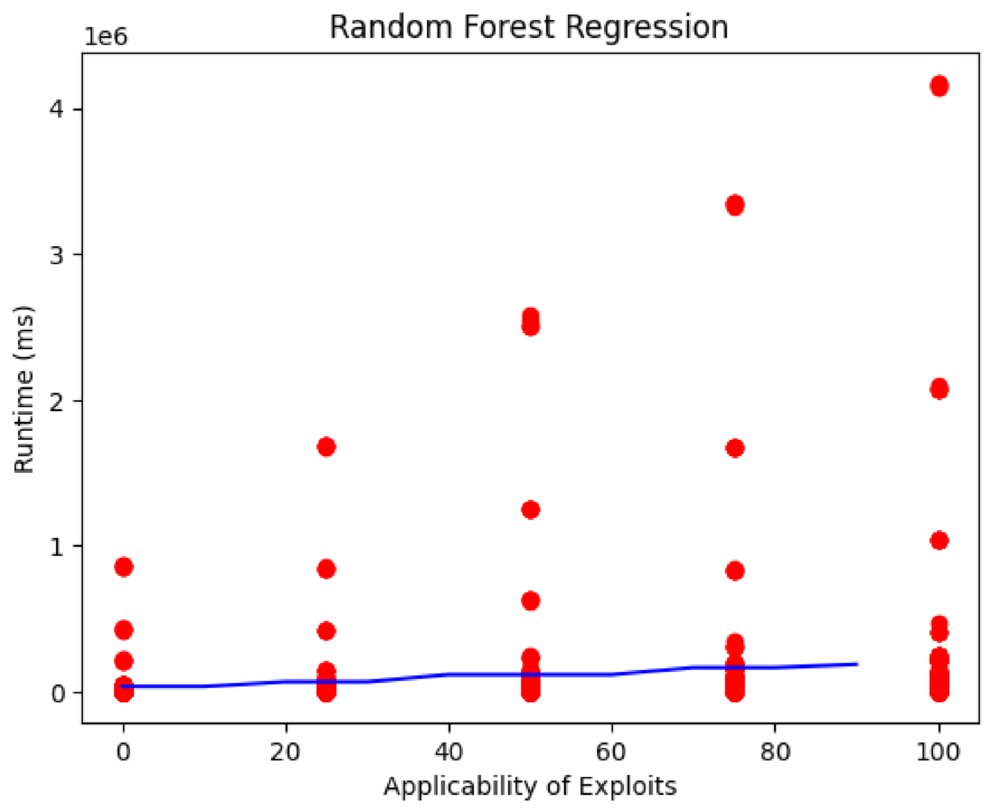
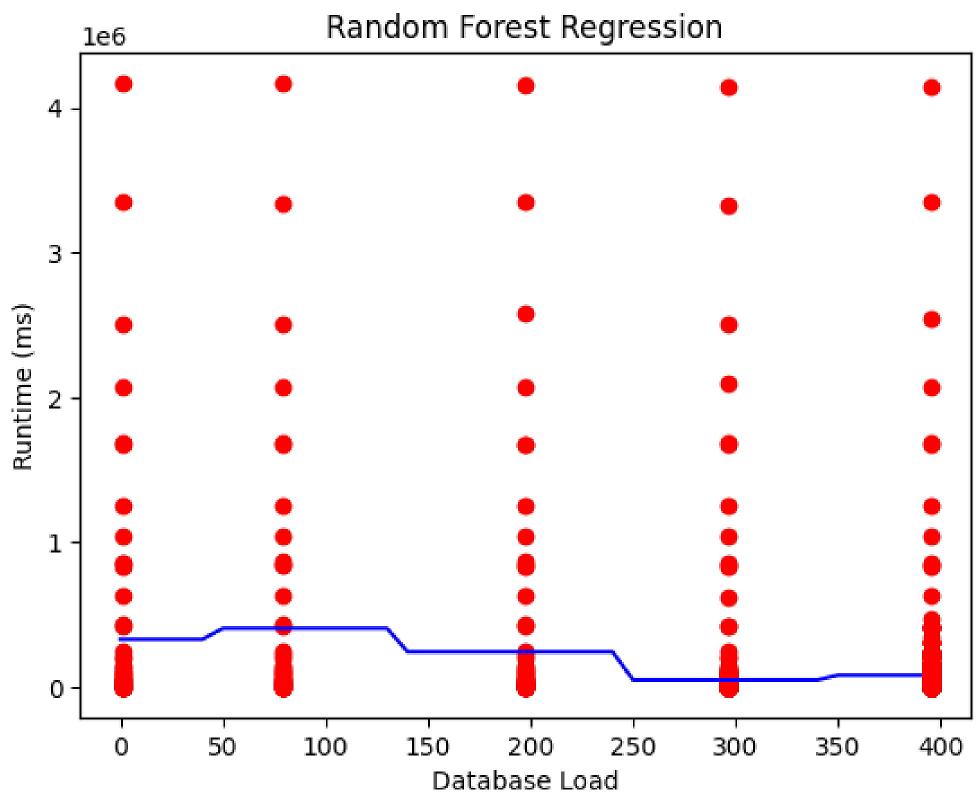


Figure 14: Random Forest Regression for the Applicability Parameter



The results of the random forest regression for the node and exploit parameter indicate an adequate performance. The results of the applicability and database load parameter highlight the difficulties in fitting a model to this data. Though the random forest approach did follow the data trend mean, it did not capture the sporadic nature or multivariate nature of the results.

2) Gradient Boosting Regressor

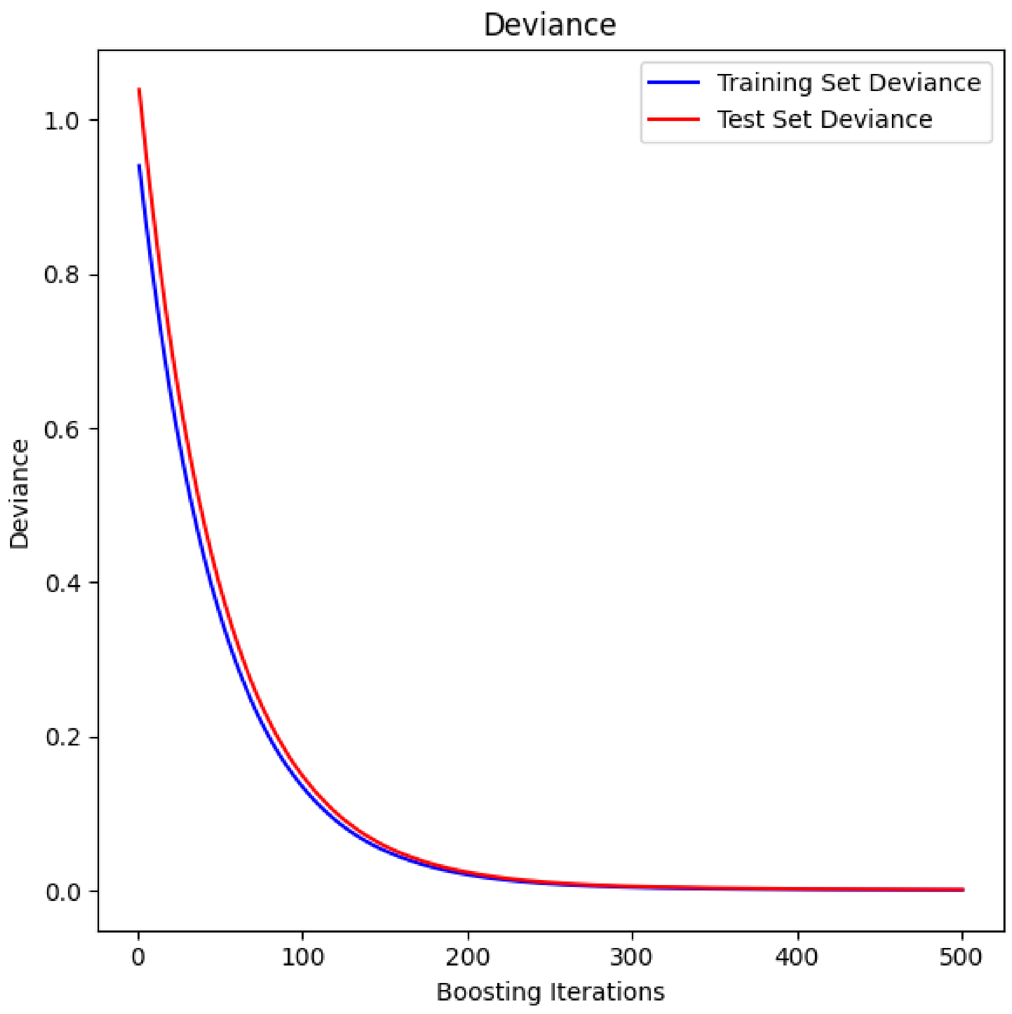
Due to the difficulties in the random forest regression, a gradient boosting regressor technique was attempted. For this approach, a multivariate attempt was performed, where all parameters were used as predictors with the runtime as the outcome. The parameters for the approach are shown below in Table 6.

Table 6: Gradient Boosting Regressor Parameters

Parameter	**Value**
n_estimators	500
max_depth	4
min_samples_split	5
learning_rate	0.01
Loss	squared_error

A gradient boosting regressor was created using these parameters, where the model was fit to the training data. Using the testing data, predictions were made. In addition to computing the mean square error, the deviance of the model was identified versus the boosting iterations. Starting with n_estimators set to 0 and iterating through to 500, the deviance was plotted. The results are shown below.

Figure 15: Gradient Boosting Regressor Deviance vs Boosting Iterations



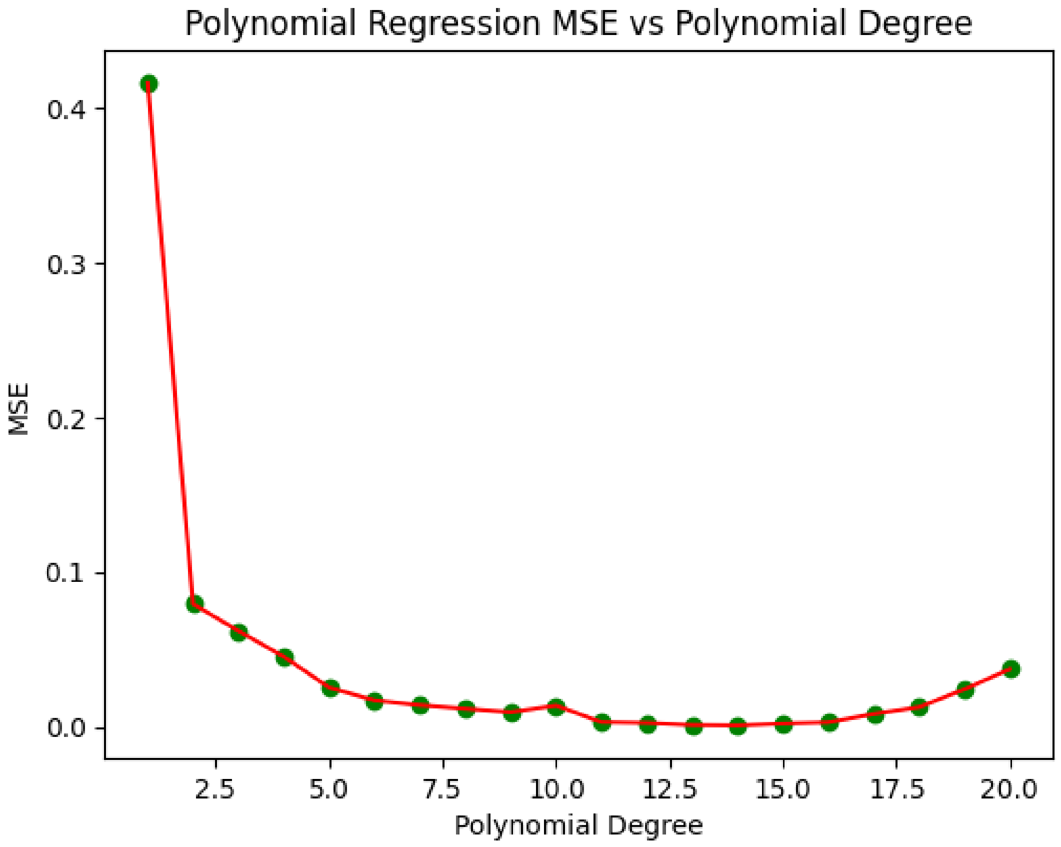
With n_estimators set to 500, the MSE obtained was 0.0015, highlighting the success of this approach.

3) Polynomial Regression

Since each individual parameter appeared to follow either a linear or polynomial trend, multivariate polynomial regression was also attempted. In this approach, rather than guessing at a polynomial degree to fit, a loop was created. In this loop, polynomial regression would be fit based on the loop iteration. The degree began at degree 1 (linear), and looped through degree 20. At each iteration, the

MSE was calculated and stored. After completion of the loop, results were plotted based on the degree. The results are shown below in Figure 16.

Figure 16: Polynomial Regression MSE vs Polynomial Degree



From this figure, it can be observed that while the MSE started at 0.4 for a linear fit, there was substantial improvement from increasing the degree by 1. Even with degree set to 2, the MSE dropped below 0.1. From this figure, it is shown that degrees 11 through 14 had the lowest MSE value. This figure also highlights that as the degree increases, performance does not necessarily also increase. After degree 14, the MSE began to increase. Table 7 shown below displays the MSE value per degree.

From Table 7, it is observed that the lowest MSE values are obtained when the polynomial degree is 13 or 14, but polynomial degrees of 11, 12, 15, and 16 are only slightly worse, with comparable results.

Table 7: MSE per Polynomial Degree for Multivariate Polynomial Regression

Degree	**MSE**
1	0.416
2	0.079
3	0.062
4	0.045
5	0.025
6	0.017
7	0.014
8	0.012
9	0.009
10	0.014
11	0.003
12	0.002
13	0.001
14	0.001
15	0.002
16	0.003
17	0.008
18	0.013
19	0.024
20	0.037

4) Support Vector Regression

As a means of comparison, Support Vector Regression (SVR) was also performed. For the SVR, a multivariate approach was taken, where each parameter was used a predictor versus the outcome variable of runtime. In order to determine optimal parameters, GridSearchCV was used. The parameter grid had the following values, shown in Table 8.

Table 8: Parameter Grid Values for the SVR GridSearch Cross-Validation

Parameter	**Range**
Regularization	[0.1, 1.0, 10, 100, 1000]
Kernel Coefficient	[1.0, 0.1, 0.01, 0.001, 0.0001]
Kernel	rbf

After optimal parameters were identified, the MSE obtained was 0.0063.

Section IV: Results

Each data mining technique provided insight on the underlying data. Some approaches were unable to properly capture the multivariate nature of the data, some linear regression approaches were unable to properly model the parameters due to the nonlinear aspects, and some approaches were not well-suited to be applied to the data. Table 9 highlights and compares the approaches.

Table 9: MSE Comparisons to Multivariate Regression Techniques

Technique	Best MSE
Gradient Boosting Regressor	0.0015
Polynomial Regression	0.00082
Support Vector Regression	0.0063

From this table, it is noted that the polynomial regression approach has the lowest MSE value. However, the MSE value is not the only consideration to make. In this case, there are 4 predictor variables (nodes, exploits, applicability of exploits, and database load). For polynomial regression, in order to get the best MSE, a degree of 14 had to be specified. When fitting a degree of 14 to data with only 4 predictor variables, the model will be overfitted to the data. Though the results appear to match for this set of data, it is likely that the model will be unable to properly capture any future data.

For per-feature regression, multiple solutions will be necessary. The number of nodes and the number of exploits can be adequately modeled using linear regression since the data fits a linear trend. However, the applicability of exploits and the database load parameter do not follow a linear trend, and therefore perform poorly under a linear model. Under the same guideline, random forest regression is unable to adequately capture the trend of the data for certain parameters.

There are a few considerations to make when deciding on which approach best modeled the data, and there is no single correct answer. The linear regression approach for the node and exploit parameter performed quite well, and was able to fit the data. However, it was unable to capture the multivariate nature of the data. On the other hand, the nonlinear techniques were able to capture the multivariate nature of the data. Unfortunately, though it could capture the trend across all variables, it also ran the risk of overfitting the data. Polynomial regression performed best under circumstances where the degree was much larger than the number of predictor variables. As a result, it is necessary to analyze each model, and use various results from each model. Each technique used for this data contributed useful information, and there is no single model that can answer all questions.

Section V: Conclusions and Future Works

This work implemented various data mining techniques to timing data collected by a new algorithm for large-scale attack and compliance graph generation. Each technique highlights trends of the underlying data, though some approaches were notably better than others. Exploratory Data Analysis (EDA) was able to highlight the shape of the data, along with information regarding the multicolinearity of it through variance inflation factors.

For comprehensiveness, both linear and nonlinear techniques were applied to the data. By applying both, various aspects of the data were able to be captured. Since certain parameters such as the number of nodes and the number of exploits were linear, linear regression techniques were adequately able to model the data. Since other parameters were nonlinear, nonlinear techniques were useful in modeling a fit. In addition, since the data is multivariate, nonlinear techniques were able to make sense of the data as a whole, rather than analyzing each feature individually.

The analysis portion of this work also has room for additional investigations. This work measured speedup according to Amdahl [10]. Since tasks are clearly defined and timing data is collected for each task, other speedup metrics can be used. Both Gustafson's Law [11] and Sun and Ni's Law [12] can be leveraged to obtain other results regarding the speedup of the tasking approach.

Though the MSE values obtained for nonlinear techniques appear to be promising, additional investigation can be performed to determine the validity of the model. By simulating or obtaining data beyond the bounds of the data available, the model can be analyzed to determine if it fits future data. This would be especially useful in the case of polynomial regression to determine by what degree the model may be overfitted.

REFERENCES

[1] X. Ou, W. F. Boyer, and M. A. Mcqueen, “A Scalable Approach to Attack Graph Generation,” CCS ’06: Proceedings of the 13th ACM conference on Computer and communications security, pp. 336–345, 2006.

[2] J. Zhang, S. Khoram, and J. Li, “Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search,” FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 207–216, 2017

[3] S. Arifuzzaman and M. Khan, “Fast parallel conversion of edge list to adjacency list for large-scale graphs,” in HPC ’15: Proceedings of the Symposium on High Performance Computing, pp. 17–24, Apr. 2015.

[4] X. Yu, W. Chen, J. Miao, J. Chen, H. Mao, Q. Luo, and L. Gu, “The Construction of Large Graph Data Structures in a Scalable Distributed Message System,” in HPCCT 2018: Proceedings of the 2018 2nd High Performance Computing and Cluster Technologies Conference, pp. 6–10, June 2018.

[5] P. Liakos, K. Papakonstantinopoulou, and A. Delis, “Memory-Optimized Distributed Graph Processing through Novel Compression Techniques,” in CIKM ’16: Proceedings of the 25th ACM International Conference on Information and Knowledge Management, pp. 2317–2322, Oct. 2016.

[6] J. Balaji and R. Sunderraman, “Graph Topology Abstraction for Distributed Path Queries,” in HPGP ’16: Proceedings of the ACM Workshop on High Performance Graph Processing, pp. 27–34, May 2016.

[7] K. Kaynar and F. Sivrikaya, “Distributed attack graph generation,” IEEE Transactions on Dependable and Secure Computing, vol. 13, no. 5, pp. 519–532, 2016.

[8] K. Cook, T. Shaw, J. Hale, and P. Hawrylak, “Scalable attack graph generation,” Proceedings of the 11th Annual Cyber and Information Security Research Conference, CISRC 2016, 2016.

[9] M. Li, P. Hawrylak, and J. Hale, “Concurrency Strategies for Attack Graph Generation,” Proceedings - 2019 2nd International Conference on Data Intelligence and Security, ICDIS 2019, pp. 174–179, 2019.

[10] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS ’67 (Spring), (New York, NY, USA), p. 483–485, Association for Computing Machinery, 1967.

[11] J. L. Gustafson, “Reevaluating amdahl’s law,” Commun. ACM, vol. 31, p. 532–533, may 1988.

[12] X.-H. Sun and L. M. Ni, “Another view on parallel speedup,” in Proceedings of the 1990 ACM/IEEE conference on Supercomputing, pp. 324–333, 1990.

APPENDIX - Code

```
# Group 7 - Noah L. Schrick

# Imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from dmba import regressionSummary
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
from sklearn.datasets import make_hastie_10_2
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report,confusion_matrix
from sklearn import preprocessing


# Import data
timing_df = pd.read_csv('timing.csv')

import plotly.express as px
overall_df = timing_df[['nodes', 'exploit', 'appl', 'load', 'runtime']]
overall_df['load'] = overall_df['load'].replace(395,0)
overall_df['load'] = overall_df['load'].replace(296,25)
overall_df['load'] = overall_df['load'].replace(197,50)
overall_df['load'] = overall_df['load'].replace(79,75)
overall_df['load'] = overall_df['load'].replace(1,100)

# , [296,25], [197,50], [79,75], [1,100]
fig = px.parallel_coordinates(overall_df, color="runtime", labels={"runtime": "runtime",
            "nodes": "nodes", "exploit": "exploit",
            "appl": "appl", "load": "load", },
                    color_continuous_scale=px.colors.diverging.Tealrose,
                    color_continuous_midpoint=2)
fig.show()
```

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
predictors = ['nodes', 'exploit', 'appl', 'load']
X = timing_df[predictors]

# VIF dataframe
vif_data = pd.DataFrame()
vif_data["feature"] = X.columns

# calculating VIF for each feature
vif_data["VIF"] = [variance_inflation_factor(X.values, i)
                          for i in range(len(X.columns))]

print(vif_data)


for feat in ['nodes', 'exploit', 'appl', 'load']:
    plt.figure()
    plt.plot(timing_df[feat], timing_df['runtime'])
    plt.title(feat)
    plt.ylabel('runtime')


node_pivot = timing_df.pivot_table(index=["nodes"], values=["runtime"], aggfunc='mean')
exploit_pivot = timing_df.pivot_table(index=["exploit"], values=["runtime"], aggfunc='mean')
appl_pivot = timing_df.pivot_table(index=["appl"], values=["runtime"], aggfunc='mean')

overall_df['load'] = overall_df['load'].replace(395,0)
overall_df['load'] = overall_df['load'].replace(296,25)
overall_df['load'] = overall_df['load'].replace(197,50)
overall_df['load'] = overall_df['load'].replace(79,75)
overall_df['load'] = overall_df['load'].replace(1,100)
load_pivot = overall_df.pivot_table(index=["load"], values=["runtime"], aggfunc='mean')

node_pivot.plot(kind='bar', title='Number of Nodes vs Runtime', ylabel='Runtime (ms)',
xlabel="Nodes")
exploit_pivot.plot(kind='bar', title='Number of Exploits vs Runtime', ylabel='Runtime (ms)',
xlabel="Number of Exploits")
appl_pivot.plot(kind='bar', title='Applicability of Exploits vs Runtime', ylabel='Runtime (ms)',
xlabel="Applicability of Exploits (%)")
load_pivot.plot(kind='bar', title='Database Load vs Runtime', ylabel='Runtime (ms)', xlabel="Database
Load (%)")


timing_corr = timing_df.corr().round(3)
# print(timing_corr)
```

```
fig, ax = plt.subplots()
fig.set_size_inches(11, 7)
sns.heatmap(timing_corr, annot=True, fmt=".1f", cmap="RdBu", center=0, ax=ax)

# Add speedup and efficiency columns
timing_df['speedup'] = np.nan
timing_df['efficiency'] = np.nan
for idx, row in timing_df.iterrows():
    nodes,exploits,appl,load = row['nodes'], row['exploit'], row['appl'], row['load']
    timing_df.at[idx,'speedup'] = timing_df['runtime'][timing_df['nodes'] == 1]
[timing_df['exploit']==exploits][timing_df['appl']==appl][timing_df['load']==load].values[0]/
timing_df['runtime'][timing_df['nodes'] == nodes][timing_df['exploit']==exploits]
[timing_df['appl']==appl][timing_df['load']==load].values[0]
    timing_df.at[idx,'efficiency'] = timing_df['runtime'][timing_df['nodes'] == 1]
[timing_df['exploit']==exploits][timing_df['appl']==appl][timing_df['load']==load].values[0]/
(timing_df['runtime'][timing_df['nodes'] == nodes][timing_df['exploit']==exploits]
[timing_df['appl']==appl][timing_df['load']==load].values[0]*nodes)

res_spd_df = pd.DataFrame(timing_df.pivot_table(index=["nodes"], values=["speedup"],
aggfunc=['max', 'min', 'mean']).to_records())
res_spd_df.set_axis(['Nodes', 'Max', 'Min', 'Mean'], axis=1, inplace=True)
res_spd_df.plot(kind='bar', x='Nodes', title='Minimum, Maximum, and Mean Speedups of MPI
Tasking\n for Increasing Problem Sizes', ylabel="Speedup (Amdahl's)", xlabel="Number of Nodes")

# x axis: exploits
# y axis: speedup and eff
# At each xtick: compartments 0-100 for appl. Min, mean, max for each?
tmp_pv = pd.DataFrame(timing_df.pivot_table(index=["exploit", "nodes"], values=["speedup"],
aggfunc='mean').to_records())
ax = sns.lineplot(x="exploit", y="speedup", hue="nodes", palette="colorblind",
data=tmp_pv).set(title='Mean Speedup for the Exploit Parameter\n Across the Number of Compute
Nodes')
plt.show()

tmp_pv = pd.DataFrame(timing_df.pivot_table(index=["appl", "nodes"], values=["speedup"],
aggfunc='mean').to_records())
ax = sns.lineplot(x="appl", y="speedup", hue="nodes", palette="colorblind",
data=tmp_pv).set(title='Mean Speedup for the Applicable Exploit Parameter\n Across the Number of
Compute Nodes')
plt.show()

tmp_pv = pd.DataFrame(timing_df.pivot_table(index=["load", "nodes"], values=["speedup"],
aggfunc='mean').to_records())
tmp_pv['load'] = tmp_pv['load'].replace(395,0)
tmp_pv['load'] = tmp_pv['load'].replace(296,25)
```

```
tmp_pv['load'] = tmp_pv['load'].replace(197,50)
tmp_pv['load'] = tmp_pv['load'].replace(79,75)
tmp_pv['load'] = tmp_pv['load'].replace(1,100)
ax = sns.lineplot(x="load", y="speedup", hue="nodes", palette="colorblind",
data=tmp_pv).set(title='Mean Speedup for the Database Load Parameter\n Across the Number of
Compute Nodes')
plt.show()


# x axis: exploits
# y axis: speedup and eff
# At each xtick: compartments 0-100 for appl. Min, mean, max for each?
tmp_pv = pd.DataFrame(timing_df.pivot_table(index=["exploit", "nodes"], values=["efficiency"],
aggfunc='mean').to_records())
ax = sns.lineplot(x="exploit", y="efficiency", hue="nodes", palette="colorblind",
data=tmp_pv).set(title='Mean Efficiency for the Exploit Parameter\n Across the Number of Compute
Nodes')
plt.show()

tmp_pv = pd.DataFrame(timing_df.pivot_table(index=["appl", "nodes"], values=["efficiency"],
aggfunc='mean').to_records())
ax = sns.lineplot(x="appl", y="efficiency", hue="nodes", palette="colorblind",
data=tmp_pv).set(title='Mean Efficiency for the Applicable Exploit Parameter\n Across the Number of
Compute Nodes')
plt.show()

tmp_pv = pd.DataFrame(timing_df.pivot_table(index=["load", "nodes"], values=["efficiency"],
aggfunc='mean').to_records())
tmp_pv['load'] = tmp_pv['load'].replace(395,0)
tmp_pv['load'] = tmp_pv['load'].replace(296,25)
tmp_pv['load'] = tmp_pv['load'].replace(197,50)
tmp_pv['load'] = tmp_pv['load'].replace(79,75)
tmp_pv['load'] = tmp_pv['load'].replace(1,100)
ax = sns.lineplot(x="load", y="efficiency", hue="nodes", palette="colorblind",
data=tmp_pv).set(title='Mean Efficiency for the Database Load Parameter\n Across the Number of
Compute Nodes')
plt.show()


predictors = ['nodes']
overall_outcome = 'runtime'
norm_df = (timing_df-timing_df.mean())/timing_df.std()
# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
```

```
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)
runtime_lm = LinearRegression()
runtime_lm.fit(train_X, train_y)
# print coefficients
print('intercept ', runtime_lm.intercept_)
print(pd.DataFrame({'Predictor': X.columns, 'coefficient': runtime_lm.coef_}))
# print performance measures
regressionSummary(train_y, runtime_lm.predict(train_X))

predictors = ['exploit']
overall_outcome = 'runtime'

# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)
runtime_lm = LinearRegression()
runtime_lm.fit(train_X, train_y)
# print coefficients
print('intercept ', runtime_lm.intercept_)
print(pd.DataFrame({'Predictor': X.columns, 'coefficient': runtime_lm.coef_}))
# print performance measures
regressionSummary(train_y, runtime_lm.predict(train_X))

predictors = ['appl']
overall_outcome = 'runtime'

# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)
runtime_lm = LinearRegression()
runtime_lm.fit(train_X, train_y)
# print coefficients
print('intercept ', runtime_lm.intercept_)
print(pd.DataFrame({'Predictor': X.columns, 'coefficient': runtime_lm.coef_}))
# print performance measures
regressionSummary(train_y, runtime_lm.predict(train_X))

predictors = ['load']
overall_outcome = 'runtime'

# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
```

```
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)
runtime_lm = LinearRegression()
runtime_lm.fit(train_X, train_y)
# print coefficients
print('intercept ', runtime_lm.intercept_)
print(pd.DataFrame({'Predictor': X.columns, 'coefficient': runtime_lm.coef_}))
# print performance measures
regressionSummary(train_y, runtime_lm.predict(train_X))

predictors = ['nodes', 'exploit', 'appl', 'load']
overall_outcome = 'runtime'

# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)
runtime_lm = LinearRegression()
runtime_lm.fit(train_X, train_y)
# print coefficients
print('intercept ', runtime_lm.intercept_)
print(pd.DataFrame({'Predictor': X.columns, 'coefficient': runtime_lm.coef_}))
# print performance measures
regressionSummary(train_y, runtime_lm.predict(train_X))

# Random Forest
#predictors = ['nodes', 'exploit', 'appl', 'load']
predictors = ['nodes']
overall_outcome = 'runtime'

# partition data
X = timing_df[predictors]
overall_y = timing_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)

regressor = RandomForestRegressor(n_estimators = 10, random_state = 0)
regressor.fit(train_X, train_y)

X_grid = np.arange(0, 12, 1)
X_grid = X_grid.reshape((len(X_grid),1))
plt.scatter(X, overall_y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Random Forest Regression')
plt.xlabel('Nodes')
plt.ylabel('Runtime (ms)')
plt.show()
```

```
# Random Forest
#predictors = ['nodes', 'exploit', 'appl', 'load']
predictors = ['exploit']
overall_outcome = 'runtime'

# partition data
X = timing_df[predictors]
overall_y = timing_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)

regressor = RandomForestRegressor(n_estimators = 10, random_state = 0)
regressor.fit(train_X, train_y)

X_grid = np.arange(0, 49152, 10)
X_grid = X_grid.reshape((len(X_grid),1))
plt.scatter(X, overall_y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Random Forest Regression')
plt.xlabel('Exploits')
plt.ylabel('Runtime (ms)')
plt.show()

# Random Forest
#predictors = ['nodes', 'exploit', 'appl', 'load']
predictors = ['appl']
overall_outcome = 'runtime'

# partition data
X = timing_df[predictors]
overall_y = timing_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)

regressor = RandomForestRegressor(n_estimators = 10, random_state = 0)
regressor.fit(train_X, train_y)

X_grid = np.arange(0, 100, 10)
X_grid = X_grid.reshape((len(X_grid),1))
plt.scatter(X, overall_y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Random Forest Regression')
plt.xlabel('Applicability of Exploits')
plt.ylabel('Runtime (ms)')
plt.show()
```

```
# Random Forest
#predictors = ['nodes', 'exploit', 'appl', 'load']
predictors = ['load']
overall_outcome = 'runtime'

# partition data
X = timing_df[predictors]
overall_y = timing_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)

regressor = RandomForestRegressor(n_estimators = 10, random_state = 0)
regressor.fit(train_X, train_y)

X_grid = np.arange(0, 400, 10)
X_grid = X_grid.reshape((len(X_grid),1))
plt.scatter(X, overall_y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Random Forest Regression')
plt.xlabel('Database Load')
plt.ylabel('Runtime (ms)')
plt.show()

predictors = ['nodes', 'exploit', 'appl', 'load']
overall_outcome = 'runtime'

# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)

params = {
    "n_estimators": 500,
    "max_depth": 4,
    "min_samples_split": 5,
    "learning_rate": 0.01,
    "loss": "squared_error",
}

reg = GradientBoostingRegressor(**params)
reg.fit(train_X, train_y)

mse = mean_squared_error(valid_y, reg.predict(valid_X))
print("The mean squared error (MSE) on test set: {:.4f}".format(mse))
```

```
test_score = np.zeros((params["n_estimators"],), dtype=np.float64)
for i, y_pred in enumerate(reg.staged_predict(valid_X)):
    test_score[i] = mean_squared_error(valid_y, y_pred)

fig = plt.figure(figsize=(6, 6))
plt.subplot(1, 1, 1)
plt.title("Deviance")
plt.plot(
    np.arange(params["n_estimators"]) + 1,
    reg.train_score_,
    "b-",
    label="Training Set Deviance",
)
plt.plot(
    np.arange(params["n_estimators"]) + 1, test_score, "r-", label="Test Set Deviance"
)
plt.legend(loc="upper right")
plt.xlabel("Boosting Iterations")
plt.ylabel("Deviance")
fig.tight_layout()
plt.show()

predictors = ['nodes', 'exploit', 'appl', 'load']
overall_outcome = 'runtime'

# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)

plt_mean_squared_error = []
deg_range = range(1,21)
for degree in deg_range:
    poly = PolynomialFeatures(degree = degree)
    X_poly = poly.fit_transform(train_X)
    poly.fit(X_poly, train_y)

    regression_model = LinearRegression()
    regression_model.fit(X_poly, train_y)
    y_pred = regression_model.predict(X_poly)
    plt_mean_squared_error.append(mean_squared_error(train_y, y_pred, squared=False))


plt.scatter(deg_range,plt_mean_squared_error, color="green")
plt.plot(deg_range,plt_mean_squared_error, color="red")
plt.xlabel('Polynomial Degree')
plt.ylabel('MSE')
plt.title('Polynomial Regression MSE vs Polynomial Degree')
plt_mean_squared_error

# SVR
predictors = ['nodes', 'exploit', 'appl', 'load']
overall_outcome = 'runtime'

# partition data
X = norm_df[predictors]
overall_y = norm_df[overall_outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, overall_y, test_size=0.4, random_state=1)

param_grid = {'C': [0.1,1, 10, 100, 1000], 'gamma': [1,0.1,0.01,0.001,0.0001], 'kernel': ['rbf']}
grid = GridSearchCV(SVR(),param_grid,refit=True,verbose=0)
grid.fit(train_X,train_y)

grid_predictions = grid.predict(valid_X)

mean_squared_error(valid_y, grid_predictions, squared=True)
```