



Data Mining for Software Management: Automatic marking of complex Rust code using software metrics

Panagiotis Karatakis

SID: 3308220009

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Data Science

DECEMBER 2023
THESSALONIKI – GREECE



Data Mining for Software Management: Automatic marking of complex Rust code using software metrics

Panagiotis Karatakis

SID: 3308220009

Supervisor:	Dr. C. Tjortjis
Supervising Committee	Dr. L. Akritidis
Members:	Dr. P. Koukaras

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Data Science

NOVEMBER 2023
THESSALONIKI – GREECE

Abstract

This thesis explores the use of data mining and machine learning techniques to mark complex Rust code using software metrics automatically. The methodology proposed involves four main procedures: dataset construction, feature extraction, model training and finetuning, and model evaluation. The dataset construction involves the creation of a ground truth dataset by collecting commit messages and their metadata, performing NLP analysis, and extracting software metrics. Feature extraction involves enhancing the dataset with additional features to improve model performance. Model training and finetuning involve training and optimizing the models using various machine learning algorithms. Finally, model evaluation involves assessing the performance of the models using various evaluation metrics. The results show promising performance in detecting software defects, with F1 scores of 77% and AUC scores of 85%. The study also highlights limitations and future research opportunities, such as advanced feature engineering, larger sample sizes, and more complex algorithms. Overall, this thesis contributes to the development of automated methods for software management and provides valuable insights for stakeholders in the software development industry.

Lastly, I would like to acknowledge the help from Professor Christos Tjortjis, who mentored my thesis writing journey, giving valuable advice and guidelines, and the rest of the Data Mining and Analytics research team of IHU, DAMA for short, who reviewed and advised me on the more practical parts of the research.

Panagiotis Karataakis

2024-01-08

Contents

1 Introduction.....	1
1.1 Background.....	1
1.2 Statement of Problem.....	2
1.3 Statement of Purpose.....	2
1.4 Research Questions and Objectives.....	2
1.5 Research Methodology.....	3
1.6 Research Limitations.....	3
1.7 Organization of Dissertation.....	3
2 Literature Review.....	5
2.1 Background Information.....	5
2.1.1 Software Quality.....	5
2.1.2 Software Metrics.....	6
2.1.3 Data Mining.....	9
2.1.4 Machine Learning.....	11
2.1.5 Data Classification Evaluation Metrics.....	13
2.1.6 Natural Language Processing.....	14
2.2 Related Work.....	15
2.2.1 Unsupervised Learning Software Quality Evaluation.....	16
2.2.2 Supervised Learning Software Quality Evaluation.....	16
2.2.3 Semisupervised Learning Software Quality Evaluation.....	17
2.2.4 Marking Bug Commits.....	17
3 Methodology.....	17
3.1 Data Collection.....	18
3.1.1 Sample Repositories.....	18
3.1.2 Collect Repositories Metadata.....	24
3.1.3 NLP Analysis of Commit Messages.....	25
3.1.4 Mark Bug-Fixing Commits.....	27
3.1.5 Extract Commit Hashes of Files.....	27
3.1.6 Extract Software Analytics.....	28
3.1.7 Merge Information.....	28
3.2 Feature Extraction.....	29
3.2.1 Filtering.....	29
3.2.2 Clustering.....	30
3.2.3 Extra Features.....	31
3.3 Models' Training.....	32
3.3.1 Data Configurations.....	32
3.3.2 Algorithms And Configuration.....	33
3.3.3 Training Pipeline.....	33

3.4 Evaluation.....	34
4 Results.....	34
4.1 Rejected Algorithms.....	35
4.2 Accuracy Score.....	35
4.3 Precision Score.....	37
4.4 Recall Score.....	39
4.5 F1 Score.....	40
4.6 Area Under Curve Score.....	42
4.7 Important Software Metrics.....	44
4.8 Model Explainability.....	47
5 Discussion.....	48
5.1 Interpretation of Results.....	48
5.1.1 Score Metrics.....	48
5.1.2 Best Performing Model.....	49
5.1.3 Feature Engineering.....	50
5.1.4 Model Explainability.....	51
5.1.5 Decision Tree vs. Random Forest.....	51
5.2 Comparison with Literature.....	51
5.3 Theoretical and Practical Implications.....	52
5.4 Limitations and Future Research.....	52
5.4.1 Limitations.....	52
5.4.2 Advanced Feature Engineering.....	53
5.4.3 Use LLMs for Marking Bug-Fixed Files.....	53
5.4.4 Larger Sample.....	53
5.4.5 Survey Users.....	53
5.4.6 Complex Algorithms.....	54
5.4.7 Explain Models in Depth.....	54
5.4.8 Finetuning of Models.....	54
6 Conclusion.....	55
Bibliography.....	55
Appendix.....	57
Source Code.....	57
Dataset.....	57
Accuracy Score Results.....	57
Precision Score Results.....	68
Recall Score Results.....	78
F1 Score Results.....	88
Area Under Curve Score Results.....	98

1 Introduction

This chapter presents the information required to understand the importance of the problem, the purpose of this research, and the structure of this research paper. The study aims to develop a framework for automatically evaluating software quality using software metrics and machine learning algorithms. It employs a quantitative approach to address research questions related to the selection of code metrics, machine learning algorithms, evaluation metrics, and the potential impact of feature engineering techniques on the framework's performance. The methodology involves dataset construction, feature extraction, model training and finetuning, and model evaluation. The primary objectives include assessing the effectiveness of the proposed framework in detecting software defects, identifying impactful software metrics, exploring the effectiveness of different machine learning algorithms, and providing valuable insights for stakeholders in the software development industry to optimize their development process. The first section introduces the background information needed to highlight the importance of the problem. The second section presents the problem and its characteristics. Next, the third section discusses the statement of purpose of this research. The fourth section presents the research questions and objectives. The fifth section states our research methodology approach, while the sixth section presents our research limitations. Lastly, the seventh section presents the structure of the research paper and some brief information about each chapter.

1.1 Background

In today's world, management plays an essential role by organizing the limited resources the environment provides to accomplish a goal. Like all fields, software development has limited resources (time, budget) to deliver software projects, so management is essential to deliver results. A study has shown that most tech companies have non-technical managers in leading positions [1]. We can safely assume that those managers need to learn the technical aspects of a software project, but they are excellent at utilizing the teams to a great extent [1], [2]. Some research studies have shown that 31% of all software projects are terminated due to quality problems, generating losses of about \$81 billion annually. In comparison, those that managed to be delivered have only 42% of the originally planned features; this indicates that quality is critical for the project's success [1]–[4]. Code quality metrics are a great way to control the

code-base quality by helping eliminate several contributors to "bad code," such as foul smells and the effects of poor design or implementation choices [5]. These quality metrics are also helpful in managing the developers better, which improves their development process and saves time [2]. A study suggested that most code issues can be avoided by simply applying quality checks when the work is committed to repositories [5]. Managers should find a way to evaluate the quality of the code base to save resources and optimize the development process; for this, code quality metrics are a valuable tool. Nevertheless, since software projects depend on the skills of a few individuals performing intense manual tasks, there is a need to develop more automated methods [3].

1.2 Statement of Problem

As technology progresses, new languages emerge, intending to increase productivity and produce more secure code. One of those languages is Rust, which became public in 2015. The language aims to produce reliable and safe code with solid guarantees about isolation, concurrency, and memory safety [6]. Moreover, Rust tends to perform like C++ in some cases due to the transparent performance model that makes it easier to reason about code efficiency [6]. An interesting fact about Rust is that 87% of surveyed developers voted it the most loved language for the seventh consecutive year and wanted to keep using it [7]. Rust as a language has an average complexity compared to mainstream languages [8], but as software systems written in Rust grow, they become complex and challenging to comprehend [9]. As a result, more faults appear, causing delays that raise the development cost [9]. That is why managers depend on software metrics to comprehend system complexity, which makes the need for an automated system more important [9], [10].

1.3 Statement of Purpose

During the development of the theoretical background, we found that many papers mention the need to develop the evaluation of software quality using data mining in other languages [11]. Moreover, a few have raised the need for a more interactive platform where software engineers can explore the data and label the groups appropriately [10]. In contrast, others suggest different classification methods [10]–[12]. In this paper, we suggest a process that takes several code repositories, automatically annotates code files that had a bug fix in the past, and trains an artificial intelligence model to detect files with those characteristics. In the literature, files that contain bug fixes in their past display characteristics of complex code,

files that are hard to comprehend and more accessible to introduce bugs [13]. Our purpose is to evaluate if this procedure can display positive results and evaluate various machine algorithms and the models they produce.

1.4 Research Questions and Objectives

To approach a solution to the problem, we constructed the following research questions. The main question is whether or not our proposed framework produces acceptable results. However, to answer the main question, we must first justify five sub-questions that further strengthen its importance. Those sub-questions study which software metrics and machine learning algorithms work best, what evaluation metrics to use, and whether we can improve the result by utilizing feature engineering techniques.

- **RQ:** Does our proposed automatic software quality evaluation framework produce acceptable results?
- **RQ1:** Which code metrics should we use?
- **RQ2:** What subsets of data should we use?
- **RQ3:** Which machine learning algorithms produce acceptable results?
- **RQ4:** Which classification evaluation metrics should we use?
- **RQ5:** Could we use feature engineering techniques to improve the result?

1.5 Research Methodology

The first step in the methodology was the construction of the dataset. That involves creating a ground truth dataset to train the models and evaluate their performance. We constructed the dataset in five steps: the creation of the repositories sample, the collection of the commit messages and their metadata, the NLP analysis of the commit messages, the extraction of the software metrics, and the merging of all the information produced from steps three and four. The second step involves enhancing the dataset with features to check if the models can be improved. This step involves filtering the data to remove outliers, applying feature engineering, and extracting more advanced features by clustering the entries. The third step involves training and finetuning the models. The study used machine learning algorithms, including Decision Tree, Random Forest, Perceptron, and XGBoost. We trained the models using the dataset constructed in step one and the features extracted in step two. The study used RandomizedSearchCV to stay within time constraints and not exhaustively test the whole parameter set space. The final step involves evaluating the performance of the models

and picking the best-performing model. The study used various evaluation metrics, including accuracy, precision, recall, F1 score, and AUC score. The proposed methodology involves constructing a dataset, extracting features, training and finetuning models, and evaluating their performance. The study used a quantitative approach to cover the research questions and compared the results for different methods to pick the ones that performed better based on various evaluation metrics [14]–[16]. We discuss the methodology further in Chapter 3.

1.6 Research Limitations

To conduct this research, we took the following assumptions and hypotheses to provide a solution with our limited resources. Because many code repositories are available, extracting and training a model on this data size would take a long time, so we sampled the repositories with a very detailed procedure. In addition, we assume that code that contains bug fixes in the past we consider as complex. A paper by Khoshgoftaar presented that when code is complex, developers need help understanding it, making it harder to maintain and more accessible to introduce bugs [13]. Lastly, in the literature, many papers state that data extracted from software metrics can contain outliers and noisy samples. To combat that, we used a large sample size and features extracted from applying clustering to alleviate that problem [10].

1.7 Organization of Dissertation

This section describes the thesis's structure and details each chapter. Chapter 2 contains the theoretical background essential to our work and related work that we depend on or improve their results. Chapter 3 presents our proposed methodology in detail, including the data gathering, model training, and evaluation. Chapter 4 contains all the results produced during our experiments with essential insights. Chapter 5 discusses our most important observations during our research and comments on the results. Chapter 6 collects all the future work suggestions that could improve the proposed framework. Lastly, Chapter 7 concludes our research work and answers the research questions.

2 Literature Review

This chapter presents all the background information and the current research efforts relevant to our research. The first section presents theoretical background information, which we will use as building blocks for our methodology. In the second section, we present related works from the literature that we use their methodology to improve further and bring value, as well as some works that act as a baseline to compare in the research field.

2.1 Background Information

This section presents the theoretical information we use in our proposed methodology. The first subsection introduces software quality, its importance, and a proposed quality evaluation model. In the following subsection, we present software metrics and a brief description of various software metrics. Next, we give details about Data Mining and Machine learning and their research fields and highlight essential areas we use. The following subsection presents various classification model evaluation metrics we used in the research, and lastly, we make a short introduction to the Natural Language Processing research field.

2.1.1 Software Quality

Software quality is essential because many businesses depend on software to function, and quality software can be safe and more straightforward to extend. There are many models to assess the quality of software components. The literature separates software evaluation methods into two categories based on whether the approach is product-specific or generalized. Product-specific models utilize historical trends, expert observations, and requirements to construct a model that evaluates the quality of a specific code repository. In contrast, generalized models use various code repositories as a sample to construct the quality estimator [2].

One of the most cited quality standards is ISO/IEC 9126, designed to provide guidelines and strategies for evaluating the quality of software components and making evaluation reproducible. The standard was proposed incrementally in 4 parts. The first part mentions the connection between different evaluation approaches and additional

characteristics that help detect quality. The second part defined external metrics that assist in quantifying quality characteristics. The third part is similar to the second but refers to internal metrics. The fourth part is more user-centric and studies metrics that measure the quality characteristics of the user [17].

2.1.2 Software Metrics

To decide whether a software component is simple and assess its maintainability, an individual has to read the code and decide. However, this requires knowledge of software programming, so they must inspect the code quality using other means. One of those means is software metrics, where they extract various characteristics of the code and translate them into numbers. That makes it easy for individuals who do not understand software programming or do not have too much time to take a quick glimpse at the modules, compare them with others, and decide if they fit the requirements. Our research uses the rust-code-analyzer tool to extract software metrics from the source code. The tool computes 11 software metrics for the whole file and each component individually and structures them in a nested tree format. If the software module contains children, the tool calculates statistics (average, min, max) for each software metric of the values of its children [18].

Metric	Description	Calculated Values
Cognitive Complexity	This metric describes how difficult it is to understand a unit of code. It studies how much the code branches and the amount of variables in the current scope and, through an empirical study, quantifies the effort a developer will consume to understand the code [18].	Value Sum Average Min Max
Cyclomatic	This metric calculates the branching factor of a code fragment. The higher the number, the more complex a fragment and the harder it is to maintain [18].	Value Sum Average Min Max

SLoc	This metric counts a code element's total code lines [18].	Value Average Min Max
PLoc	This metric counts a code element's logic code and comment lines [18].	Average Min Max
CLoc	This metric counts a code element's total comment lines [18].	Average Min Max
LLoc	This metric counts a code element's total logic code lines [18].	Average Min Max
Blank	This metric counts a code element's total logic code lines [18].	Average Min Max
Loc	This metric contains average values of all previous metrics (SLoc, PLoc, LLoc, CLoc, Blank) [18].	SLoc PLoc LLoc CLoc Blank

Halstead	That is a collection of metrics that encapsulate the maintainability of a software component. Some metrics include the estimated time to implement this module and the estimated bugs that hide in the source code [18].	n1_min N1_max n2_min N2_max Length Estimated_program_length Purity_ratio Vocabulary Volume Difficulty Level Effort Time Bugs
Mi	This metric is the Maintainability index, another measure that tries to represent how maintainable a piece of software is. This number is between 0 and 100 [18].	Original Sei Visual_studio
Nom	The number of method metrics counts the number of methods that appear in a source file [18].	Functions Closures Functions_average Closures_average Total Average Functions_min Functions_max Closures_min

		Closures_max
NArgs	This metric measures the number of arguments for a function/method [18].	Total_functions Total_closures Average_functions Average_closures Total Average Functions_min Functions_max Closures_min Closures_max
NExit	This metric counts the number of possible exit points from a function/method [18].	Sum Average Min Max

Table 1. Software Metrics we used in our research and their description.

2.1.3 Data Mining

In recent years, rapid digital transformation has converted many legacy systems into digital equivalents, generating enormous amounts of data hiding valuable insights researchers can extract. *Data mining* is the research field that studies how to extract valuable facts and insights from data through a standardized process of collection, cleaning, processing, analyzing, and extraction. This field is characterized by various challenges, like the differences between multiple data types, even in the same dataset, the missing values, or the size of the datasets are some of them. The field studies many algorithms that try to fix the issues a researcher can face during the data mining process and produce the result [16].

Data Mining Workflow

The data mining steps are standard, and the researcher should pay attention to them to succeed in their research. The list below describes the steps and some of their details [16].

1. **Data Collection:** In this step, the researcher should pick the sources from where they will collect the data, check if the source is reliable, and find a way to store it efficiently without information loss. A data warehouse is an essential tool to store data if they have a considerable volume because the system might fail to record that much data [16].
2. **Data Cleaning & Preprocessing:** The sources might be unreliable and provide inaccurate or corrupt data, and those values should be detected and removed. Moreover, the dataset balance is significant and can impact the extracted information, so the person performing this step should take extra care. There are many algorithms and approaches to solve this problem, like dropping rows with missing values or detecting outliers through statistical clustering processes. Lastly, the researcher has to convert the data types and encode them into other formats that the algorithms can use [16].
3. **Feature Extraction:** In this step, the researcher seeks to extract other data from existing data. One notable example is extracting a person's age from their date of birth. In addition to simple feature extraction, the researcher can use more advanced feature engineering techniques, such as the distance from various clusters or even the classification ID of a record [16].
4. **Analytical Processing:** In this step, the research utilizes multiple algorithms, from classification, associative rules, clustering, and outlier detection, to extract insights from the data. Depending on the problem, the researcher has to pick the algorithm that covers their needs and finetune it to improve its effectiveness. If the data do not work with the current algorithm, the research should reconsider, preprocess, and scale them to fit the algorithms' requirements [16].
5. **Extracting:** After completing all the previous steps, the researcher can study the results and collect all the interesting insights and observations [16].

Areas of Data Mining

Data mining has four areas of interest: association and pattern matching, clustering, classification, and outlier detection. Different algorithms can solve those problems, but one can combine them to reach a common goal [16].

1. Association and Pattern Mining: In large datasets, there are patterns of associations between the data that are hard for someone to identify with the naked eye due to the complexity or size of the dataset. In this problem, we utilize algorithms to find the data's rule associations or occurring patterns. For example, in a dataset of supermarket transactions, we can identify if a customer buys Product A and Product B; they will probably purchase Product C, so we can utilize this insight to design the shop floors. Known algorithms of this category are the Apriori Algorithm and FP-Growth [16].
2. Data Clustering: Sometimes, we must find groups in the datasets for marketing purposes or find outliers to remove them. Clustering is a great tool that we can use to find groups with common characteristics and identify noisy records that do not belong to any to remove them. We can distinguish the clustering algorithms into different categories based on how they separate the groups; the most noticeable approaches are Density-based Clustering, Distribution-based Clustering, Centroid-based Clustering, and Hierarchical Clustering. The known algorithms of this category are K-Means, Aggregate clustering, and DBScan [16].
3. Outlier Detection: Some data mining algorithms can be proven helpful in detecting outliers or noisy records from a dataset. Outlier detection is proper when we must act if we detect one. One example of such a case is the credit fraud prevention systems. Another case is when noisy data contribute negatively to the performance of other data mining algorithms. Some algorithms that can perform outlier detection are KNN, DBScan, and Local Outlier Factor [16].
4. Data Classification: In this area, the researcher aims to figure out how the dataset items are partitioned already in their categories. The researcher can achieve this by constructing a model representing that knowledge and then using this model to predict unknown records and classify them. In our research, we will mainly utilize classification algorithms to construct a model that will predict if a code module contained a bug fix in the past. We can then use this model to predict code modules likely to contain a bug and require maintenance. Some noticeable classification algorithms are decision trees, random forests, and SVM [16].

2.1.4 Machine Learning

Machine learning is a research field that studies algorithms that solve complex problems that traditional programming cannot solve. An example of such a problem could be image recognition and object detection. Machine learning algorithms solve many problems

generically by producing models that can predict the labels of a record. The creation of the model process is called training. The problems that machine learning is trying to solve are classification, clustering, and regression. Classification is about predicting whether a record belongs to a class. Clustering studies how to group records with similar characteristics. Regression studies how to create models that predict continuous numbers [15].

2.1.4.1 Learning Models

The literature separates machine learning algorithms into three groups based on how they create the models. That distinction depends on the available data we have on hand and the goals we are trying to achieve [15].

1. Supervised Learning: This learning method's algorithm input is a labeled dataset in this learning method. That means the data contains the target feature we want to predict. Then, the algorithm trains a model that encapsulates knowledge on predicting that feature using the other features. This learning method can solve classification and regression problems [15].
2. Unsupervised Learning: In this case, we provide the machine learning algorithm with data that do not contain a target feature, but instead, we try to uncover groups that show a trend in the data [15].
3. Semi-Supervised Learning: In this scenario, we initially perform an unsupervised step to cluster the data and assign cluster ID to the records, and then we try to create a supervised model that predicts this grouping [15].
4. Reinforced Learning: In this learning method, we create a reward function, and the model inputs its following action into the defined function, and this function then rewards or penalizes the algorithm. Overall, the algorithm tries to create a model that displays desirable behavior that maximizes the results and minimizes the penalties [15].

2.1.4.2 Overlap with Data Mining

It is noted in the literature that those two fields have overlapping interests, especially in data classification. However, machine learning tends to study the theoretical and statistical aspects of the algorithms, whereas data mining studies how we can extract knowledge from the provided data [16].

2.1.5 Data Classification Evaluation Metrics

In classification problems, we aim to predict a variable Y from a dataset X , and we can do this by applying a wide array of algorithms that train models that can perform this operation $f(X) = Y$. The idea behind those models is straightforward: the models try to assign the most probable Y value given X features. Nevertheless, in some cases, we have to compare which algorithm is the best, and then we have to find which hyper-parameters tune the model to have the best performance. Classification evaluation metrics provide a way to compare how different algorithm models perform or find the hyperparameters that produce the best result. The subsections below will describe some basic evaluation metrics we plan to use in our research [19].

Confusion Matrix

This matrix will act as a foundation for other classification matrices. In this table, we can observe the Actual vs Predicted count of records after using the model to predict the values [16].

Actual / Predicted	Positive	Negative
Positive	TP	FN
Negative	FP	TN

Table 2. Confusion Matrix

Accuracy

This evaluation is the easiest to understand because we compare the correctly predicted items against all the items in the dataset. Unfortunately, the metric can hide the model's actual performance if the dataset is unbalanced or the weights of the classification labels are different. On the other hand, Balanced Accuracy treats each class equally and considers possible imbalances [16].

$$Accuracy = (TP + TN) / (TP + TN + FP + FN)$$

$$Balanced\ Accuracy = ((TP / Total_{row1}) + (TN / Total_{row2})) / 2$$

Precision

This metric describes the number of TP records the model predicted correctly compared to the total predicted Positive entries [16].

$$Precision = \frac{TP}{TP + FP}$$

Recall

This metric describes the number of TP records the model predicted correctly compared to the total number of positive entries [16].

$$Recall = \frac{TP}{TP + FN}$$

F1

This metric is the harmonic mean of Precision and Recall. It is a safe way to combine the Recall and Precision metrics and account for significant differences between those two metrics [16].

$$F = 2 \cdot \left(\frac{Precision \cdot Recall}{Precision + Recall} \right)$$

Micro and Macro Average

In case of imbalances, the mentioned metrics favor the target label, so if we want to care for those scenarios, we can use their micro or macro average statistics [16].

$$Precision_{macro} = \frac{\sum_{k=1}^K Precision_k}{K}, Recall_{macro} = \frac{\sum_{k=1}^K Recall_k}{K}, F1_{macro} = 2 \cdot \left(\frac{Precision_{macro} \cdot Recall_{macro}}{Precision_{macro}^{-1} + Recall_{macro}^{-1}} \right)$$
$$Precision_{micro} = Recall_{micro} = F1_{micro} = \frac{\sum_{k=1}^K TP_k}{Total\ Items}$$

AUC ROC

This metric is the receiver operating characteristic or area under the curve. It measures the area under the line formed by plotting the True Positive Rate and False Positive Rate at each threshold setting [20].

$$TPR = \frac{TP}{TP + FN}, FPR = \frac{FP}{FP + TN}$$

2.1.6 Natural Language Processing

The field of Natural Language Processing studies how machines can understand and interpret human languages. Some challenges NLP tries to solve include text-to-speech synthesis,

speech recognition, text generation, text summarization, and question responses. The literature characterizes NLP as a complex field, and for each challenge, various algorithms try to solve it. The language context makes the construction of capable algorithms in NLP because many languages have words that can have different meanings in other contexts. For our research, we require a small background in NLP to perform an ad-hoc implementation that detects commits that perform a bug fix [15].

The first component we have to understand is the tokenizer. The tokenizer is responsible for breaking down strings into separate words in their simple form. The first step is to separate the words into separate tokens. For example, we can do this in English by utilizing the space character, as most words in English are separated by a single space character. Next, we removed all special characters and stopwords, so we only had spoken words. Then, we have to stem the words into their basic form, and for this, various algorithms do that for us. Overall, the tokenizer provides us with an array of all the words in the message in the correct order, and we can use this array to understand the context of the message [15].

The next thing we have to understand is TF-IDF. If we want to match a document for a query, the document with the most appearances of a particular word will be the first result of the query. To prevent documents that misuse or misplace words to gain rank from appearing in our search, we can use TF-IDF to compare the word appearance against how much this word appears in other documents. In this way, a document that contains a word many times will have a smaller distance from one that contains the same word fewer times [15].

2.2 Related Work

This chapter presents papers relevant to our work, their approach to the problem of software quality evaluation, and their results. In the literature, there are many approaches to solving the problem of predicting the quality of software components. Some studies use association rules to prevent errors from incomplete changes, while others use classification to prioritize bugs or classify error-prone code [9], [10]. In addition, some papers use clustering to group good-quality software modules and detect bad ones. In contrast, other papers use text methods to extract bug fixes and where the bug occurred [9], [21]. The first subsection presents works that utilize unsupervised learning algorithms to evaluate software quality. In contrast, the second subsection presents supervised learning algorithms for software evaluation. The third subsection presents semisupervised algorithms that use both methods to address the problem of software quality evaluation. Lastly, the fourth subsection presents works on how to mark

bug-fix git commits that aid us in marking files that contain bugs and studying their characteristics.

2.2.1 Unsupervised Learning Software Quality Evaluation

Some papers extract software metrics from the code files and utilize clustering methods to assign labels to them. Then, those labels and software metric characteristics are evaluated by a group of software experts and assigned whether the code is hard to maintain. In the paper of Antonellis et al., their methodology was to extract the metrics, assign weights to them, perform clustering, and then study each cluster file characteristic and mark which clusters contain hard-to-maintain code. The software evaluators adjust the weights to produce the expected result, and they propose that those weights represent their knowledge that can be utilized in other projects, too [11]. In another paper by Zhong et Al., they suggest that collecting fault measurement data is a tedious process, so using experts to perform clustering analysis to define clusters of code that display problematic characteristics can prove a valuable solution to the error-prone problem that is robust to outliers [10]. Arshad and Tjortjis propose an automated way to identify hard-to-maintain software components. They use data mining algorithms and software metrics to discover hidden patterns in the source files and plot the extracted groups and their statistics. They concluded that the plots could help a manager understand which cluster groups hide potential high complexity that might be a hotspot for faulty and hard-to-maintain code [12].

2.2.2 Supervised Learning Software Quality Evaluation

Examples in the literature use classifiers to predict software quality using software metrics combined with cluster labels. Khoshgoftaar's work uses linear models and clustering classification models, and those models in majority voting ensembles. In their work, they reached around 25% type A and type B errors and had interesting findings regarding using voting models, but they had a limited dataset, and their training process could not scale to large datasets [13]. Again, the paper proposed by Zhong et Al. demonstrated supervised learning with features extracted from a classification labeling with experts where they showed 25% False Positive and False Negative rates and utilized a large number of algorithms from linear to trees and neural networks, but only used two annotated datasets for their results and the process is not automated [10].

2.2.3 Semisupervised Learning Software Quality Evaluation

In the paper of Papas and Tjortjis, they propose a new approach to evaluate software quality. Their approach uses metrics to perform unsupervised clustering and annotate clusters with complex code. In the next step, they use classification algorithms to learn the characteristics of those clusters and train a model. In addition, they used IQR outlier detection to remove extreme values and further improve the model's performance [22].

2.2.4 Marking Bug Commits

Many papers in the literature try to provide a solution to detect and mark known defects in software components [21]. We can extract information about those bugs from bug-tracking tools, the source code, or the commit messages the developers populate during the development of an application [21]. The commit messages show incredible potential for marking fault code because they describe the developer's tasks, and if we utilize version control difference tools, the code changes [21]. Gyimesi et al.'s research created a framework for collecting bugs in source repositories by looking at the tags attached to the reported issues on GitHub. However, they discussed using commit messages and diff to mark files containing bugs [21]. In the work of Casalnuovo et al., they try to classify git commits using regular expressions. Their proposed methodology is three steps: extracting the commits, processing the text, and patch referencing. In the first step, they extract all the commits information. In the second step, they perform an NLP analysis to detect the words that refer to bug-fixing commits. In the last step, they parsed the code difference, and the process used regular expressions to detect the class of the commit [23]. Lastly, in the work of Zafar et al., they propose a framework to mark bug commits using Bidirectional Encoder Representations from Transformers (BERT). In their paper, they finetuned the BERT model to understand bug-fixing commits, and they showed 92% accuracy compared to 84% accuracy of keyword-based solutions [24].

3 Methodology

During our literature research, we observed that most work uses a narrow selection of datasets to train their models, thus making our efforts to create a framework that automatically creates the dataset and training of the models more critical [25], [26]. The proposed methodology

consists of four procedures: the construction of the dataset, the extraction of features, the training and finetuning of the models, and the evaluation of the model's performance. In the first procedure, we created the dataset as the ground truth to train the models and evaluate their performance. In the second procedure, we enhanced the dataset with features to check if we could improve the models. In the third procedure, we trained and finetuned the models; in the last, we evaluated their performance and picked the best-performing model.

3.1 Data Collection

The procedure of dataset construction is divided into five steps: the creation of the repositories sample, the collection of the commit messages and their metadata, the NLP analysis of the commit messages, the extraction of the software metrics, and the merging of all the information produced from steps three and four.

The data used in this research originate from three sources: the crates.io database, the SEART GitHub database, and the GitHub API. The crates.io database contains many open-source Rust libraries and valuable metadata (<https://crates.io/data-access>). The SEART GitHub database is an up-to-date metadata collection from all the public GitHub repositories with ten or more stars [27]. We used this database because it provides an easy way to extract information for all repositories of Rust code through its filtering functionality. Lastly, the GitHub API fetches information related to repositories not covered by the SEART GitHub database or the repository metadata [27]. This information is related to pull requests and reported issues referenced in the commit messages. We collected the data 2023-07-04 for the crates.io database, 2023-07-12 from the GitHub repository database, and 2023-07-31 for repositories and all their related metadata (pull requests and issues).

3.1.1 Sample Repositories

Since there are thousands of repositories on public code repositories like GitHub, we created a sample of repositories representing the total population. In order to satisfy our sampling strategy, we used the metadata from crates.io and SEART GitHub to pick a dataset with high-quality repositories that try to include the characteristics of the total population as much as possible. We did this to achieve acceptable performance at a reasonable train time. This step is divided further into five stages. We collected and merged the metadata from the SEART and crates.io databases in the first two stages. In the third stage, we created a new

feature to assist in picking a uniform sample. In the fourth stage, we removed the noise; in the fifth stage, we sampled the final repositories list.

In the first stage, we explored and cleaned the SEART dataset. The following filters were used (Language: Rust, Exclude Forks, Has Open Issues, Has Pull Request) for the query, resulting in a downloaded database containing metadata for 7919 Rust repositories. The provided dataset contained only repositories with ten or more stars; it does this to filter repositories with little development activity [27]. In addition, those repositories would only contribute a little to the research. Then, we removed the archived repositories, which refers to repositories that stopped their development activity, which excluded 265 repositories; after that, 7654 repositories remain. Because ethical and legal issues play an essential role in research, we only included repositories with permissive licenses (MIT and Apache). After applying all the mentioned rules, 5089 repository entries remained.

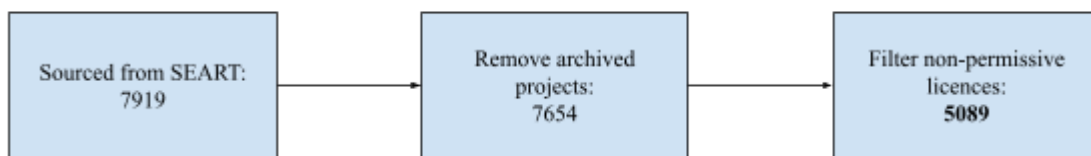


Figure 1. Sourcing repositories from the SEART database.

We merged the cleared data from the SEART GitHub database in the second stage with the crates.io database. The crates.io database consists of 104736 repositories, but after we excluded repositories hosted in alternative code repositories, like GitLab or BitBucket, 10346 were removed, and 94390 remained. Next, we merged entries of the list that had the same paths, ending up with 52904 remaining repositories. Then, the cleaned crates.io data were joined with the 5089 repository entries from the prepared SEART database, resulting in 2452 repositories. Lastly, repositories with missing values in the column of code lines were removed, excluding 822 repositories, resulting in 1630 repository entries after the filtering.

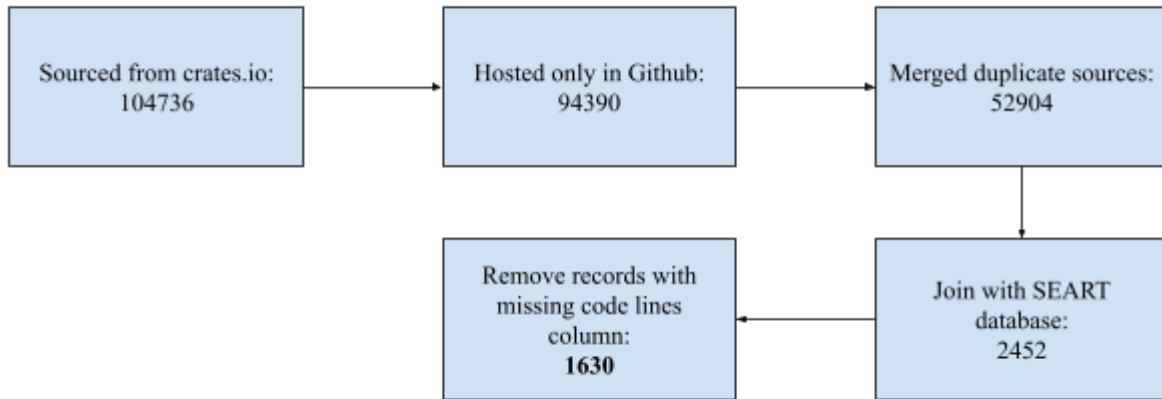


Figure 2. Sourcing repositories from crates.io database.

In the third stage, we created a new feature to separate the repositories into three groups to achieve a uniform selection of repositories. The perfect candidate for this is the total repository code lines because they allow all repositories to be selected, from large to small repositories and all in between. During the analysis, the charts showed that there are small repositories with lots of activity and large repositories with the same activity level. Still, those repositories have different characteristics regarding the software metrics, as seen in **Figure 5**. To achieve a more unbiased dataset that will result in more robust models, we created a feature called *code_size_group*, which separates the repositories into three groups: *small* with less than 2438 total lines, *medium* with more or equal to 2438 and less than 10340, and *large* with more than 10340 lines. As seen in **Figure 3**, the distribution of repositories across the *code_size_group* feature follows a uniform distribution.

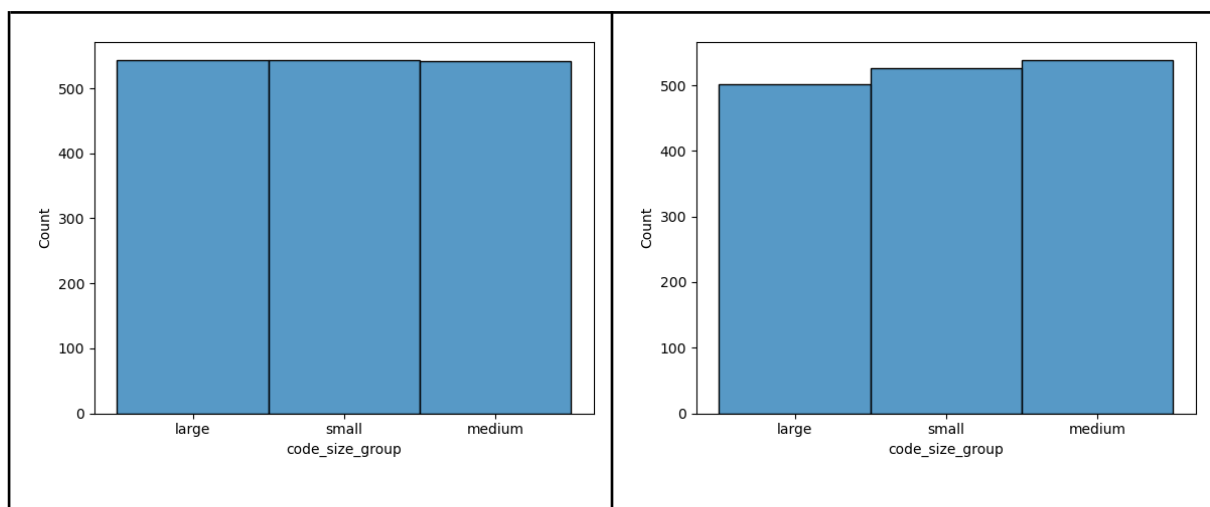


Figure 3. Before and after outlier removal number of repositories per *code_size_group*.

In the fourth stage, after some data exploration, it was evident that there were some repositories with an extreme number of commits activity, and by looking into their repositories, this activity is related to automatic bot commits. To clean the data from bot activity noise, we removed those outliers by using the *commits* feature column. A percentage target of 99.8% of the total distribution distribution is used for the outlier removal, trimming the extreme tails. In **Figure 4**, the distribution of commits activity before and after removing the outliers shows that removing outliers provides a better dataset.

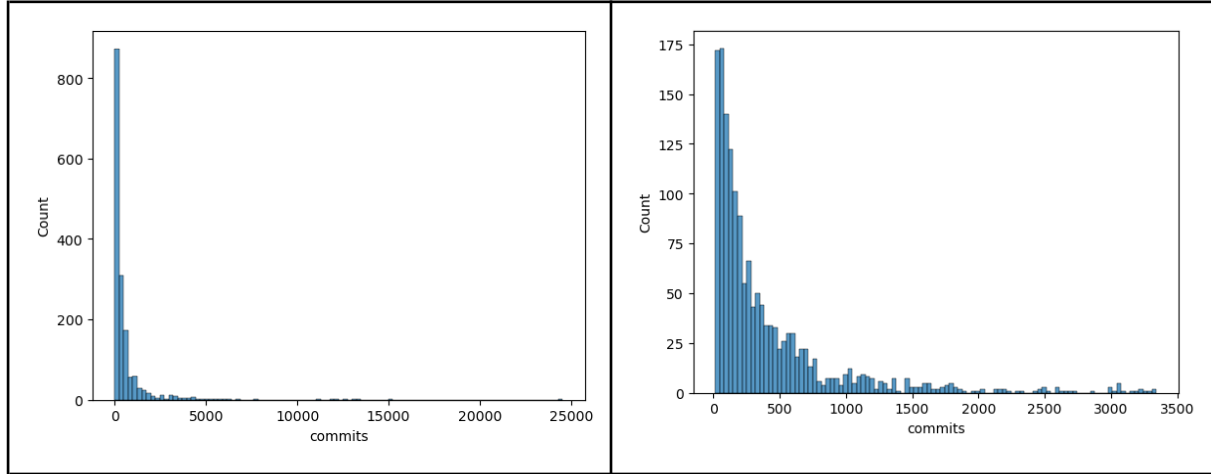
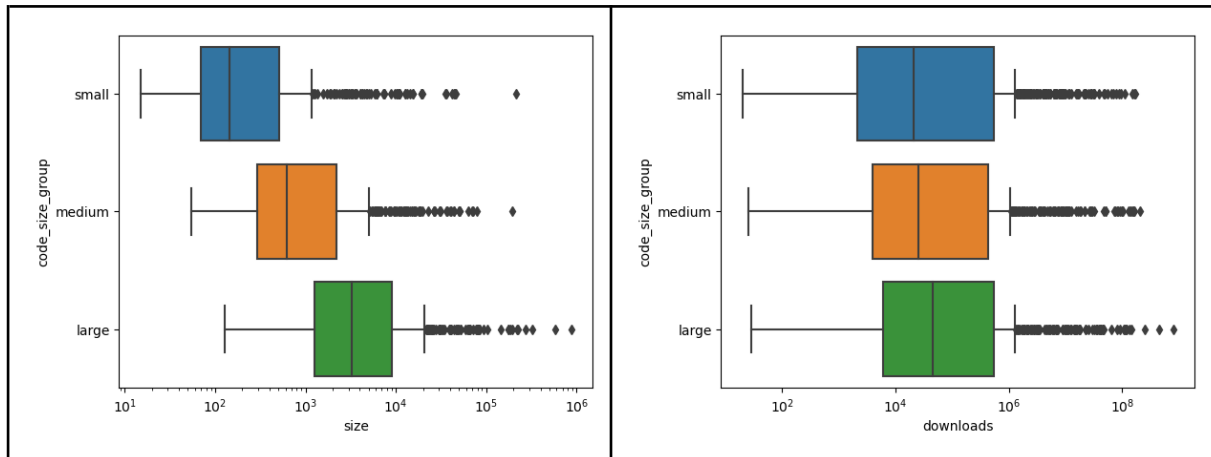


Figure 4. before and after outlier removal.

To verify if our proposed separation methodology effectively separated the repositories into three easily distinguished categories, we created various feature distributions of the dataset. Lastly, we observed that the proposed method provides a good criterion for separating the collection of repositories into three groups based on the current metadata.



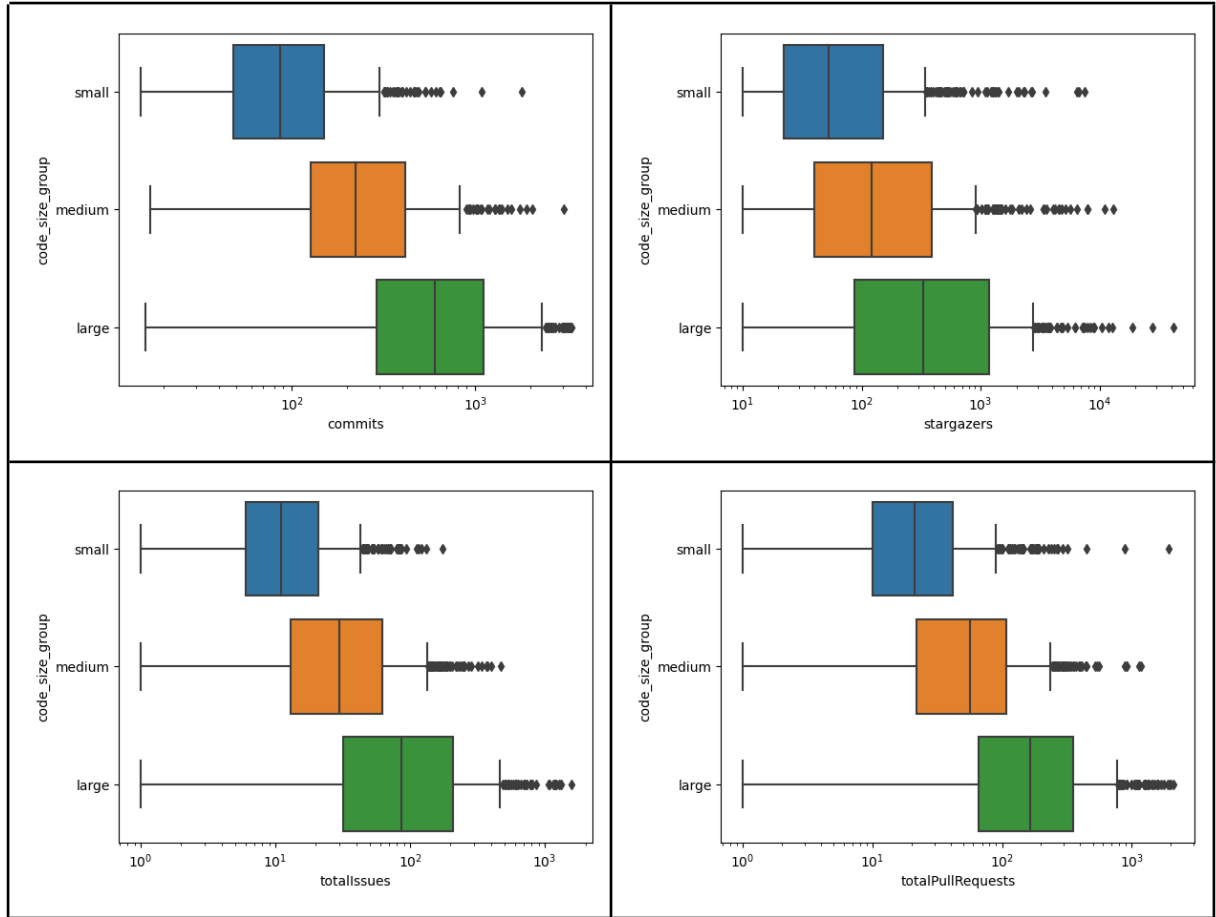


Figure 5. Comparing different feature distributions in contrast to *code_size_group*.

In the fifth stage, we sampled the final repositories list. Firstly, we sorted the repositories by *commits* and *code_size_group*. Then, we selected the first 110 from each *code_size_group* after manually skipping repositories that either had auto-generated commit activity from bots or their files generated automatically. Later, we regenerated the figures to determine if the sample could represent the more extensive list. Our sampling method could be better because it follows a hand-picked strategy using the top records of two characteristics [14], [27]. **Figure 6** shows that the repositories cover a wide range of the current metadata in the selected sample groups. If we compare it with **Figure 5**, we can observe that we achieved a good sampling.

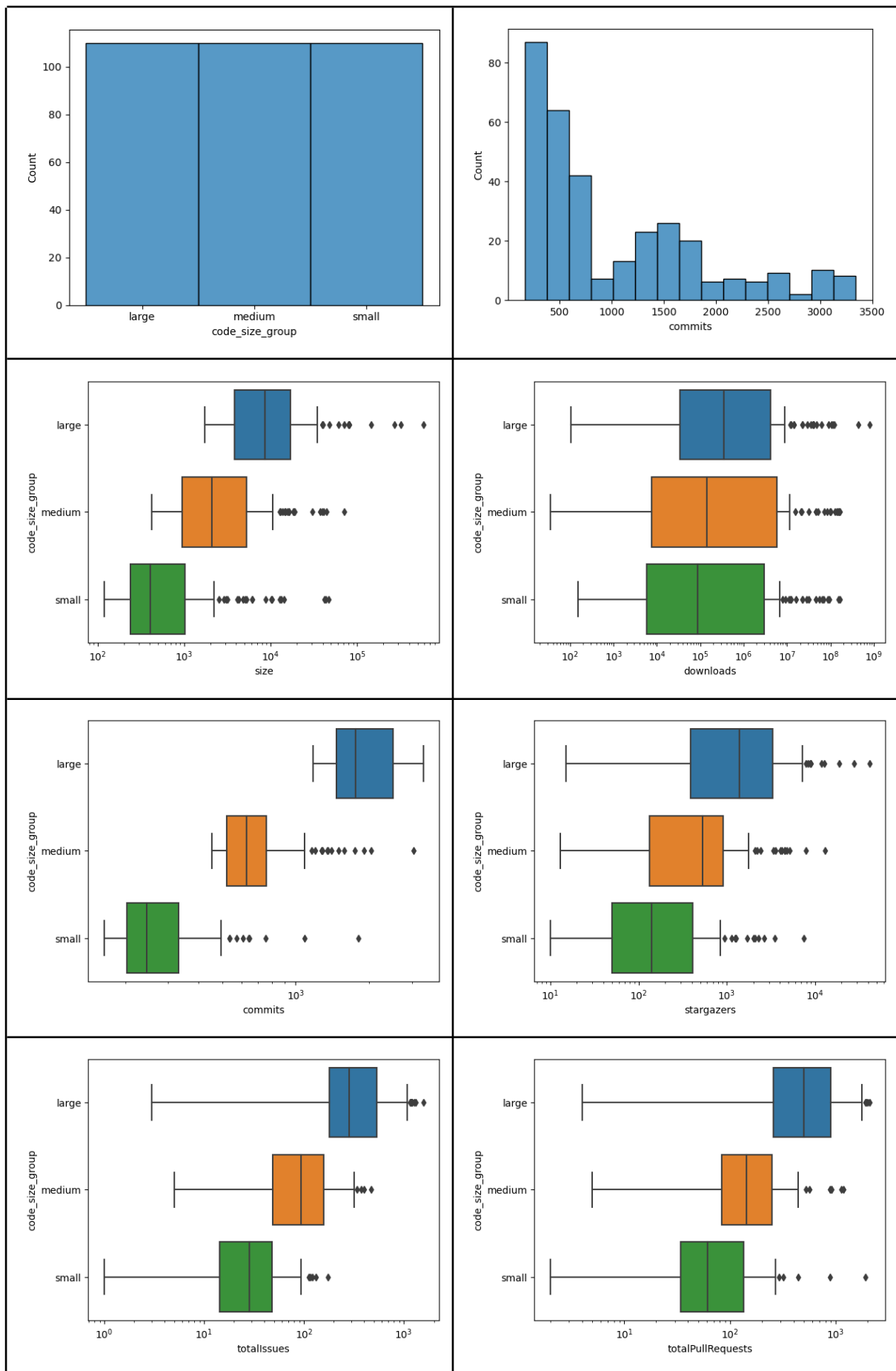


Figure 6. plots of repository features grouped by *code_size_group* of the final sample.

In conclusion, in this step, we created a sample that encapsulates as much information as possible from the wide range of characteristics the actual population holds. To achieve that, we combined metadata from two databases, removed entities with wrong details and outliers, separated the repositories into three groups based on their total lines of code, and sampled 110 repositories from each group, totaling 330 repositories for our sample size.

3.1.2 Collect Repositories Metadata

In this step, we collect all the available metadata for the repositories of the samples list. The available metadata are the code files, the commit messages, the issues, and the pull requests of the repositories. In the first stage, we downloaded all the repositories and their code files with the help of a simple bash script. The script takes as input the list of repositories and performs a git clone operation. Then, in the second stage, we extracted the commit messages for each repository and saved them into a CSV file. After that, we performed an exploratory analysis of the commit messages and observed that sometimes the commit messages reference pull requests or issues. So, in the third stage, we downloaded all the commits and pull request messages from the GitHub API of each repository and saved them into a CSV file. Lastly, in the fourth stage, we combined all the commit messages with their referenced issues and pull requests into a CSV file that contains 344420 commit messages. In **Figure 7**, there is a reduced description of the pipeline steps.

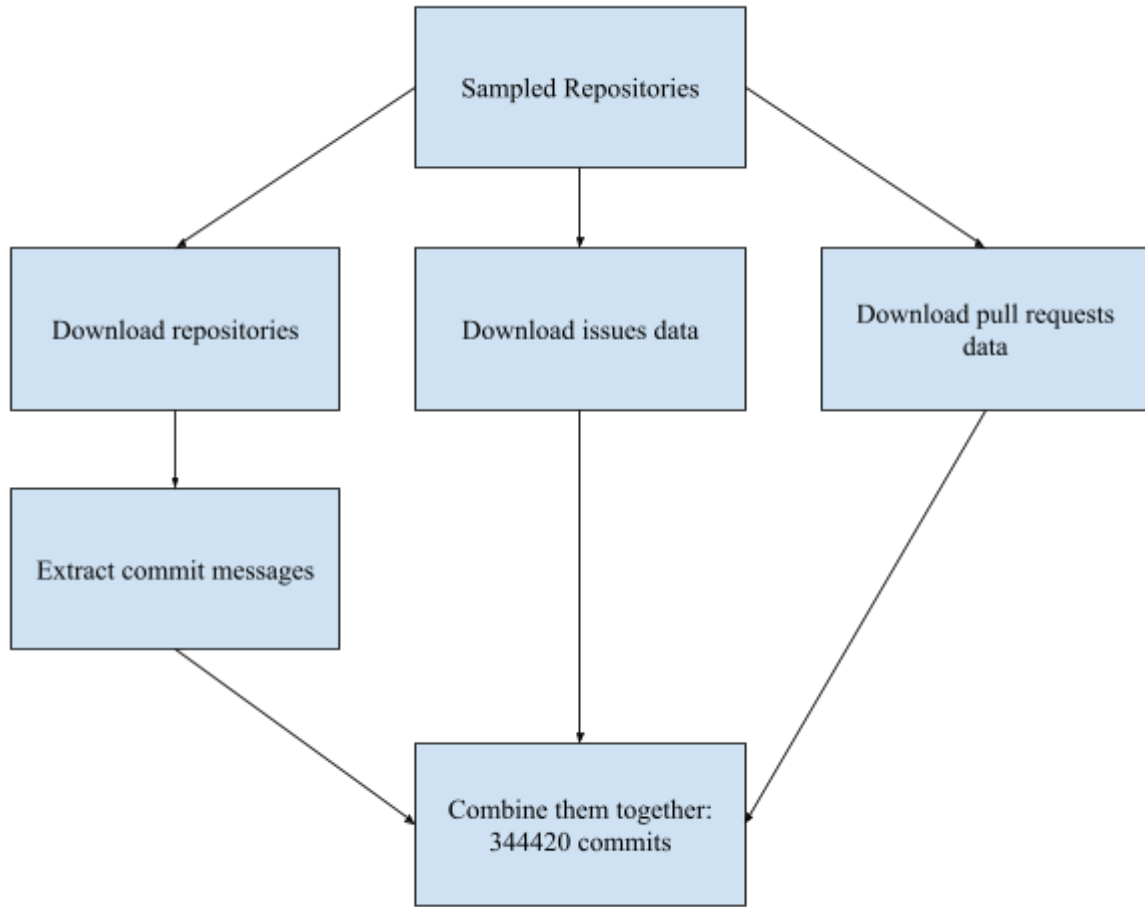


Figure 7. Commits metadata collection pipeline.

3.1.3 NLP Analysis of Commit Messages

After collecting all the commit messages of the repositories, we performed a Natural Language Processing analysis in this step to find keywords that determine if a commit message is a bug-fix or not. We utilized pandas with clustering algorithms from scikit-learn Python libraries [28], [29].

In the first step, we preprocessed the commit messages by removing stopwords, breaking the texts into arrays of tokens, and then vectorizing the tokens using TFIDF [23]. After that, we extracted the keywords by performing a loop of clustering the commit messages, extracting the keywords, and starting again. We used K-Means for the clustering algorithm and stopped the loop when the clusters reached the desired characteristics. Below are the details of the loop steps:

1. To find the optional cluster size, in this step, we perform K-Means from two to twenty cluster sizes, and for each cluster size, we calculate and store the sum of squares error.

Next, we plot the error to the cluster size chart and manually select the optimal cluster size at the elbow point [21].

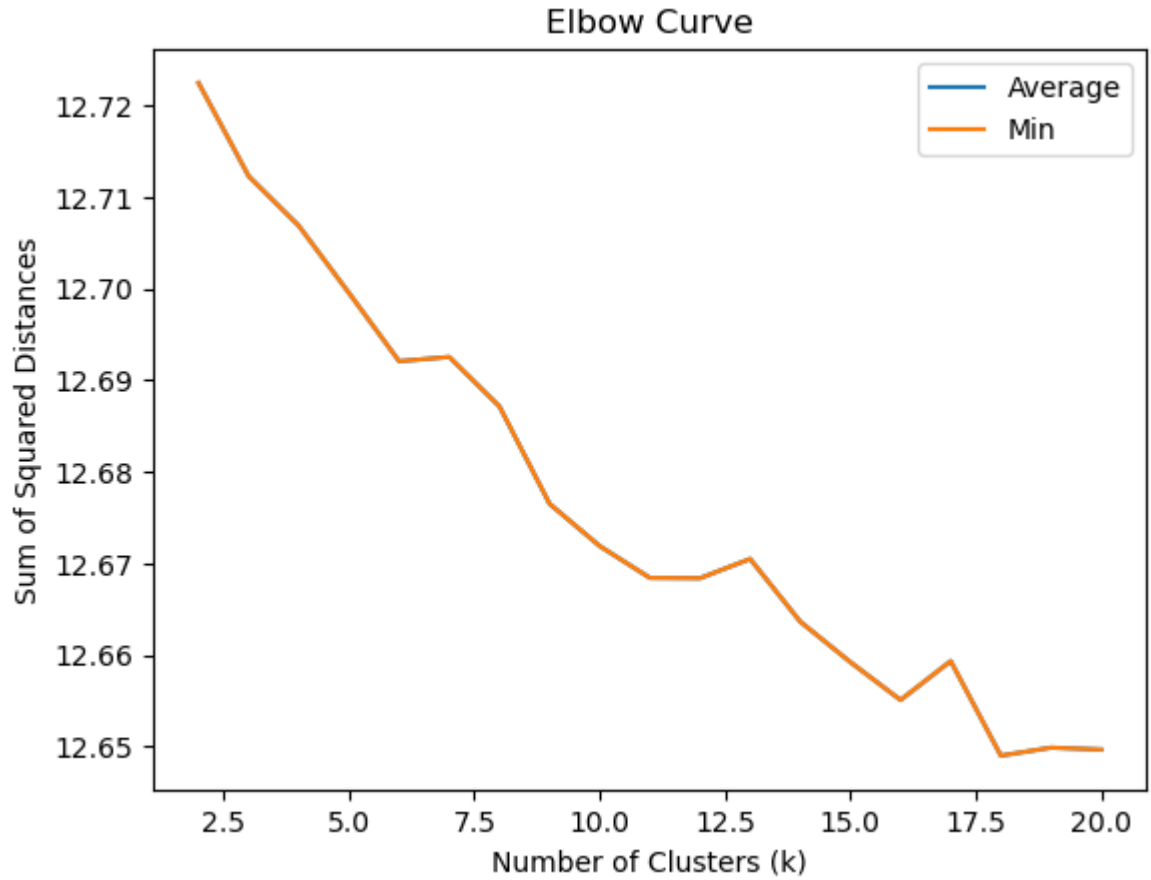


Figure 8. Elbow curve of the first run.

2. In this step, we recalculate the clusters using the cluster size found in the previous step and construct the word frequency diagram of each cluster.

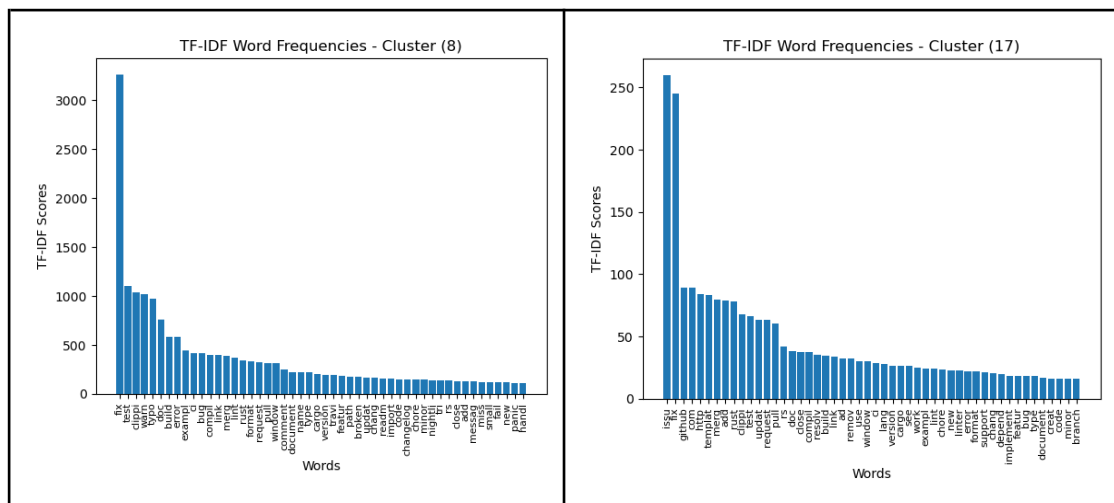


Figure 9. TF-IDF plot for cluster #8 and cluster #17.

3. In this step, by analyzing the frequency diagrams, we extract words for the include and exclude lists by observing their frequency and appearance in pairs. For example, in **Figure 9**, we can observe the appearance of the word *fix*, which is a word that indicates that this commit is a bug fix. However, this word can have a different meaning in another context. For example, as seen in Figure 9, “Fix Clippi” indicates that this commit does not fix a software bug but instead the software indentation. So we should proceed and add the word *clippi* to the skip words list.
4. To prepare for the next iteration, in this step, we remove the commit messages that contain words of the skip words list. Then, we plot the frequency diagrams to determine if the output fits our needs, and if it does, we stop the loop; if not, we continue to step one.

Our research efforts concluded with the following lists:

- Include: bug, fix, bug-fix, fixes, refactor
- Exclude: ci, build, doc, changelog, dependabot, lint, clippi, dep, chore, typo, test, fmt, readm, exampl, md, format, debug, remov, comment, name, compil, warn, harald, link, indent, window, rust, version, travi

3.1.4 Mark Bug-Fixing Commits

We used the keyword lists produced from the **3.1.3 subsection** to mark which commits are bug commits. We based this step on the work of Casalnuovo et al., who propose a framework for marking bug-fixing commits using regular expressions and NLP analysis. However, we kept only the last part because it was easier to implement and fulfilled the task of marking files that had a bug fix in the past, which is essential for our research [23]. We marked commits as bug commits when they contained any of the words in the include list but not if they contained any word from the exclude list. As a result, 24500 commits have a true value, and 319920 do not.

3.1.5 Extract Commit Hashes of Files

To assist our marking of files that appear in bug commits, in this step, we extracted the hashes for each file of the sampled repositories and collected them into a single file. We developed a script that iterates all the files of each repository folder, extracts the git commit history, collects the hashes, and appends them into a collection with information about the file.

3.1.6 Extract Software Analytics

We used the rust-code-analyzer to extract the software analytics of rust code, which we discussed in detail in the **2.1.2 subsection**. After extracting the data, we converted their format into a more friendly format for machine learning training, from the nested JSON to a flat vector format. Because the JSON produced by the tool is a nested tree of metric nodes, we wrote a function that converts each node into a separate data entry. The method extracted 520753 entries of different software components. The procedure also added the following features:

- id: unique number for each data entry
- size_group: the repository size group identifier (large, medium, small)
- repository: the name of the repository
- item_path: the location of the extracted data entry to assist in debugging
- has_spaces: if the entry is a parent of nested components
- spaces_len: the number of children's components
- max_depth: the max nest level
- nest_level: the current nest level

3.1.7 Merge Information

Each of the above steps produced different pieces of information, and in this step, we merged everything into a single data collection. Firstly, we merged the marked commits extracted from **subsection 3.1.4**, and the file hashes from **subsection 3.1.5**, with the commit hash as the join key. To keep only the unique file appearances, we sorted the files in order of whether they were in a buggy commit and then dropped the duplicates by keeping only the first appearance. By doing this, we kept whether the files appeared in at least one bug commit. Then, we merged the collection with the software metrics collection using the file name and repository as the join key. We concluded with a dataset of 520753 records of all the files of the repositories and their software metrics for their different components marked if they belong to a commit that performed a bug-fix.

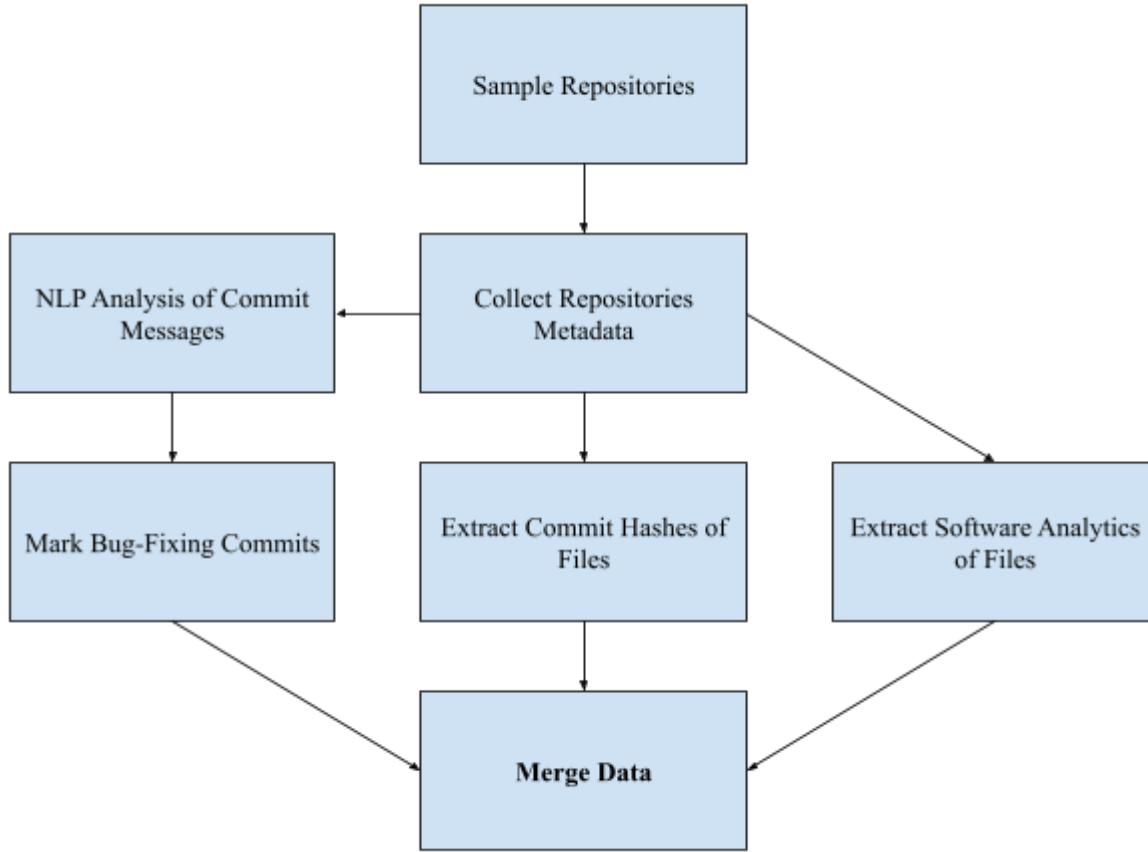


Figure 10. Overview of Data Collection Pipeline.

3.2 Feature Extraction

Frequently, due to insufficient data coding, the dataset may contain information not valuable to us. In addition, the data could contain hidden patterns or groups not visible with a simple statistics analysis. Thus, performing feature extraction can benefit us by providing extra information that we can use to increase the performance of the models. In the following subsections, we describe the procedure of feature extraction, which we divided into three steps: filtering the data, adding clustering features, and adding extra features.

3.2.1 Filtering

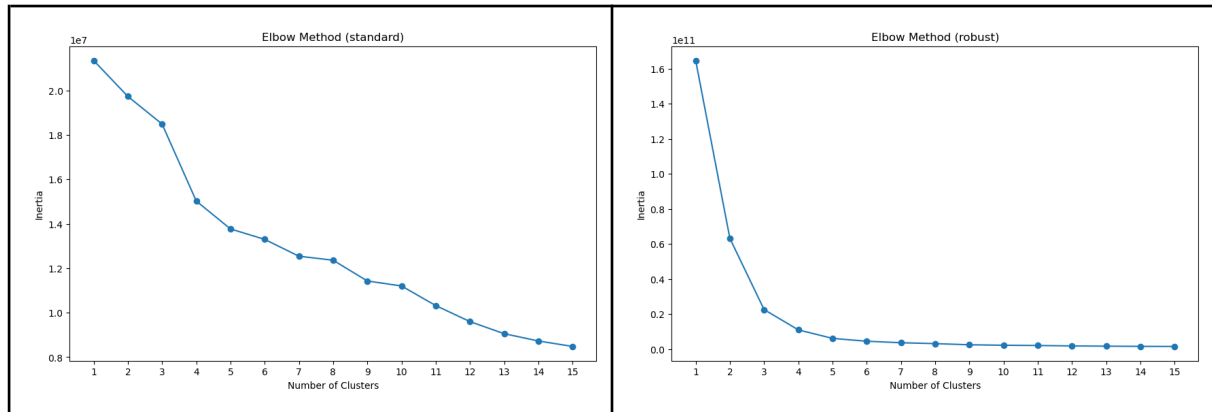
Before extracting features, we had to ensure that our data contained meaningful data, so in this step, we describe the filtering steps we performed on the dataset. Our following action was to handle missing values. We counted 23000 records that contained missing values, and to understand them better, we calculated some statistics of the lost value records. We observed that records with few lines of code failed to receive some metric values, and the metrics tool

documentation pointed out that some metrics require code structures to calculate a metric, so we proceeded and ignored records with five or fewer lines of code. In addition, items that contained only constraints or single line items failed to produce any metrics, so we removed those records. Those steps removed 21000 entities, and we removed the rest of the 2000. Lastly, in this step, we removed all the items that referred to the testing code, and as a result, we removed 27000 records in the process, leaving 266838 items in the end.

3.2.2 Clustering

To enrich our data further, we used various clustering algorithms to detect hidden groups. The clustering algorithms can group data entries with similar features, providing another piece of data for the classification algorithms. Those groupings are hard to detect with statistics analysis due to the amount of data; that is why we resort to clustering algorithms [15], [16].

The first clustering algorithm we used for classification was K Clusters, with Euclidean distance. To find the optimal cluster number, we utilized the elbow method. Moreover, we recorded the cluster groups with no data scaling with standard, robust, and min-max scalers to check whether extreme values influence the clustering. **Figure 10** shows that the optimal cluster size for the unscaled data is 6, for the standard 10, for the robust 4, and for the min-max 8. In addition, we can observe that the unscaled data does not display a solid convergence for an optimum grouping size.



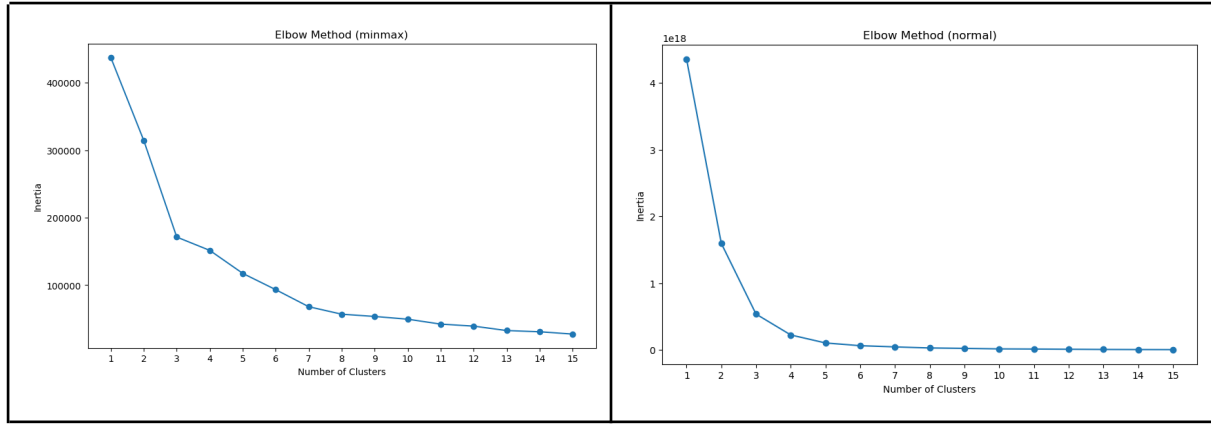


Figure 10. Elbow diagram for different scalers. Top left - None, top right - Robust, bottom left - Standard and bottom right - MinMax.

Lastly, we calculated the cluster groups using the DBScan algorithm, with `min_cluster_size` set to 1000, firstly with the StandardScaler and then with the MinMaxScaler. We used only those two scalers because when we tested the no-scaling and normal scaling.

3.2.3 Extra Features

In the last step of the procedure, we created some extra features derived manually from the dataset's data. Many datasets can contain information hidden in complex features. For example, we can extract a person's age from the date of birth, which can be a decisive factor in classification algorithms. Those features can contribute positively to the model's performance, so we performed this step to extract them. This subsection describes the features we extracted and how we did it.

- *has_spaces*: This feature is a boolean value, representing whether the entry is a parent of other software components (functions, structs, traits). In the data provided by the analysis tool, a field named *spaces* exists, and this field contains each child component for each software component, so we kept this name and did not refer to whether this entry contains space characters. For example, a function that defines a helper function in its body has a child.
- *max_depth*: This value contains the maximum depth of the entries' children. For instance, a trait that defines a function and that function defines another function will have a value of two.
- *spaces_len*: This value contains the number of children under the entity. For example, a trait with eight functions has a *space_len* of eight;

- *code_lines*: In the data provided by the tool, we do not have the lines of code the software component spans, but instead the starting and ending lines of code, so we extracted the number of lines from those two features.

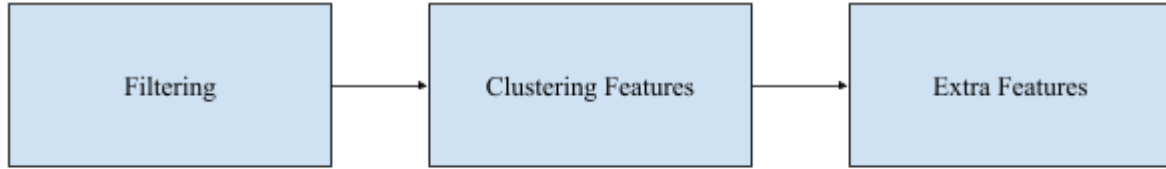


Figure 11. Filtering and Feature Engineering Pipeline Overview.

3.3 Data Configurations

We used six data configuration setups for our training and evaluation. Those configurations help us to test how different algorithms behave to varying amounts of data and detect which features impact the models the most. Moreover, those configurations assist us in determining which features contribute to the overfitting of the models if they are present and which ones do not contribute much and skip them altogether to improve the training time. Lastly, we can evaluate if the feature engineering had any effect on improving the model performance.

The first configuration is the entire dataset, which includes every feature. Next, the clusters dataset contains only the features from the clustering step, with the *size_group* and record *kind* feature, excluding the *code_lines* feature. Then, the primary dataset excludes the features we added from the feature engineering step and *code_lines* feature. Next, the plain collection contains only the raw data extracted from the tool without performing feature engineering. The following two configurations extend the plain dataset, but one contains the feature of *code_lines* of the whole project, and the other contains the feature of the project *size_group*.

3.4 Models' Training

In this section, we describe the proposed training framework of our methodology. We designed the pipeline to be flexible to use various configuration setups, models, and evaluation metrics. We used scikit-learn for the training pipeline algorithms and structure [29]. The models included Decision Tree, Random Forest, Perceptron, and XGBoost. We trained the models on the constructed dataset on each data configuration we mentioned. Moreover, we performed the training and finetuning of the models using various optimization

parameters, and we evaluated the models using several evaluation metrics, including accuracy, precision, recall, F1 score, and area under the curve (AUC). The following sections describe the model training algorithms, their optimization configuration, and how we set up the training pipeline.

3.4.1 Algorithms And Configuration

We used various training algorithms to find the perfect model to solve our problem. Each algorithm follows a different approach to solving the classification problem, with some overlaps between them. In total, we used ten algorithms with various optimization parameters.

Table 3 shows the algorithms we used and their optimization parameters.

Algorithm	Parameters	Algorithm	Parameters
Naive Bayes (GaussianNB)	None	Ridge	alpha: uniform(0, 10) solver: auto, svd, cholesky, lsqr, sparse_cg, sag, saga
SVM	C: uniform(0, 10) kernel: linear, poly, RBF, sigmoid degree: randint(1, 10) gamma: auto, scale, 0.1, 0.01, 0.001	K Nearest	n_neighbors: randint(1, 20) weights: uniform, distance p: 1, 2
Decision Tree	class_weight: balanced criterion: gini, entropy max_depth: randint(5, 30) min_samples_split: randint(2, 20) min_samples_leaf: randint(1, 20)	Random Forest	class_weight: balanced n_estimators: rand(range(5, 53, 2)) criterion: gini, entropy max_depth: randint(5, 30) min_samples_split: randint(2, 20) min_samples_leaf: randint(1, 20)
XGBoost	n_estimators: rand(range(5, 53, 2)) learning_rate: uniform(0.01, 4.9) gamma: uniform(0.0, 10.0) max_depth: randint(5, 15) reg_lambda: uniform(0.01, 9.99) reg_alpha: uniform(0.01, 9.99) max_delta_step: randint(0, 15) subsample: uniform(0.5, 0.5) colsample_bytree: uniform(0.5, 0.5) colsample_bylevel: uniform(0.5, 0.5) min_child_weight: uniform(0.5, 9.5) scale_pos_weight: uniform(0.01, 9.99)	Perceptron	'classifier_hidden_layer_sizes': [(10,), (25,), (50,), (75,), (100,), (10, 10), (25, 25), (50, 50), (100, 100), (50, 50, 50)], 'classifier_activation': ['relu', 'tanh', 'logistic'], 'classifier_alpha': 10.0 ** -np.arange(1, 7), 'classifier_learning_rate': ['constant', 'adaptive'],
Bagging	n_estimators: rand(range(5, 53, 2)) max_samples: uniform(0.1, 0.9) max_features: uniform(0.1, 0.9)	Boosting	n_estimators: rand(range(5, 53, 2)) learning_rate: uniform(0.01, 4.9) estimator__max_depth: randint(5, 30)

Table 3. The Classification algorithms we used and the optimization parameters.

3.4.2 Training Pipeline

In the training pipeline step, we take each algorithm and collect the score metrics of our choice. We used the plain data configuration to gather a baseline for each algorithm. We used standard scaling in the following algorithms: SVM because SVMs are sensitive to the scale of input features, and KNN, because it computes distances between data points, so features with larger scales, can dominate the distance calculations. We have not used any dimension reduction algorithms like PCA, and the reason is to prevent information loss and decrease performance. Because the data are balanced, we have not used data-balancing algorithms. We used a cross-validation of five to address errors in data sampling methods. Next, we used a randomized grid search to finetune the models and maximize their performance metrics. Lastly, we extracted the following metrics for each parameter set used in the random grid search algorithm: accuracy, balanced_accuracy, average_precision, f1, f1_micro, f1_macro, f1_weighted, precision, precision_micro, precision_macro, precision_weighted, recall, recall_micro, recall_macro, recall_weighted, roc_auc.

3.5 Evaluation

To evaluate which algorithm is the best, we performed the following steps to extract data to aid us in this decision. The first step is determining the best parameter configuration from the plain dataset results for each model. Then, we extract the following performance metrics for each model and data configuration: accuracy, precision, recall, f1, and roc_auc. To prevent overfitting, we selected the parameters where the difference between the train and test scores is less than or equal to 1%. Ultimately, we have 15 configurations for each algorithm and data configuration. To evaluate which configuration and algorithm are the best, we combined all the data collected from the previous step into a big table to aid our comparisons. In addition, we constructed the confusion matrix for each set and calculated which features influence the model the most.

3.5.1 Accuracy

Looking at the confusion matrixes for accuracy in the appendix, we can observe that this metric favors the primary class of the problem. The literature discusses that balanced accuracy is better because it compares both classes' average performance [16]. We suggest against using a model trained only based on accuracy because its overall performance might be satisfactory.

3.5.2 Precision

By looking at the confusion matrixes for precision in the appendix, we can observe that precision tries to create a model that eliminates False Positives but increases the chance of False Negatives. We suggest using a model optimized for precision to eliminate not detecting a code module that might require maintenance. However, in the process, we might flag more modules for maintenance that might not require it.

3.5.3 Recall

On the other hand, by looking at the confusion matrixes for recall in the appendix, we can observe that recall tries to create a model that eliminates False Negatives but increases the chance of False Positives. We suggest using a model optimized for recall to eliminate the unnecessary marking for maintenance of code modules that might not require it. However, in the process, we increase the chance of missing a code module that might require maintenance.

3.5.4 F1 and AUC

Both metrics perform similarly, and no trend is observed in the classification when using those metrics. We suggest using a model optimized on those metrics because it gives a middle ground between recall and precision and can act as a general classifier that can detect code modules that require maintenance or not. In some cases, the models created optimized for F1 perform better than those optimized for AUC and the opposite. In our case, we observed that for XGBoost, F1 showed better results, so we suggest using this metric.

4 Results

This chapter presents our findings and simple insights and observations for them. We start by listing all the classification algorithms we rejected and why they did not produce an acceptable result. The following sections present the findings for the rest of the algorithms optimized for the classification metrics of accuracy, precision, recall, F1, and AUC metrics. In the next section, we analyze the feature importance of the best-performing model. The last section presents a brief model explainability results for the Decision Tree algorithm.

4.1 Rejected Algorithms

We would not present the results for the models Naive Bayes, Ridge, SVM, KNearest, and Boosting because their evaluation metrics do not show that the models can model the classification problem. The problem does not seem linearly separable, so Naive Bayes, Ridge, and SVM cannot work with this problem [15], [16], [29]. Regarding the KNearest algorithm, because of the vast amount of information, the algorithm took very long to complete, so we skipped the collection of the results for this algorithm. However, we performed a small-scale test for the KNearest algorithm that showed no significant results.

4.2 Accuracy Score

Decision Tree performs well across different data configurations, with its highest accuracy in the “Basic” configuration. Random Forest outperforms other algorithms in the “Plain,” “Basic,” and “Size Group” configurations. XGBoost excels in the “Code Lines” and “Full” configurations. Bagging performs consistently but must show the highest accuracy in any specific configuration. Decision Tree and Random Forest have relatively low standard deviations, indicating performance stability. XGBoost, Bagging, and Perceptron show higher variability, especially the Perceptron algorithm. Perceptron is the least accurate algorithm in all configurations, suggesting limitations in capturing complex patterns present in the data. The same anomaly applies to Random Forest at the Code lines and Full data configurations. The model capacity plays a vital role in explaining those anomalies, and the literature suggests that we can fix it by making the models have more parameters and depth [15].

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	68.53%	69.01%	76.41%	68.98%	68.30%	76.11%
Random Forest	69.48%	69.82%	71.99%	69.92%	68.71%	70.97%
XGBoost	69.13%	69.60%	77.63%	69.43%	69.13%	78.35%
Bagging	68.84%	69.32%	75.92%	69.40%	69.05%	75.54%
Perceptron	58.47%	61.97%	51.16%	62.13%	60.62%	51.18%

Table 4. Accuracy score of the tested algorithms and data configurations.

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	00.16%	00.19%	00.12%	00.16%	00.06%	00.23%
Random Forest	00.19%	00.12%	00.25%	00.11%	00.18%	00.20%
XGBoost	00.20%	00.10%	00.58%	00.15%	00.14%	00.22%
Bagging	00.14%	00.06%	00.19%	00.10%	00.29%	00.12%
Perceptron	03.75%	02.15%	00.08%	02.54%	05.90%	00.11%

Table 3. Standard deviation of the tested algorithms and data configurations.

Algorithm	Parameters
Decision Tree	'classifier__criterion': 'gini', 'classifier__max_depth': 10, 'classifier__min_samples_leaf': 18, 'classifier__min_samples_split': 6
Random Forest	'classifier__criterion': 'gini', 'classifier__max_depth': 9, 'classifier__min_samples_leaf': 13, 'classifier__min_samples_split': 18, 'classifier__n_estimators': 39
XGBoost	'classifier__colsample_bylevel': 0.93, 'classifier__colsample_bytree': 0.66, 'classifier__gamma': 9.54, 'classifier__learning_rate': 0.60, 'classifier__max_delta_step': 5, 'classifier__max_depth': 8, 'classifier__min_child_weight': 8.36, 'classifier__n_estimators': 15, 'classifier__reg_alpha': 9.91, 'classifier__reg_lambda': 9.51, 'classifier__scale_pos_weight': 1.38 'classifier__subsample': 0.77
Bagging	'classifier__max_features': 0.64, 'classifier__max_samples': 0.73, 'classifier__n_estimators': 47,

	'classifier__estimator__max_depth': 8
Perceptron	'classifier__learning_rate': 'constant', 'classifier__hidden_layer_sizes': (50, 50, 50), 'classifier__alpha': 1e-06, 'classifier__activation': 'relu'

Table 5. Optimal parameters of each algorithm that maximizes the accuracy metric.

4.3 Precision Score

Decision Tree and Bagging show moderate precision scores, with Decision Tree performing better in configurations like “Code Lines” and “Full.” Perceptron has varying precision scores, with deficient scores in some configurations, particularly in “Full,” due to its limited capacity to model the complex dataset. XGBoost consistently outperforms other algorithms across all data configurations, achieving exceptionally high precision scores close to 100%. However, after closely looking at the confusion matrix in the appendix, it overfitted the data to match the True class only. That problem can be solved using a balanced metric like macro_precision [16].

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	63.95%	64.06%	71.15%	63.84%	63.94%	71.13%
Random Forest	66.99%	67.84%	67.11%	67.48%	69.44%	68.35%
XGBoost	99.89%	99.87%	99.64%	99.87%	99.87%	99.79%
Bagging	64.25%	65.93%	71.73%	65.66%	65.35%	71.70%
Perceptron	64.61%	76.64%	23.16%	80.50%	70.38%	0%

Table 6. Precision score of the tested algorithms and data configurations.

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	00.23%	00.21%	00.08%	00.22%	00.79%	00.98%

Random Forest	00.16%	00.30%	00.49%	00.20%	00.24%	00.31%
XGBoost	00.06%	00.03%	00.06%	00.03%	00.07%	00.09%
Bagging	00.12%	00.14%	00.22%	00.14%	00.32%	00.16%
Perceptron	13.73%	15.21%	28.38%	13.37%	10.99%	0%

Table 7. Standard deviation of the tested algorithms and data configurations.

Algorithm	Parameters
Decision Tree	'classifier__criterion': 'gini', 'classifier__max_depth': 9, 'classifier__min_samples_leaf': 4, 'classifier__min_samples_split': 3
Random Forest	'classifier__criterion': 'entropy', 'classifier__max_depth': 5, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split': 19, 'classifier__n_estimators': 33
XGBoost	'classifier__colsample_bylevel': 0.68, 'classifier__colsample_bytree': 0.90, 'classifier__gamma': 2.78, 'classifier__learning_rate': 1.04, 'classifier__max_delta_step': 8, 'classifier__max_depth': 7, 'classifier__min_child_weight': 9.98, 'classifier__n_estimators': 19, 'classifier__reg_alpha': 9.94, 'classifier__reg_lambda': 0.90, 'classifier__scale_pos_weight': 0.03, 'classifier__subsample': 0.52
Bagging	'classifier__max_features': 0.64, 'classifier__max_samples': 0.73, 'classifier__n_estimators': 47, 'classifier__estimator__max_depth': 8
Perceptron	'classifier__learning_rate': 'adaptive', 'classifier__hidden_layer_sizes': (10, 10), 'classifier__alpha': 0.01, 'classifier__activation': 'logistic'

Table 8. Optimal parameters of each algorithm that maximizes the precision metric.

4.4 Recall Score

XGBoost achieves perfect recall (100%) in most configurations, except for the “Full” configuration, which drops significantly. However, after looking at the confusion matrix in the appendix, we observed that the model overfitted the primary class. The literature suggests solving that using a balanced metric like macro_recall [16]. Random Forest consistently demonstrates good recall across various data configurations, with the best recall at “Code Lines.”

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	83.06%	82.17%	84.97%	81.88%	79.91%	84.88%
Random Forest	81.20%	78.83%	84.95%	79.36%	70.48%	78.78%
XGBoost	100%	100%	100%	100%	100%	26.40%
Bagging	76.78%	76.03%	82.25%	75.96%	73.80%	73.89%
Perceptron	46.78%	29.42%	0%	19.84%	72.40%	0%

Table 9. Recall score of the tested algorithms and data configurations.

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	02.33%	00.71%	00.83%	01.22%	03.56%	02.85%
Random Forest	00.60%	00.74%	00.52%	00.66%	01.26%	01.28%
XGBoost	0%	0%	0%	0%	0%	3678%
Bagging	00.22%	00.63%	00.38%	00.35%	00.61%	00.48%
Perceptron	41.63%	36.07%	0%	00.98%	27.78%	0%

Table 10. Standard deviation of the tested algorithms and data configurations.

Algorithm	Parameters
Decision Tree	'classifier__criterion': 'entropy', 'classifier__max_depth': 10, 'classifier__min_samples_leaf': 18, 'classifier__min_samples_split': 6
Random Forest	'classifier__criterion': 'gini', 'classifier__max_depth': 9, 'classifier__min_samples_leaf': 13, 'classifier__min_samples_split': 18, 'classifier__n_estimators': 39
XGBoost	'classifier__colsample_bylevel': 0.84, 'classifier__colsample_bytree': 0.88, 'classifier__gamma': 8.53, 'classifier__learning_rate': 4.77, 'classifier__max_delta_step': 7, 'classifier__max_depth': 11, 'classifier__min_child_weight': 4.24, 'classifier__n_estimators': 51, 'classifier__reg_alpha': 9.45, 'classifier__reg_lambda': 0.49, 'classifier__scale_pos_weight': 0.02, 'classifier__subsample': 0.80
Bagging	'classifier__max_features': 0.19, 'classifier__max_samples': 0.51, 'classifier__n_estimators': 45, 'classifier__estimator__max_depth': 8
Perceptron	'classifier__learning_rate': 'adaptive', 'classifier__hidden_layer_sizes': (10, 10), 'classifier__alpha': 1e-05, 'classifier__activation': 'logistic'

Table 11. Optimal parameters of each algorithm that maximizes the recall metric.

4.5 F1 Score

Because the F1 score is the harmonic mean of precision and recall, we eliminate the phenomenon of overfitting the major or minor class so that we will have a more straightforward look at the best-performing model. XGBoost consistently achieves the highest F1 scores across all data configurations, demonstrating its effectiveness in balancing precision and recall. Decision Tree and Random Forest show competitive F1 scores, with Decision Tree performing slightly better in most configurations. Bagging performs well, with F1 scores close to Decision Tree and Random Forest. XGBoost exhibits relatively stable standard

deviations, indicating a stable model. Again, Perceptron fails to model the data due to its limited capacity.

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	72.03%	72.21%	77.85%	72.11%	70.95%	77.57%
Random Forest	72.25%	71.91%	74.91%	72.08%	68.45%	72.55%
XGBoost	75.20%	75.32%	80.35%	75.32%	75.23%	80.15%
Bagging	71.94%	71.07%	77.27%	71.46%	71.15%	76.74%
Perceptron	34.61%	34.88%	0%	49.15%	44.78%	0%

Table 12. F1 score of the tested algorithms and data configurations.

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	00.60%	00.27%	00.20%	00.35%	01.12%	00.59%
Random Forest	00.22%	00.16%	00.22%	00.19%	00.24%	00.26%
XGBoost	00.08%	00.08%	00.03%	00.12%	00.10%	00.18%
Bagging	00.19%	00.09%	00.17%	00.09%	00.21%	00.11%
Perceptron	28.59%	20.64%	0%	24.69%	14.42%	0

Table 13. Standard deviation of the tested algorithms and data configurations.

Algorithm	Parameters
Decision Tree	'classifier__criterion': 'entropy', 'classifier__max_depth': 10, 'classifier__min_samples_leaf': 3, 'classifier__min_samples_split': 10
Random Forest	'classifier__criterion': 'gini',

	'classifier__max_depth': 9, 'classifier__min_samples_leaf': 5, 'classifier__min_samples_split': 13, 'classifier__n_estimators': 39
XGBoost	'classifier__colsample_bylevel': 0.79, 'classifier__colsample_bytree': 0.51, 'classifier__gamma': 9.88, 'classifier__learning_rate': 0.21, 'classifier__max_delta_step': 9, 'classifier__max_depth': 12, 'classifier__min_child_weight': 5.72, 'classifier__n_estimators': 51, 'classifier__reg_alpha': 3.16, 'classifier__reg_lambda': 3.93, 'classifier__scale_pos_weight': 2.38, 'classifier__subsample': 0.55
Bagging	'classifier__max_features': 0.64, 'classifier__max_samples': 0.73, 'classifier__n_estimators': 47, 'classifier__estimator__max_depth': 8
Perceptron	'classifier__learning_rate': 'constant', 'classifier__hidden_layer_sizes': (10,), 'classifier__alpha': 1e-05, 'classifier__activation': 'logistic'

Table 14. Optimal parameters of each algorithm that maximizes the f1 metric.

4.6 Area Under Curve Score

AUC can act as an unbiased metric between different classes, so those results give us another unbiased view of which model is the best. XGBoost consistently achieves the highest AUC scores across all data configurations, indicating its effectiveness in classification tasks. Decision Tree, Random Forest, and Bagging show competitive AUC scores, with Decision Tree performing slightly better in most configurations. Nevertheless, if the data are limited, the Random Forest performs better than the Decision Tree classifier. Perceptron has lower AUC scores across all configurations, suggesting challenges in distinguishing between positive and negative instances. In addition, Perceptron has the highest standard deviations among all algorithms, indicating significant variability in AUC performance across different data configurations. XGBoost exhibits relatively lower standard deviations, indicating consistent AUC performance across different data configurations.

Model \	Plain	Size	Code	Basic	Clusters	Full
---------	-------	------	------	-------	----------	------

Data Config		Group	Lines			
Decision Tree	77.11%	77.51%	85.43%	77.51%	76.81%	85.21%
Random Forest	77.96%	78.55%	81.01%	78.53%	77.61%	79.36%
XGBoost	78.17%	78.68%	84.94%	78.82%	77.96%	85.15%
Bagging	78.12%	78.69%	85.84%	78.70%	78.27%	85.46%
Perceptron	62.89%	69.22%	50.05%	67.88%	68.22%	50.07%

Table 15. Area Under Curve score of the tested algorithms and data configurations.

Model \ Data Config	Plain	Size Group	Code Lines	Basic	Clusters	Full
Decision Tree	00.27%	0.021%	00.13%	00.17%	00.44%	00.18%
Random Forest	00.19%	00.16%	00.38%	00.15%	00.15%	00.27%
XGBoost	00.19%	00.08%	00.43%	00.19%	00.20%	00.35%
Bagging	00.18%	00.14%	00.14%	00.15%	00.23%	00.17%
Perceptron	06.72%	02.33%	06.72%	02.33%	00.09%	04.88%

Table 16. Standard deviation of the tested algorithms and data configurations.

Algorithm	Parameters
Decision Tree	'classifier__criterion': 'entropy', 'classifier__max_depth': 9, 'classifier__min_samples_leaf': 8, 'classifier__min_samples_split': 11
Random Forest	'classifier__criterion': 'gini', 'classifier__max_depth': 8, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split': 14, 'classifier__n_estimators': 43

XGBoost	'classifier__colsample_bylevel': 0.58, 'classifier__colsample_bytree': 0.86, 'classifier__gamma': 6.59, 'classifier__learning_rate': 0.14, 'classifier__max_delta_step': 6, 'classifier__max_depth': 7, 'classifier__min_child_weight': 2.695, 'classifier__n_estimators': 19, 'classifier__reg_alpha': 1.05, 'classifier__reg_lambda': 8.00, 'classifier__scale_pos_weight': 1.79, 'classifier__subsample': 0.83
Bagging	'classifier__max_features': 0.64, 'classifier__max_samples': 0.73, 'classifier__n_estimators': 47, 'classifier__estimator__max_depth': 8
Perceptron	'classifier__learning_rate': 'constant', 'classifier__hidden_layer_sizes': (50, 50, 50), 'classifier__alpha': 1e-06, 'classifier__activation': 'relu'

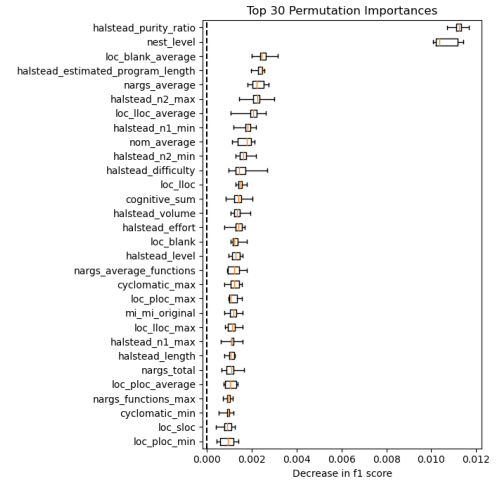
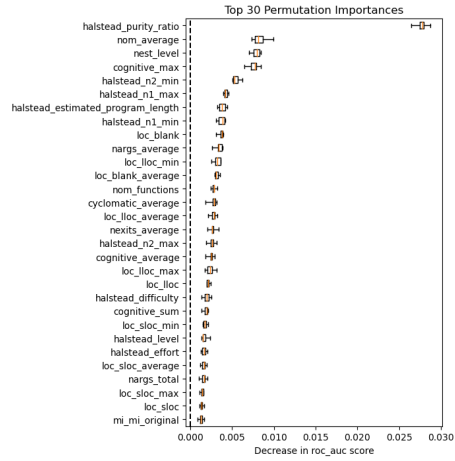
Table 17. Optimal parameters of each algorithm that maximizes the AUC metric.

4.7 Important Software Metrics

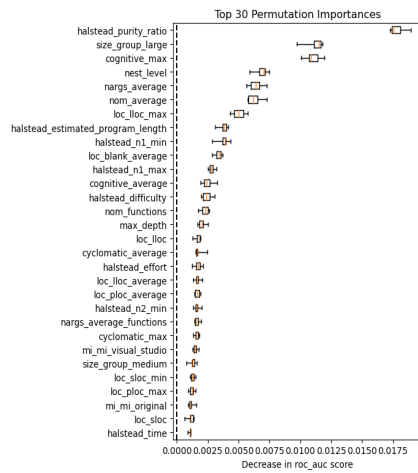
After examining the feature importance charts below, we conclude that the *Halstead purity ratio* is the most important feature on the dataset, followed by *nom_average* and nest level. The *cognitive* and *cyclomatic* metrics also play an essential role in various configurations. The *Nargs* feature also impacts some cases where the *size group* is preset, which can positively affect the decision if the *size group* is “large.” The feature *kind* impacts the models' decision if it equals “unit.” The cluster features that had the most significant impact on the models' decision were DBScan complex standard / min-max ID=1 and DBScan complex standard ID=4. Lastly, when present, code lines overwhelm the model's capacity but provide helpful information that significantly impacts the model's decision.

Dataset	AUC	F1
---------	-----	----

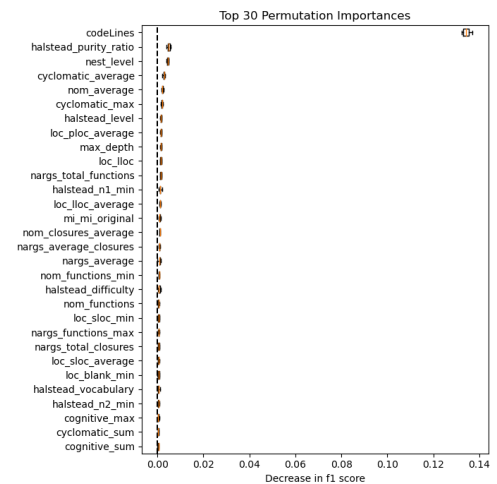
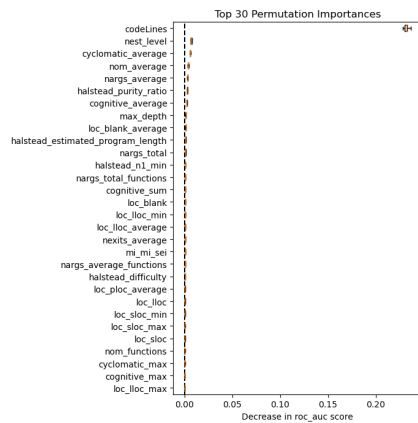
Plain



Size Group



Code Lines



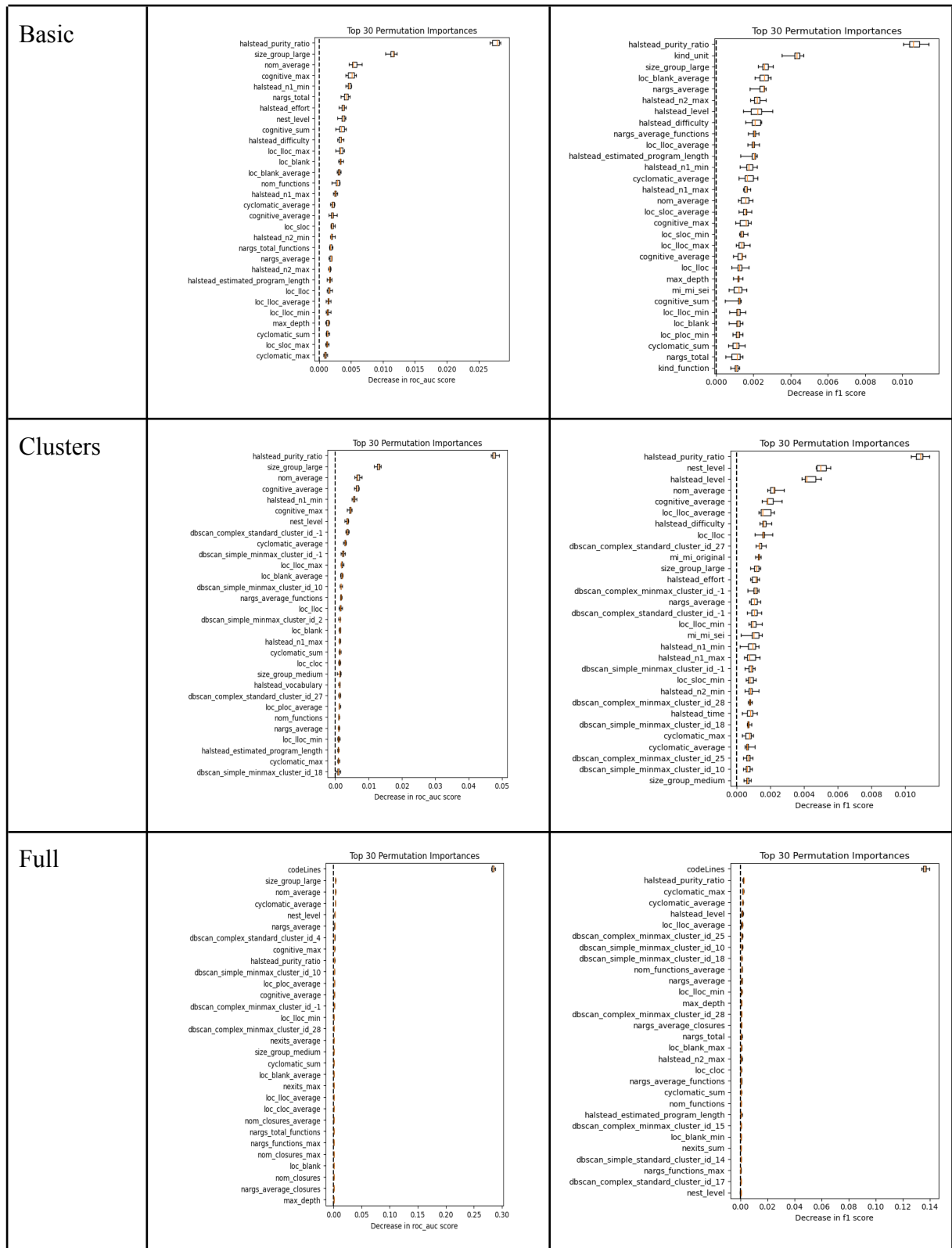


Table 18. Feature Importance for XGBoost algorithm for all F1 and AUC metric and data configurations.

4.8 Model Explainability

While constructing the Decision Tree classifier model, we plotted the tree to inspect which criteria the model used to classify the software component. For brevity, we plotted only for the Plain and Full dataset optimized for the F1 score. In **Figure 12**, we observe that `halstead_purity_ratio` is the core decisive factor, followed by `nom_average`, `nest_level`, `loc_lloc_min`, and `cyclomatic_min`. We can verify those important features by referencing the feature importance plots in the appendix. On the other hand, in **Figure 13**, the entire dataset is utilized, and in this `cascode_lines` feature is dominant, followed by `nexists_max`, `nargs_average_functions`, `dbscan` complex standard cluster-id 4. In the entire dataset, the tree follows different patterns and features based on the `code_lines` of the project, and a `code` module is present. In the left tree, DBScan complex standard cluster-id, `halstead_n2_max`, and `lox_sloc_average`, while `nexists_max` and `nargs_average_functions` play an essential role on the right side. Lastly, we observed that in the entire dataset, the model terminated early, indicating that it had more information that made the model representation simpler.

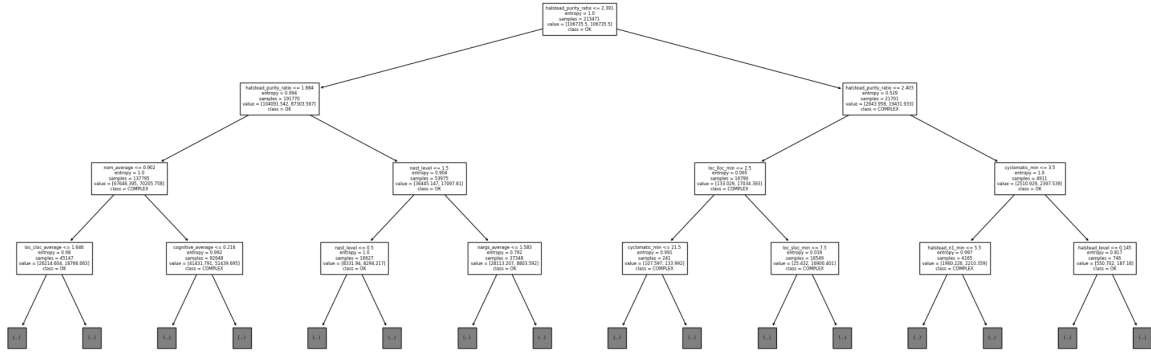


Figure 12. Representation of Decision Tree for plain dataset configuration optimized for F1 metric first three layers.

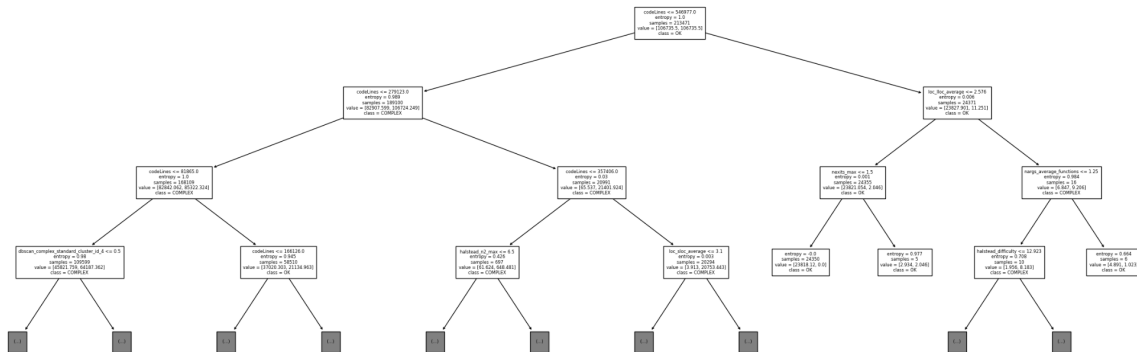


Figure 13. Representation of Decision Tree for complete dataset configuration optimized for F1 metric first three layers.

5 Discussion

In the discussion chapter, we interpret the results, compare them with existing literature, and explore the theoretical and practical implications of the research. Additionally, the chapter addresses the study's limitations and suggests avenues for future research. The discussion provides valuable insights into the performance of the proposed framework, the impact of feature engineering, and the choice of machine learning models, offering a comprehensive analysis of the study's findings and their significance.

5.1 Interpretation of Results

This section presents a detailed interpretation of our results and highlights insights and observations. The first subsection discusses which software metrics to use and for what situations. The first subsection presents the best overall performing classification algorithm. The second subsection points out if our feature engineering efforts had any impact. The third subsection highlights the importance of model explainability and gives insights into the decision tree model explanation. The fourth subsection compares the Decision Tree and Random Forest and observes why one performs better in some cases.

5.1.1 Best Performing Model

After thoroughly examining the extracted results, we concluded that the XGBoost optimized for the F1 metric gives the best overall performance. **Table 17** shows various classification evaluation metrics of the model. Although the model is imperfect, and In **Table 18**, in the confusion matrix, we can observe that it favors the classification of the main class (code modules that had a bug-fix in the past), so further work is required to create a more generalized model that is not biased.

Title	Mean	Standard Deviation (STD)
Accuracy	72.82%	00.45%
Balanced Accuracy	73.31%	00.43%
F1	77.42%	00.23%
F1 Macro	71.64%	00.55%
Precision	65.19%	00.47%

Precision Macro	78.59%	00.14%
Recall	95.32%	00.34%
Recall Macro	73.31%	00.43%
AUC	85.15%	00.35%

Table 19. Classification metrics for the XGBoost algorithm on the Full dataset configuration optimized for the F1 metric.

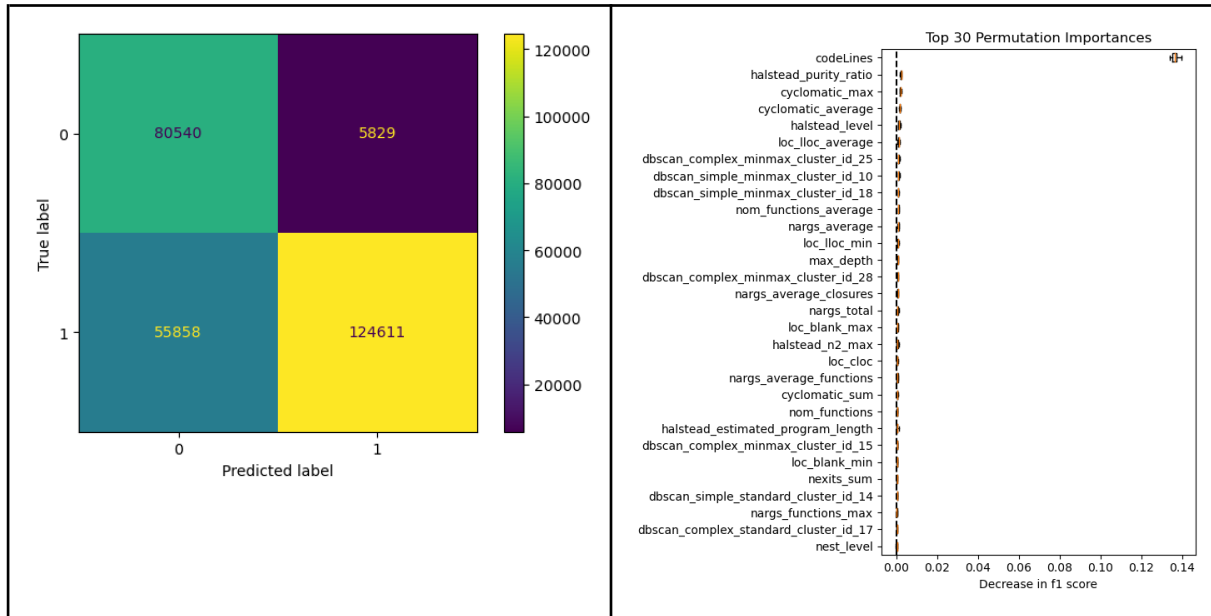


Figure 14. Confusion Matrix and Feature Importance figure for the XGBoost algorithm on the Full dataset configuration optimized for the F1 metric.

5.1.2 Feature Engineering

In our research, we performed feature engineering to improve the models' performance, so we created the following features from the initial data: `has_spaces`, `max_depth`, and `spaces_len`, which we describe in detail in **Section 3.2**. In addition, from the raw data from Github, we injected the `code_lines` and the `size_group` features into the code modules. Moreover, we used clustering to cluster data and assigned clusters for everyone. The results show that the regular features (`has_spaces`, `max_depth`, and `spaces_len`) had no impact. The clustering features slightly impacted the performance, especially the cluster IDs extracted using the DBScan algorithm. The clusters with ID=1 and ID=4 extracted using the DBScan method appear often in our results, so modules included in those groups are considered complex. The feature `size_group` was a decisive factor when referring to large code projects. That means there is evidence that large projects have different patterns on which modules are complex. Lastly, the `code_lines` feature had the most significant impact on the performance of the models, and the

performance can increase between 1% and 8% in all metrics. Further research is required to study the distribution of the tipping points where the models follow different patterns to classify the code modules.

5.1.3 Model Explainability

During our research, we were curious why the models behave so, and model explainability could solve that issue. We plotted the Decision Tree structure for various configurations, and those plots gave us valuable information. One of those was which features were necessary, and we confirmed that by comparing them with the feature importance plots. Moreover, those plots are simple to understand by both developers and managers. Lastly, we can use those models in tools like IDEs to provide feedback to the developers on their code and act as a guideline to improve quality.

5.1.4 Decision Tree vs. Random Forest

The results showed that the Random Forest sometimes falls behind the Decision Tree classifier. We attribute this to the fact that Random Forest takes a subset of features each time and constructs a tree based on those limited features a total number of estimators times. So, in most cases, the Random Forest algorithm does not have the feature `code_lines` to use in constructing the tree. In comparison, the Bagging algorithm has all the features and constructs a different tree each time to cover uncertain areas. However, if we compare Random Forest and Decision trees in the datasets that do not contain the `code_lines` feature, the Random Forest classifiers consistently outperform the Decision Tree ones.

5.2 Comparison with Literature

Most of the works in the literature use existing datasets for popular languages like C and Java, whereas we created our dataset using the proposed methodology [23], [25]. Regarding the algorithms, we used a vast collection mostly met in other works, except for CNN and other deep neural network techniques, that show promising results [25]. In the literature, some works showed $F1 = 75\%$ while others $AUC = 88\%$. Comparing those with our results, we achieved similar performance $F1 = 77\%$ and $AUC = 85\%$. Nevertheless, we must remember that we constructed our dataset, and comparing them is not right, but it can act as a base comparison that we are going in the right direction. Another difference we pointed out was that we could not make SVM work in our dataset. At the same time, many papers in the literature achieved that [25], [26].

5.3 Theoretical and Practical Implications

Regarding the theoretical implications, we concluded that one can use a synthetic dataset of software defects to train models that detect them. Further work is required to verify that this has value to stakeholders like developers and managers. Relevant works were limited to existing datasets that might need an update to meet current software trends. On the other hand, regarding the practical implications, our work can be developed further by companies and offered as a paid solution. Moreover, we showcased which software metrics impact the quality of software components. Lastly, the explainability of why software components are complex to maintain is an issue, so by using models like Decision Trees, we can justify the reasons.

5.4 Limitations

Due to limited time and resource constraints, we limited our work to finish on time. One of those limitations is the population size; on the GitHub website, over 30K projects are written in Rust in the SEART database. To train the models with the current hardware and time, we picked a smaller sample of 110 projects [27]. We made a custom sampling strategy by separating the projects into three categories and picking the most active ones for each category, aiming to represent as much of the population as possible [14]. However, during our literature review, our choice of projects for the sample might hide open source and popularity bias that might affect the results if run on closed-source scenarios or unpopular projects. In addition, during the parameter optimization step in the methodology, we limited the iterations of RandomizedSearchCV to 500. Lastly, another limitation we assumed was that the appearance of bug-fixing commits determines code quality in a code module [13].

6 Future Work

In this chapter, we propose further work for the following research projects. Firstly, we discuss advanced feature engineering methods and propose further research into using Language Model Models (LLMs) for marking bug-fixed files. Secondly, we propose using a larger sample size and more complex algorithms to determine if we can achieve better results. Lastly, more applied research is required, and future work should survey stakeholders to see

whether it brings value to their work. These future research directions aim to build upon the current study's findings and improve the performance of the models.

6.1 Advanced Feature Engineering

In our experiments, we investigated if feature engineering could improve the performance of the results. There is evidence that cluster features can improve the models' performances of the models. However, the injection of the feature code_lines in the code modules showed impressive improvement, overshadowing the other engineered features. Due to picking the more straightforward feature engineering methods, there are more advanced ones in the literature, like distance from clusters. We propose future work to research the bibliography, include more advanced feature methodologies, and compare if they can further improve the performance of the models [15], [16].

6.2 Use LLMs for Marking Bug-Fixed Files

In the work of Zafar et al., they propose a methodology for detecting if a git commit includes a bug fix. Their work finetuned a BERT LLM to classify if the commit included a bug fix. It performs better than ad-hoc regex or NLP analysis solutions because it can distinguish between bug fixes on actual code and test code and changes on comments only in cases where context can change the decision (e.g., refactoring changes) [24]. We propose a future work to include the mentioned methodology for marking the bug-fixed files, then recalculate the results and compare them with the current work to identify if it improves the performance of the models.

6.3 Larger Sample

During our experiments, we tried a smaller sample and got poor results, so we are curious about what will happen if we scale our sample size to include more repositories. In the literature, there are references to the fact that the size of the dataset plays a vital role in the performance of the trained models. We advise future work to include a larger sample and study its effects on various classification algorithms[16].

6.4 Survey Stakeholders

Our research is primarily theoretical, and the evaluation is based on our definition of complex code. We suggest more practical research to survey field experts and ask them to use a

collection (Recall, Precision, F1) models and evaluate the results of the models. Then, use a fuzzy algorithm to compare the experts' opinions and decide if the experts agree that the models produce a meaningful result.

6.5 Complex Algorithms

Our work concluded that the Perceptron model performs poorly because of its limited capacity and requires further parameter optimization. However, more complex Neural Network architectures like CNN, Transformers, and others exist. We propose future research to focus only on using more complex Neural Network architectures and explore how they perform on the defined problem [15].

6.6 Explain Models in Depth

Our efforts to explain why the models behave the way they do barely scratch the surface, so we suggest future research to study the characteristics of the DBScan clusters mentioned in **Subsection 5.1.3**. Moreover, the distribution of the features for the different size_group classes, but most importantly, to identify where various models have a tipping point for the feature code_lines, collect them, and study the distribution of the features for each range [15], [16].

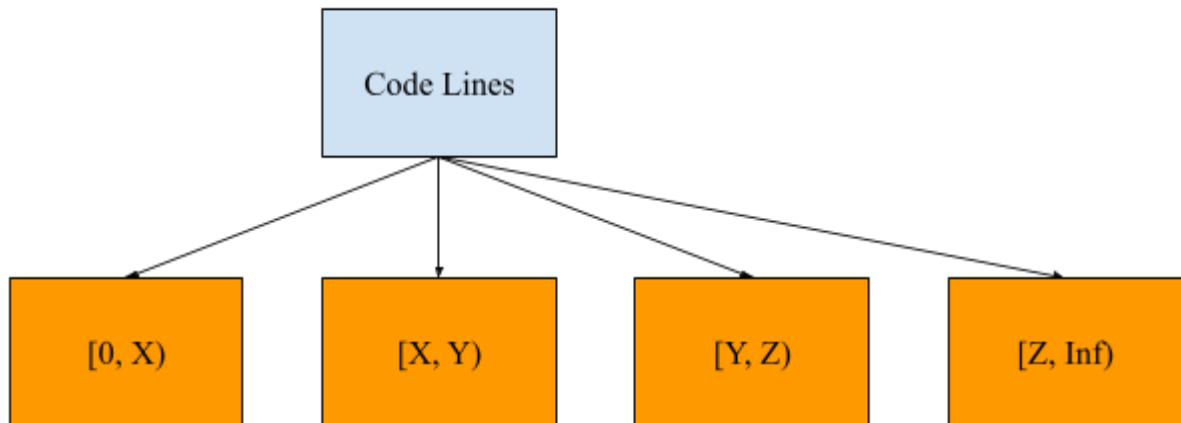


Figure 15. We suggest finding where the models (e.g., Decision Tree) decide based on the feature code_lines and studying the other features' distribution for each range.

6.7 Finetuning of Models

Because we wanted to test as many models as possible, we used RandomizedSearchCV to stay within time constraints and not exhaustively test the whole parameter set space. There is

a possibility of missing important parameters for each classification algorithm. We propose that future researchers focus on the best-performing algorithm, XGBoost, and optimize it further. In addition, we suggest taking time to understand and explain the importance of the parameters' effect on the model's training and how sensitive a model is to a slight change in the parameters. Lastly, we advise further research on why the Perceptron classifier performs poorly and investigate our hypothesis due to the limited model capacity [15].

7 Conclusion

In this thesis, we proposed a framework for automatically marking complex Rust code using software metrics. We aimed to evaluate the effectiveness of our proposed methodology in detecting software defects and to explore the impact of different software metrics and machine learning algorithms on the results. Our results showed promising performance in detecting software defects, with F1 scores of 77% and AUC scores of 85%. We also identified the software metrics that have the most significant impact on the quality of software components and explored the effectiveness of different machine learning algorithms in detecting software defects. However, our study also had limitations, such as the small sample size and the limited number of algorithms used. Future research could explore more advanced feature engineering techniques, larger sample sizes, and more complex algorithms to improve the accuracy of the results. Overall, our proposed framework provides a valuable tool for stakeholders in the software development industry to evaluate the quality of their code base and optimize their development process. By automating the process of detecting software defects, we can save resources and improve the efficiency of the development process. In conclusion, returning to our initial Research Questions:

- **RQ:** *Does our proposed automatic software quality evaluation framework produce acceptable results?*

The study results showed promising performance in detecting software defects, with F1 scores of 77% and AUC scores of 85%. Therefore, we can consider the proposed framework to produce an acceptable result

- **RQ1:** *Which code metrics should we use?*

The study identified several software metrics that significantly impact the quality of software components, but the most noticeable are the Halstead suite metrics, code_lines, and cyclomatic complexity. Lastly, DBScan standard scaled cluster ID=1, and ID=4 caught our interest because they appeared in many feature importance graphs.

- **RQ2:** *What subsets of data should we use?*

The study used six different data configurations, each with a different combination of features, to evaluate the impact of different data subsets on model performance. The results showed that different data subsets can significantly impact model performance. We advise using the Basic dataset with code_lines included as it provides the best performance, but keep in mind to increase the capacity of the models so they can comprehend and model the problem space when the code_lines feature is present.

- **RQ3:** *Which machine learning algorithms produce acceptable results?*

The study evaluated several machine learning algorithms, including Decision Tree, Random Forest, Perceptron, and XGBoost. The results showed that different algorithms have varying levels of effectiveness in detecting software defects, with XGBoost performing the best overall.

- **RQ4:** *Which classification evaluation metrics should we use?*

The study used several evaluation metrics, including accuracy, precision, recall, F1 score, and AUC, to evaluate the performance of the models. The results showed that different metrics can provide different insights into model performance, and the most appropriate metrics depend on the specific problem and context.

- **RQ5:** *Could we use feature engineering techniques to improve the result?*

Our results showed that the returns of performing feature engineering are diminishing, suggesting that further research is required. However, we found that injecting the code_lines of the project into the records or transforming them into size_group groups brought significant results because the models followed different patterns for each category. Lastly, the clustering efforts did not bring the expected results we had but triggered our interest in studying two cluster groups that might contain meaningful characteristics of complex code.

Bibliography

- [1] A. Haider, Q. Zahid, and N. Wasim, “The myth of the technical manager,” in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, Beijing, China: IEEE, 2009, pp. 255–258. doi: 10.1109/ICCSIT.2009.5234717.
- [2] J. Tian, “Quality-evaluation models and measurements,” *IEEE Softw.*, vol. 21, no. 3, Art. no. 3, May 2004, doi: 10.1109/MS.2004.1293078.
- [3] J. Greenfield and K. Short, “Assembling Applications with Patterns, Models, Frameworks and Tools”.
- [4] N. Gorla and S.-C. Lin, “Determinants of software quality: A survey of information systems project managers,” *Inf. Softw. Technol.*, vol. 52, no. 6, pp. 602–610, Jun. 2010, doi: 10.1016/j.infsof.2009.11.012.
- [5] M. Tufano *et al.*, “When and Why Your Code Starts to Smell Bad,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, Italy: IEEE, May 2015, pp. 403–414. doi: 10.1109/ICSE.2015.59.
- [6] N. D. Matsakis and F. S. Klock, “The rust language,” in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, Portland Oregon USA: ACM, Oct. 2014, pp. 103–104. doi: 10.1145/2663171.2663188.
- [7] “Stack Overflow Developer Survey 2022,” *Stack Overflow*. [Online]. Available: <https://survey.stackoverflow.co/2022/>
- [8] L. Ardito, L. Barbato, R. Coppola, and M. Valsesia, “Evaluation of Rust code verbosity, understandability and complexity,” *PeerJ Comput. Sci.*, vol. 7, p. e406, Feb. 2021, doi: 10.7717/peerj-cs.406.
- [9] Q. Taylor, C. G. Carrier, and C. D. Knutson, “Applications of data mining in software engineering,” *Int. J. Data Anal. Tech. Strateg.*, vol. 2, no. 3, Art. no. 3, 2010, doi: 10.1504/IJDATS.2010.034058.
- [10] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, “Analyzing software measurement data with clustering techniques,” *IEEE Intell. Syst.*, vol. 19, no. 2, Art. no. 2, Mar. 2004, doi: 10.1109/MIS.2004.1274907.
- [11] P. Antonellis *et al.*, “A Data Mining Methodology for Evaluating Maintainability according to ISO/IEC-9126 Software Engineering-Product Quality Standard”.
- [12] S. Arshad and C. Tjortjis, “Clustering Software Metric Values Extracted from C# Code for Maintainability Assessment,” in *Proceedings of the 9th Hellenic Conference on Artificial Intelligence*, Thessaloniki Greece: ACM, May 2016, pp. 1–4. doi: 10.1145/2903220.2903252.
- [13] T. M. Khoshgoftaar, “Analogy-Based Practical Classification Rules for Software Quality Estimation”.
- [14] Z. O’Leary, “The Essential Guide to Doing Your Research Project”.
- [15] G. Rebala, A. Ravi, and S. Churiwala, *An Introduction to Machine Learning*. Cham: Springer International Publishing, 2019. doi: 10.1007/978-3-030-15729-6.
- [16] C. C. Aggarwal, *Data Mining: The Textbook*. Cham: Springer International Publishing, 2015. doi: 10.1007/978-3-319-14142-8.
- [17] H. Al-Kilidar, K. Cox, and B. Kitchenham, “The use and usefulness of the ISO/IEC 9126 quality standard,” in *2005 International Symposium on Empirical Software Engineering, 2005.*, Queensland, Australia: IEEE, 2005, pp. 122–128. doi: 10.1109/ISESE.2005.1541821.
- [18] L. Ardito *et al.*, “rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes,” *SoftwareX*, vol. 12, p. 100635, Jul. 2020,

doi: 10.1016/j.softx.2020.100635.

- [19] M. Grandini, E. Bagli, and G. Visani, “Metrics for Multi-Class Classification: an Overview,” arXiv, Aug. 13, 2020. Accessed: Dec. 21, 2023. [Online]. Available: <http://arxiv.org/abs/2008.05756>
- [20] D. J. Hand, “Measuring classifier performance: a coherent alternative to the area under the ROC curve,” *Mach. Learn.*, vol. 77, no. 1, pp. 103–123, 2009.
- [21] P. Gyimesi, G. Gyimesi, Z. Tóth, and R. Ferenc, “Characterization of Source Code Defects by Data Mining Conducted on GitHub,” in *Computational Science and Its Applications -- ICCSA 2015*, vol. 9159, O. Gervasi, B. Murgante, S. Misra, M. L. Gavrilova, A. M. A. C. Rocha, C. Torre, D. Tanar, and B. O. Apduhan, Eds., in Lecture Notes in Computer Science, vol. 9159, Cham: Springer International Publishing, 2015, pp. 47–62. doi: 10.1007/978-3-319-21413-9_4.
- [22] D. Papas and C. Tjortjis, “Combining Clustering and Classification for Software Quality Evaluation,” in *Artificial Intelligence: Methods and Applications*, vol. 8445, A. Likas, K. Blekas, and D. Kalles, Eds., in Lecture Notes in Computer Science, vol. 8445, Cham: Springer International Publishing, 2014, pp. 273–286. doi: 10.1007/978-3-319-07064-3_22.
- [23] C. Casalnuovo, Y. Suchak, B. Ray, and C. Rubio-González, “GitProc: a tool for processing and classifying GitHub commits,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Santa Barbara CA USA: ACM, Jul. 2017, pp. 396–399. doi: 10.1145/3092703.3098230.
- [24] S. Zafar, M. Z. Malik, and G. S. Walia, “Towards Standardizing and Improving Classification of Bug-Fix Commits,” in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Porto de Galinhas, Recife, Brazil: IEEE, Sep. 2019, pp. 1–6. doi: 10.1109/ESEM.2019.8870174.
- [25] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, “A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools,” *Eng. Appl. Artif. Intell.*, vol. 111, p. 104773, May 2022, doi: 10.1016/j.engappai.2022.104773.
- [26] S. Stradowski and L. Madeyski, “Machine learning in software defect prediction: A business-driven systematic mapping study,” *Inf. Softw. Technol.*, vol. 155, p. 107128, Mar. 2023, doi: 10.1016/j.infsof.2022.107128.
- [27] O. Dabic, E. Aghajani, and G. Bavota, “Sampling Projects in GitHub for MSR Studies,” no. arXiv:2103.04682. arXiv, Mar. 08, 2021. Accessed: Jul. 06, 2023. [Online]. Available: <http://arxiv.org/abs/2103.04682>
- [28] T. pandas development team, “pandas-dev/pandas: Pandas.” Zenodo, Feb. 2020. doi: 10.5281/zenodo.3509134.
- [29] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

Appendix

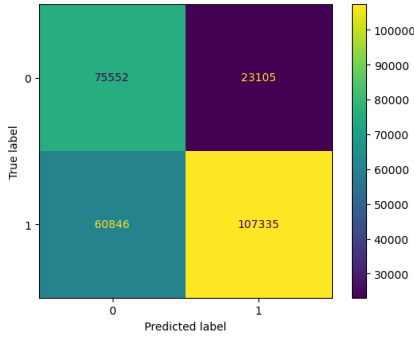
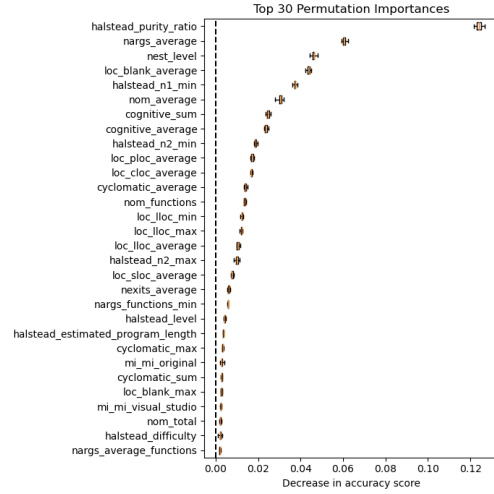
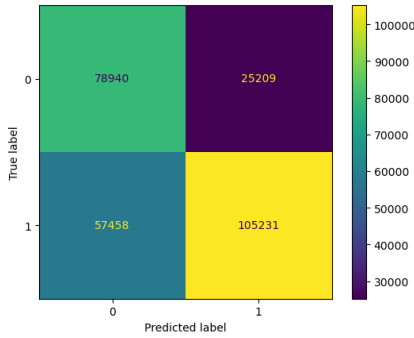
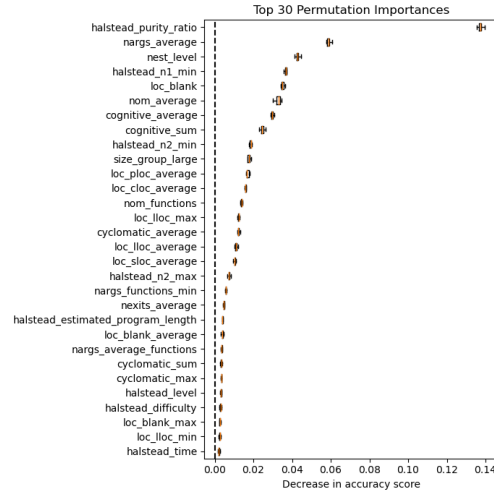
Source Code

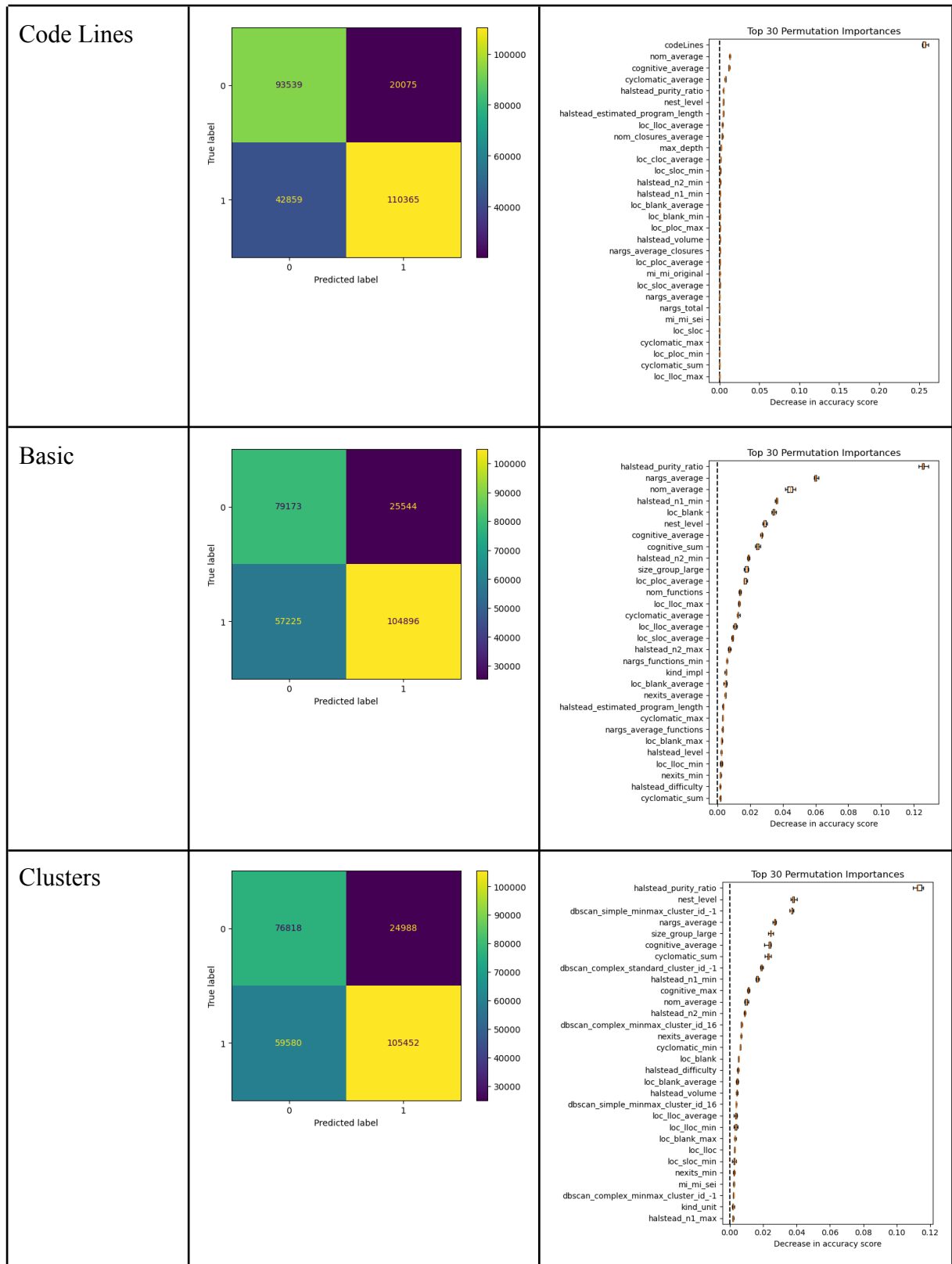
<https://github.com/karataki/rust-code-complexity-ml>

Dataset

<https://www.kaggle.com/datasets/karataki/rust-code-software-metrics-vs-bug-fix-past>

Accuracy Score Results

Decision Tree	Confusion Matrix	Feature Importance									
Plain	 <p>Confusion Matrix for Plain Decision Tree model:</p> <table border="1"> <thead> <tr> <th>True label \ Predicted label</th> <th>0</th> <th>1</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>75552</td> <td>23105</td> </tr> <tr> <th>1</th> <td>60846</td> <td>107335</td> </tr> </tbody> </table>	True label \ Predicted label	0	1	0	75552	23105	1	60846	107335	 <p>Top 30 Permutation Importances for Plain Decision Tree model:</p> <ul style="list-style-type: none"> halstead_purity_ratio nargs_average nest_level loc_blank_average halstead_n1_min nom_average cognitive_sum cognitive_average halstead_n2_min loc_ploc_average loc_cloc_average cyclomatic_average nom_functions loc_lloc_min loc_lloc_max loc_lloc_average halstead_n2_max loc_slloc_average nexits_average nargs_functions_min halstead_level halstead_estimated_program_length cyclomatic_max mi_mi_original cyclomatic_sum loc_blank_max mi_mi_visual_studio nom_total halstead_difficulty nargs_average_functions
True label \ Predicted label	0	1									
0	75552	23105									
1	60846	107335									
Size Group	 <p>Confusion Matrix for Size Group Decision Tree model:</p> <table border="1"> <thead> <tr> <th>True label \ Predicted label</th> <th>0</th> <th>1</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>78940</td> <td>25209</td> </tr> <tr> <th>1</th> <td>57458</td> <td>105231</td> </tr> </tbody> </table>	True label \ Predicted label	0	1	0	78940	25209	1	57458	105231	 <p>Top 30 Permutation Importances for Size Group Decision Tree model:</p> <ul style="list-style-type: none"> halstead_purity_ratio nargs_average nest_level halstead_n1_min loc_blank nom_average cognitive_average cognitive_sum halstead_n2_min size_group_large loc_ploc_average loc_cloc_average nom_functions loc_lloc_max cyclomatic_average loc_lloc_average loc_slloc_average halstead_n2_max nargs_functions_min nexits_average halstead_estimated_program_length loc_blank_average nargs_average_functions cyclomatic_sum cyclomatic_max halstead_level halstead_difficulty loc_blank_max loc_lloc_min halstead_time
True label \ Predicted label	0	1									
0	78940	25209									
1	57458	105231									



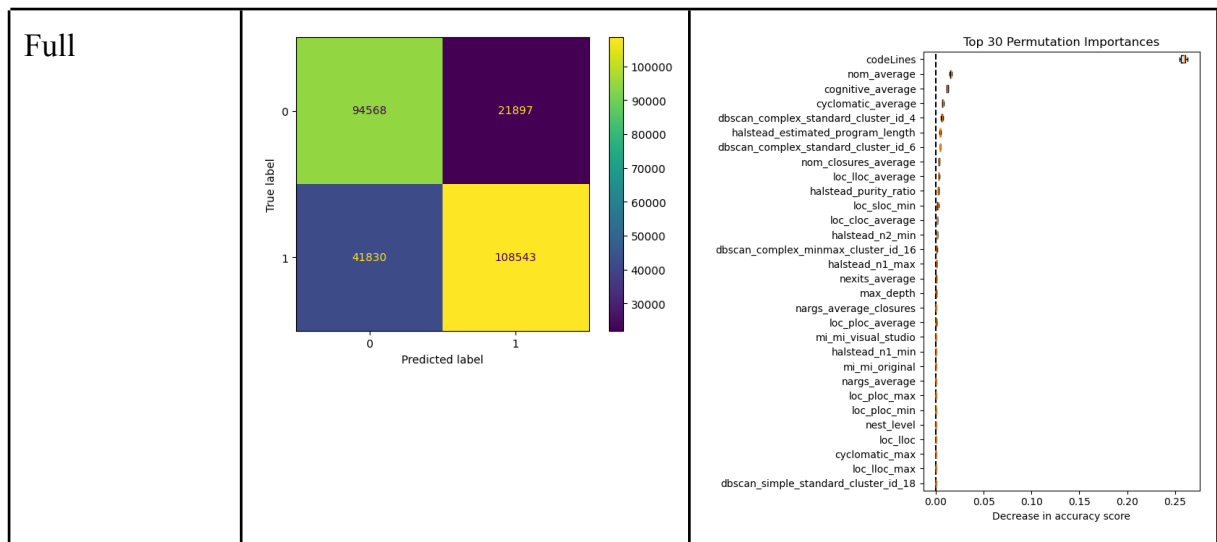
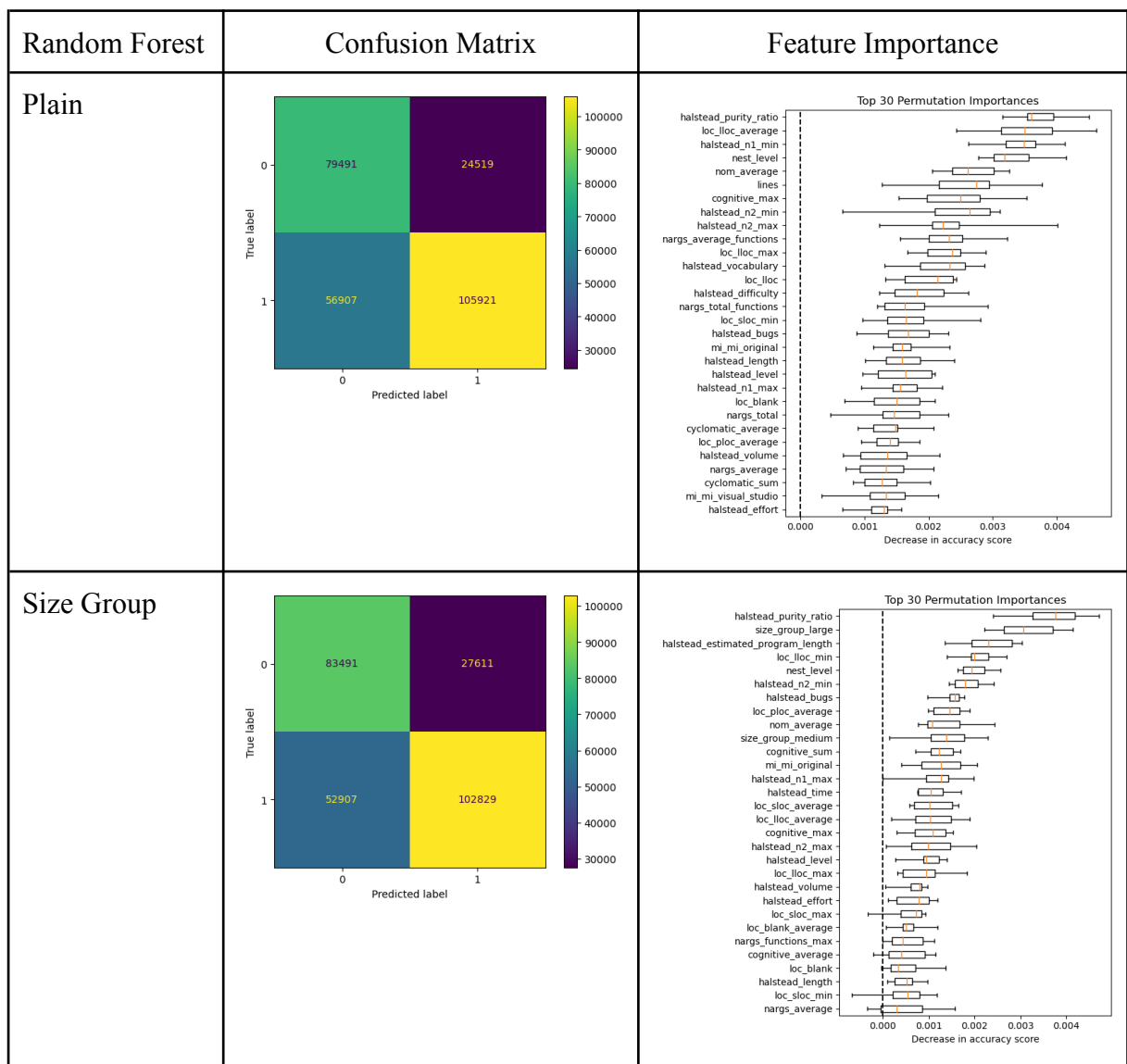
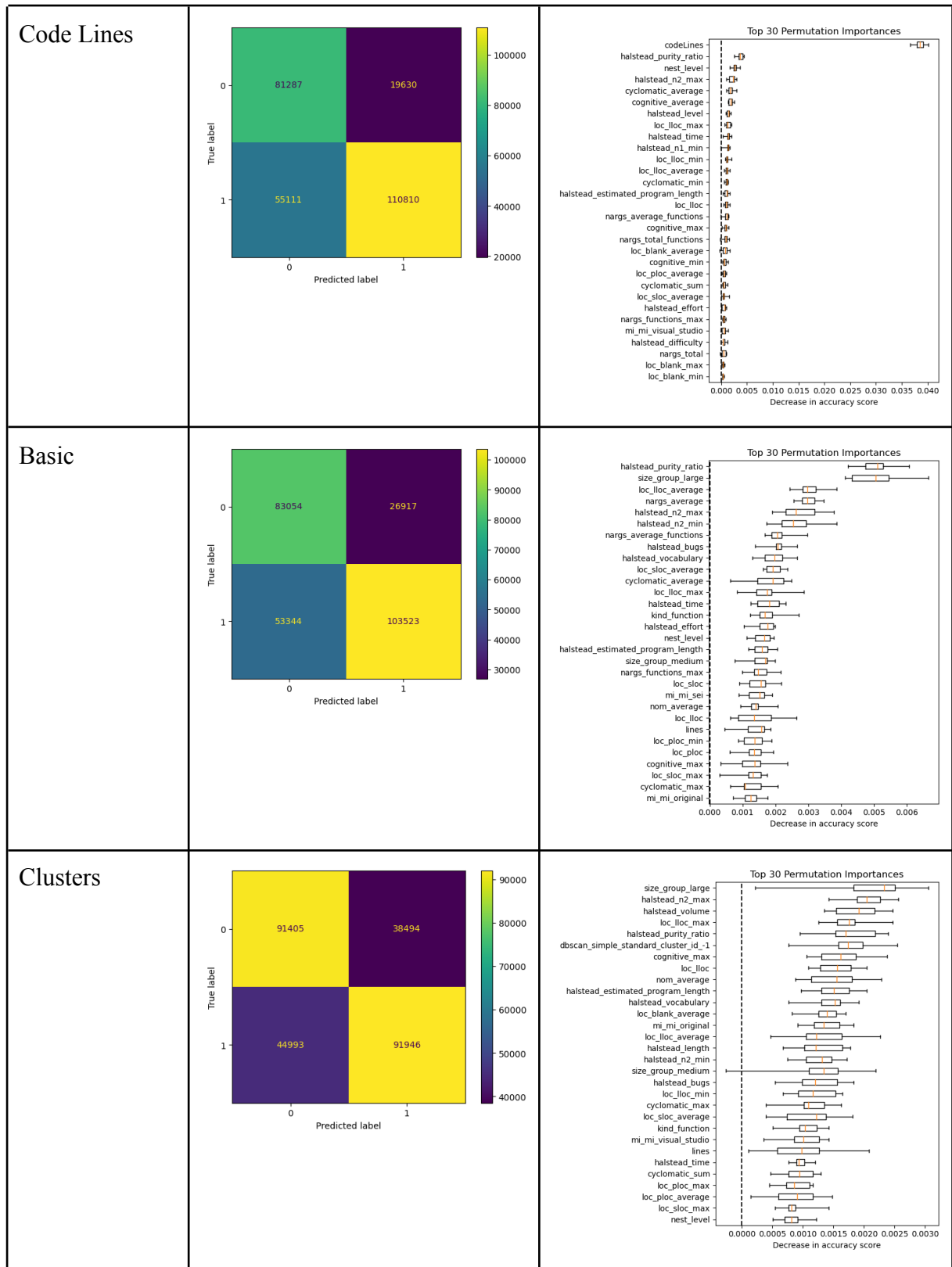


Figure. Confusion Matrix and Feature Importance for Decision Tree algorithm for all data configurations for the accuracy metric.





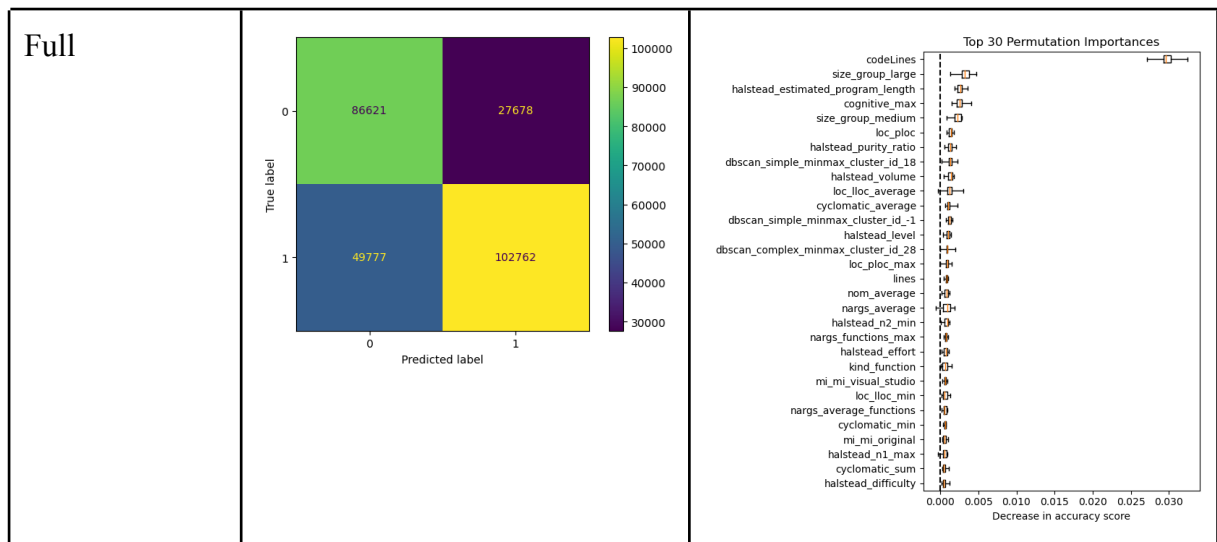
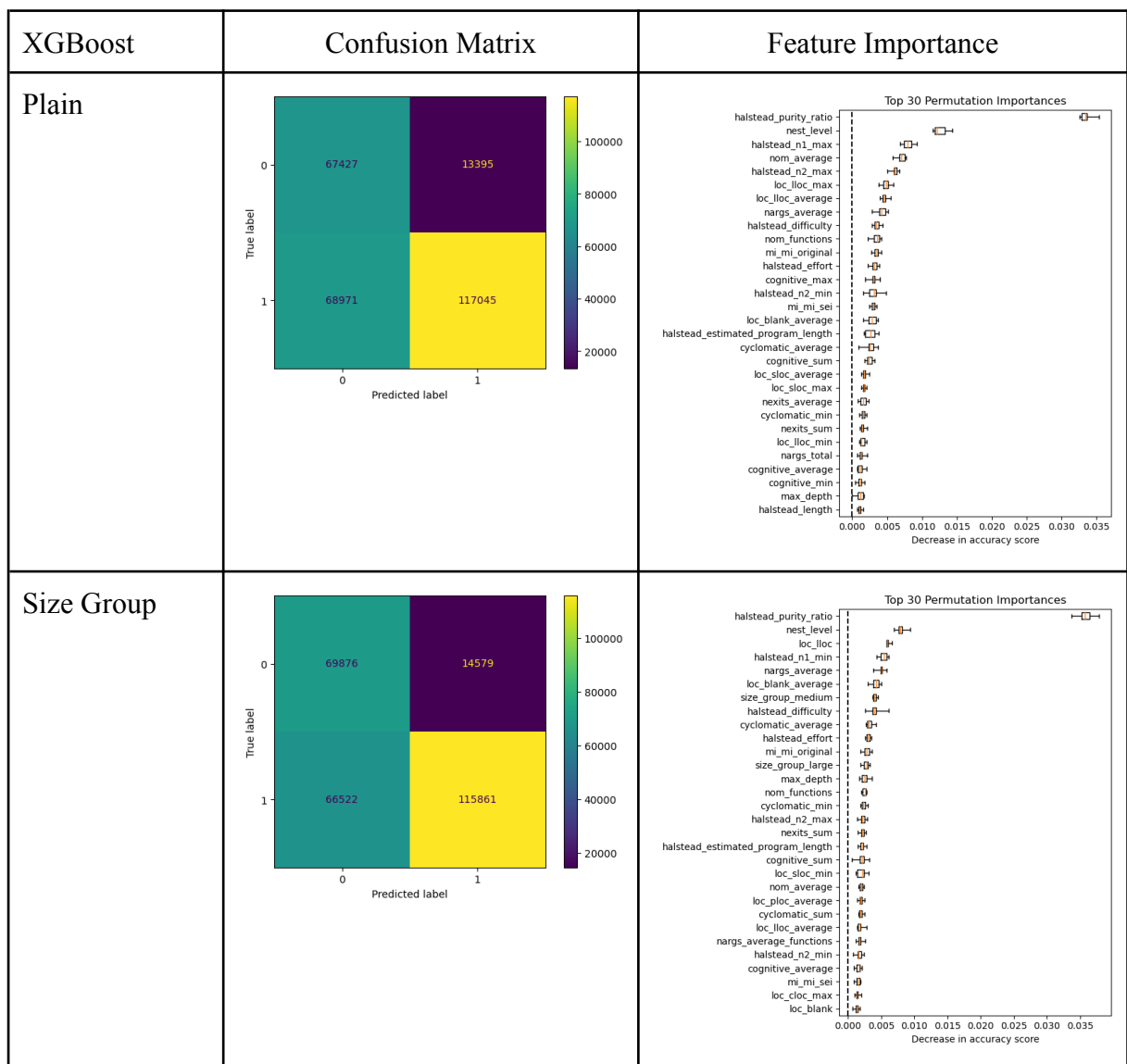
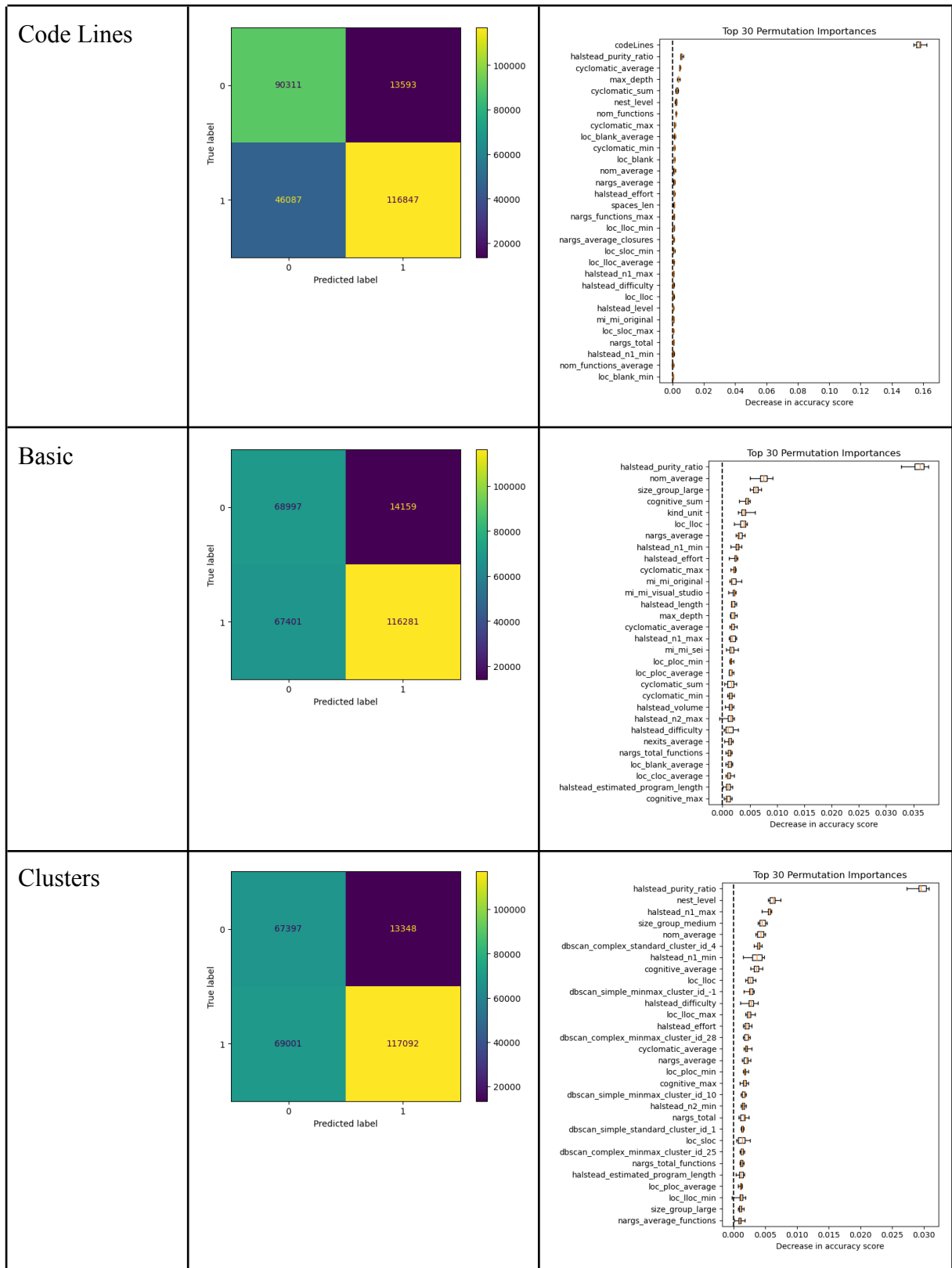


Figure. Confusion Matrix and Feature Importance for Random Forest algorithm for all data configurations for the accuracy metric.





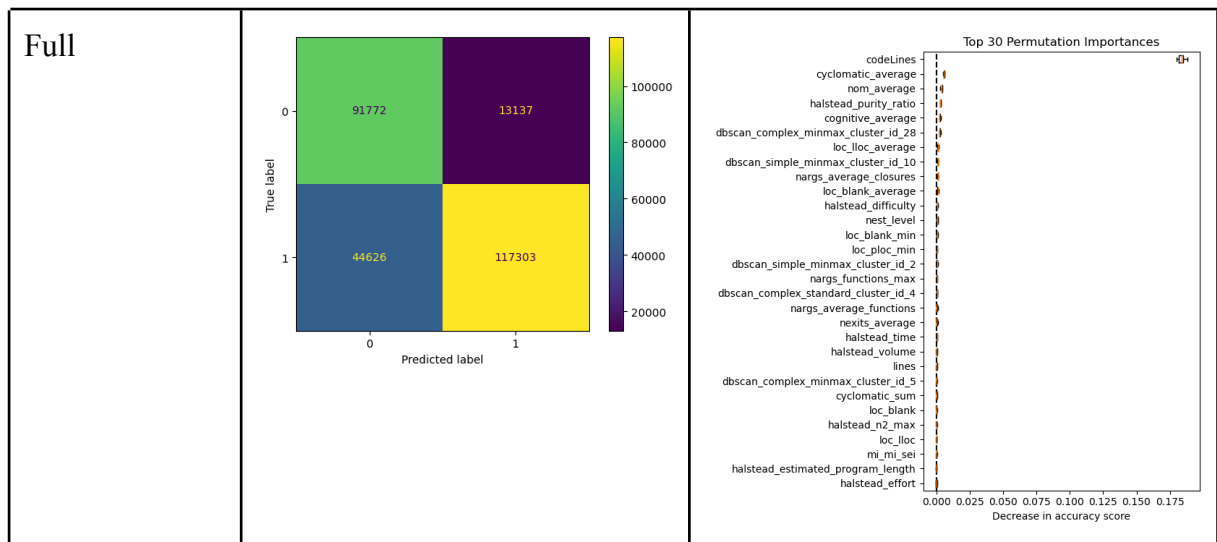
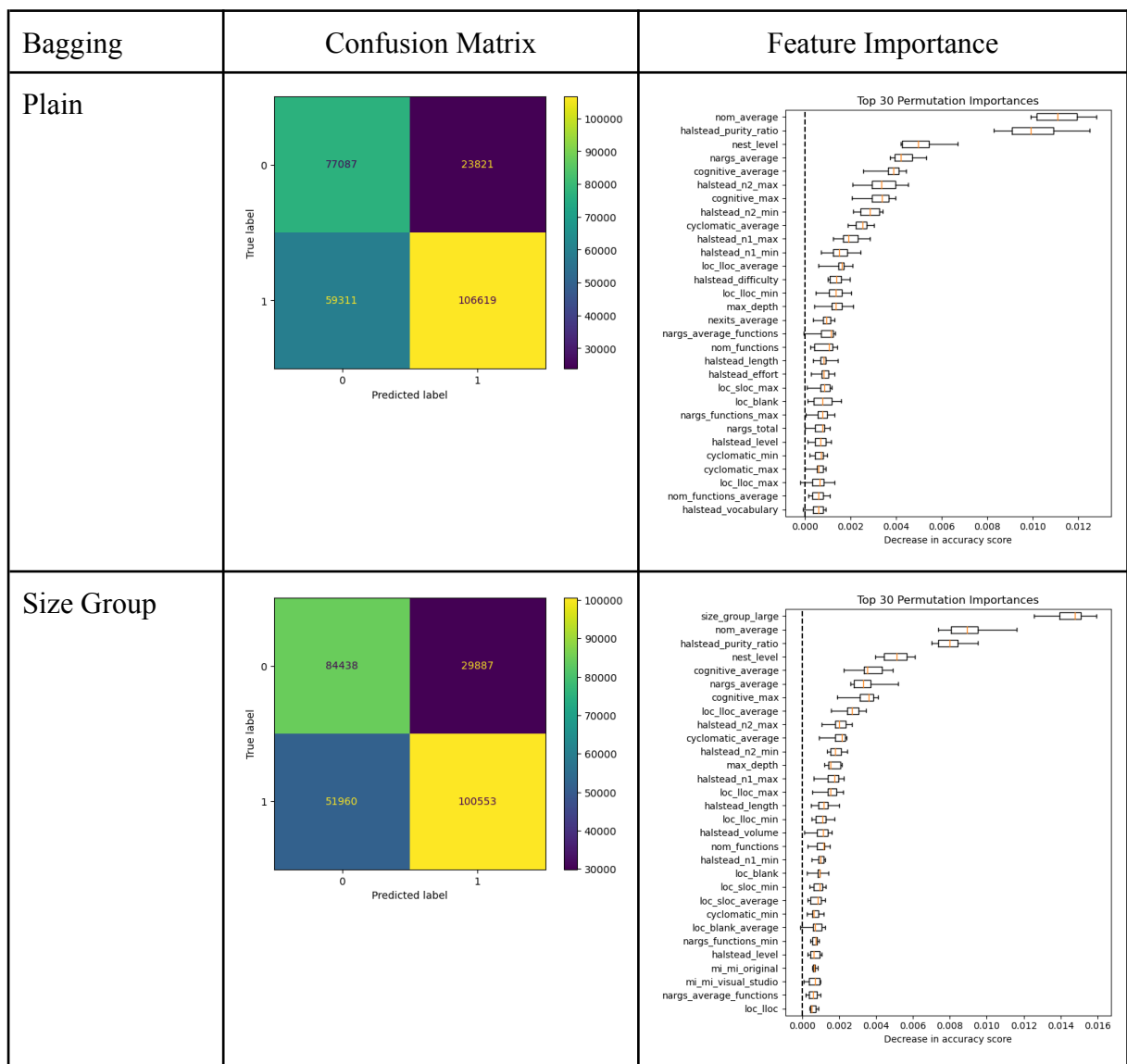
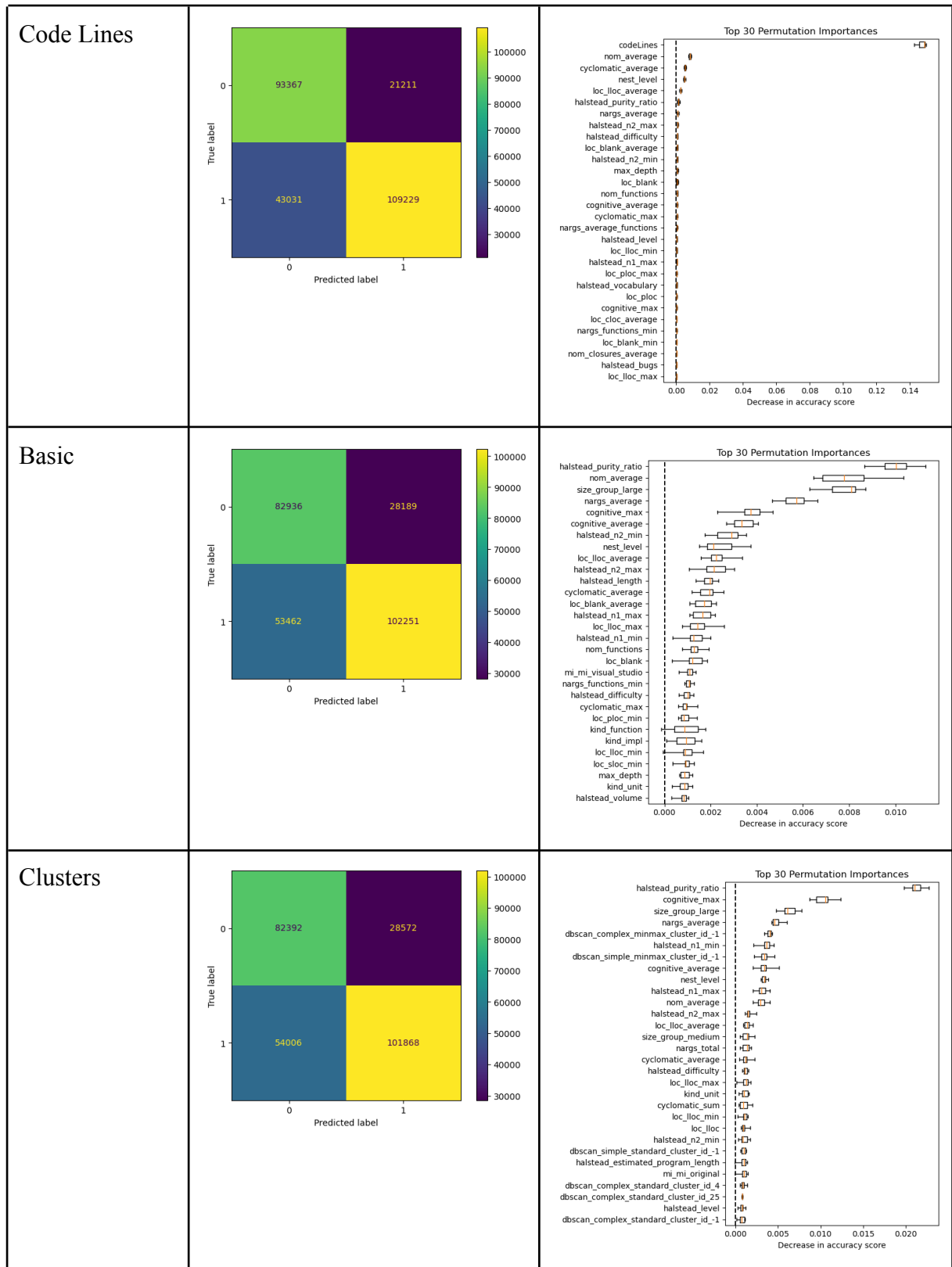


Figure. Confusion Matrix and Feature Importance for XGBoost algorithm for all data configurations for the accuracy metric.





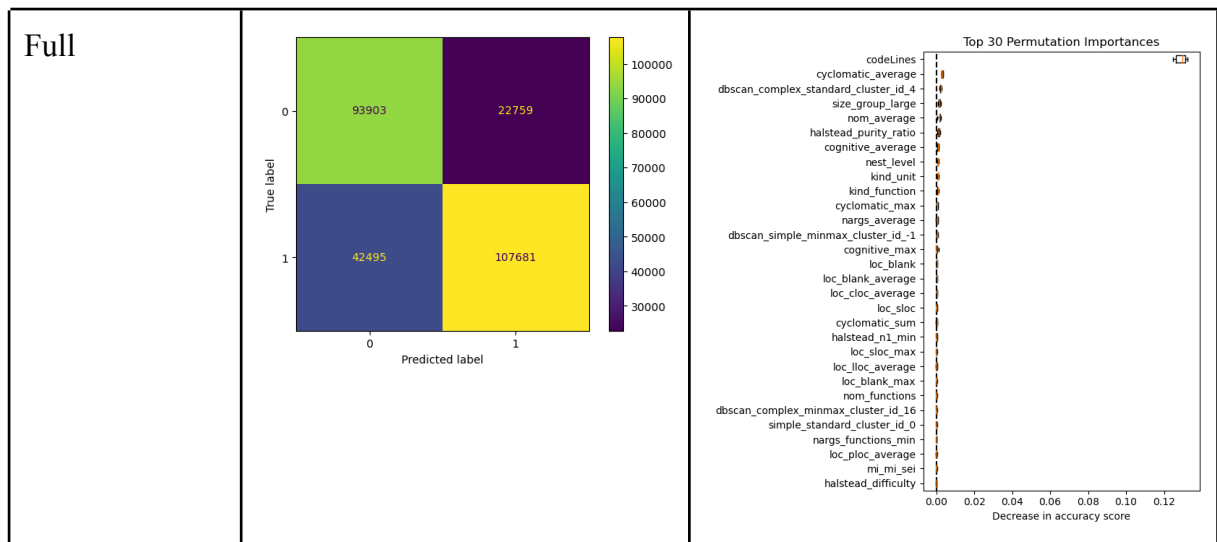
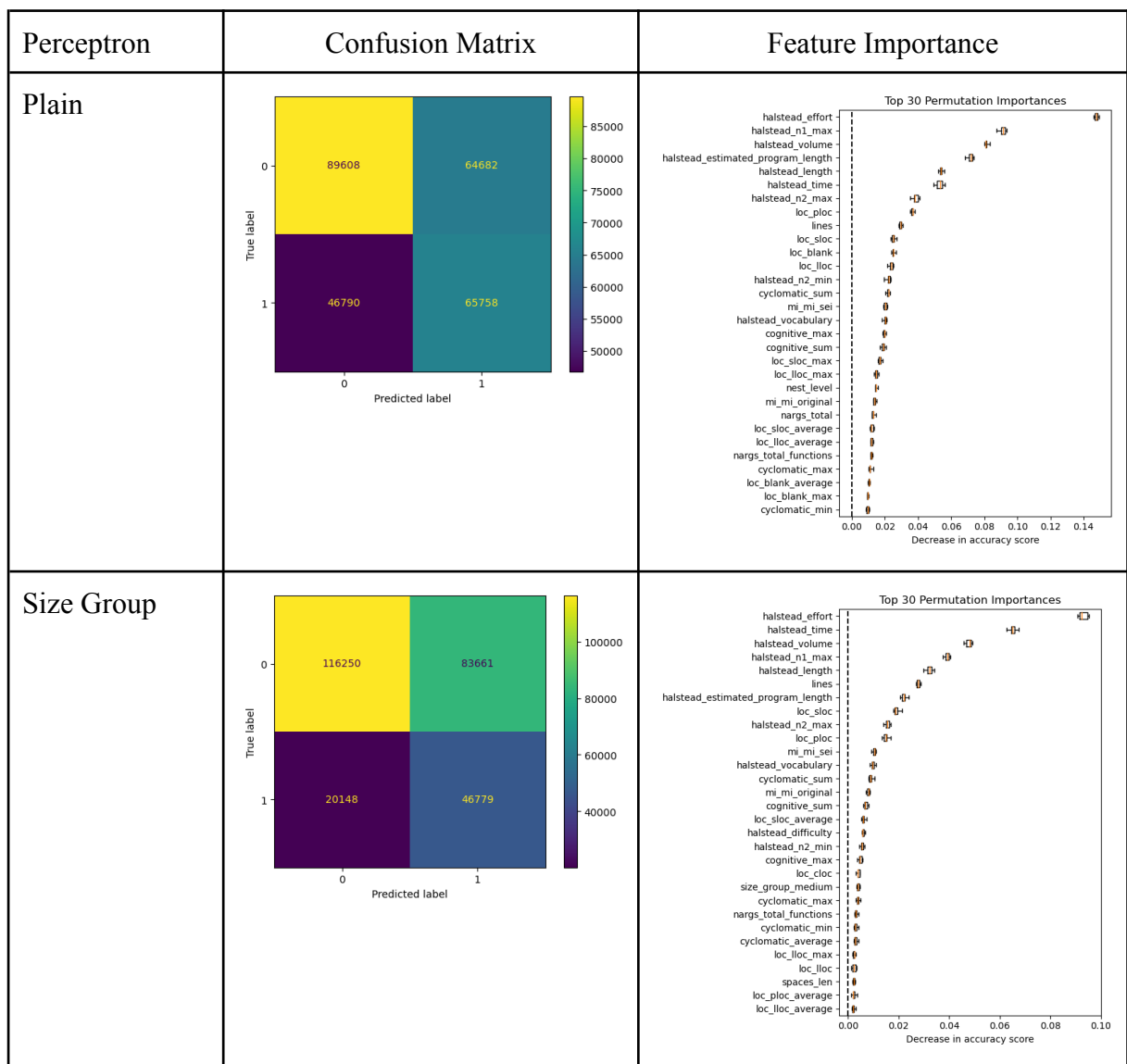
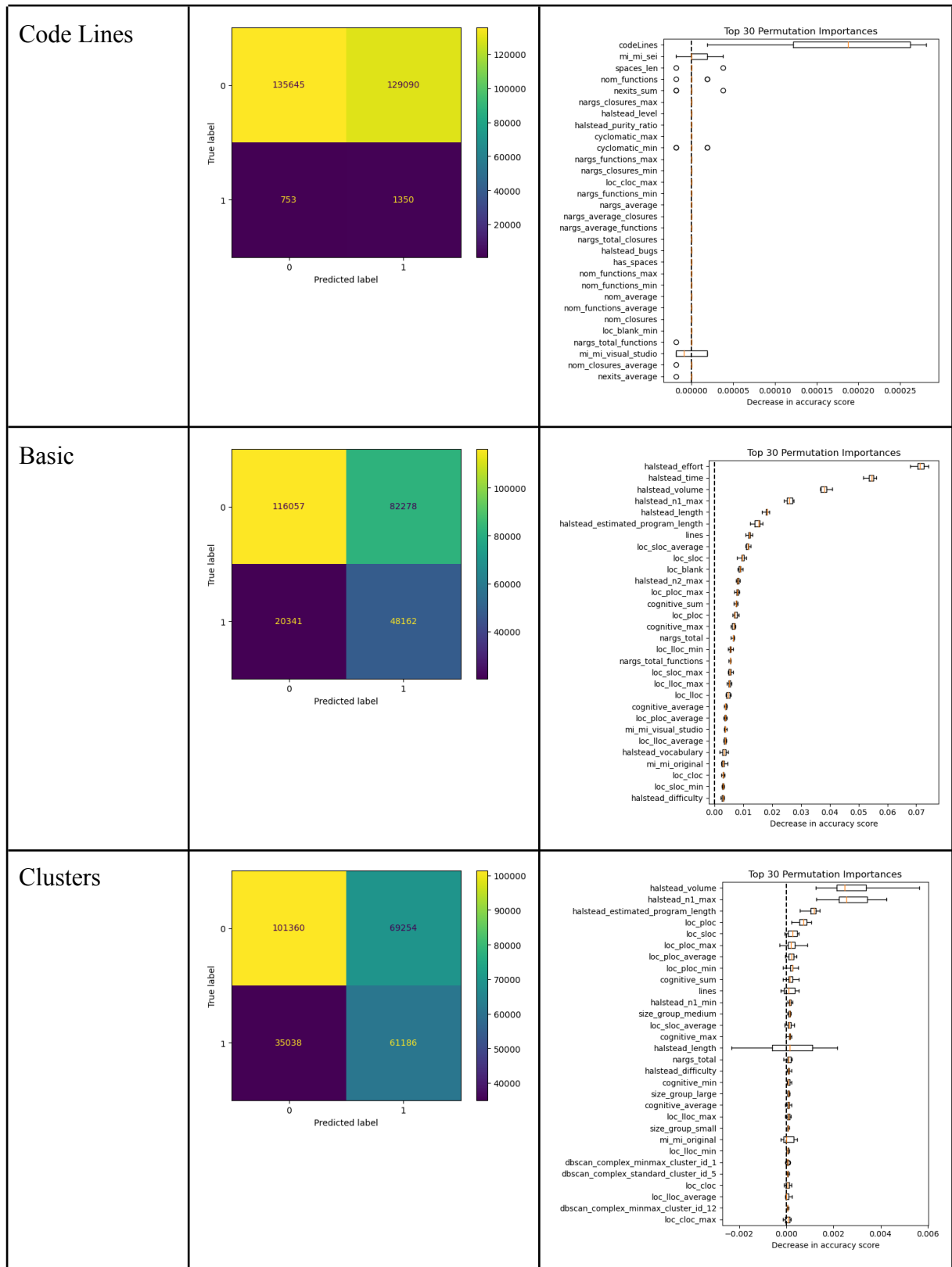


Figure. Confusion Matrix and Feature Importance for Bagging algorithm for all data configurations for the accuracy metric.





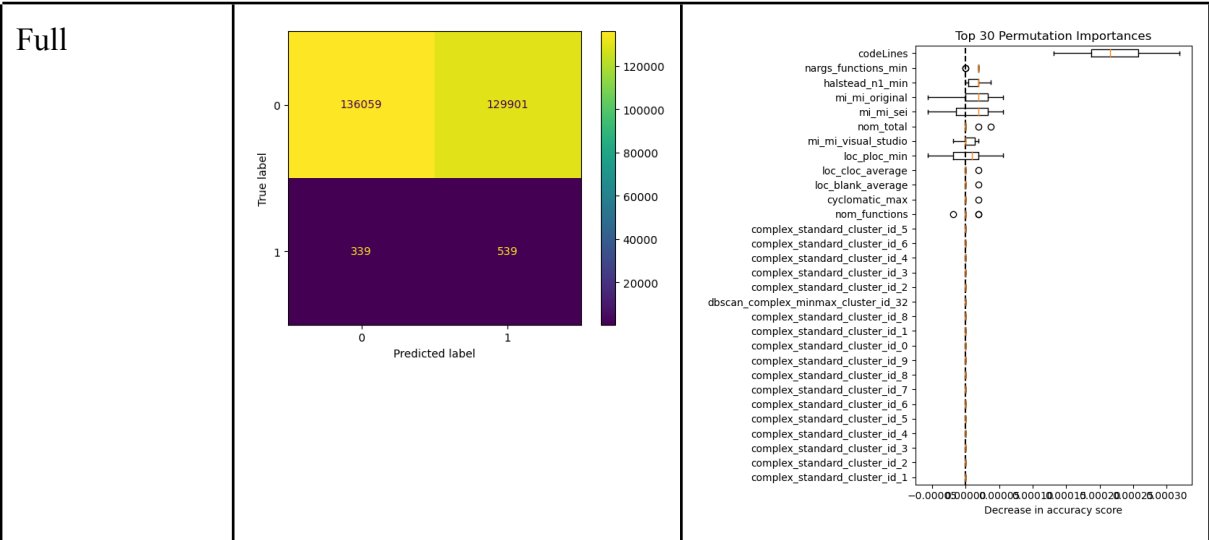
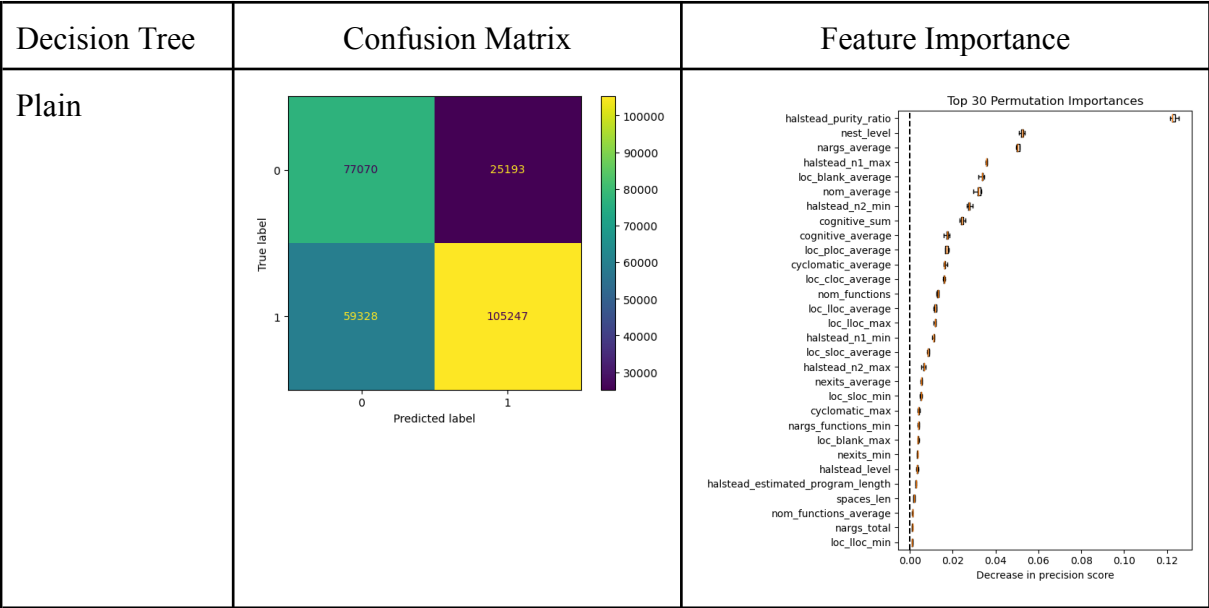
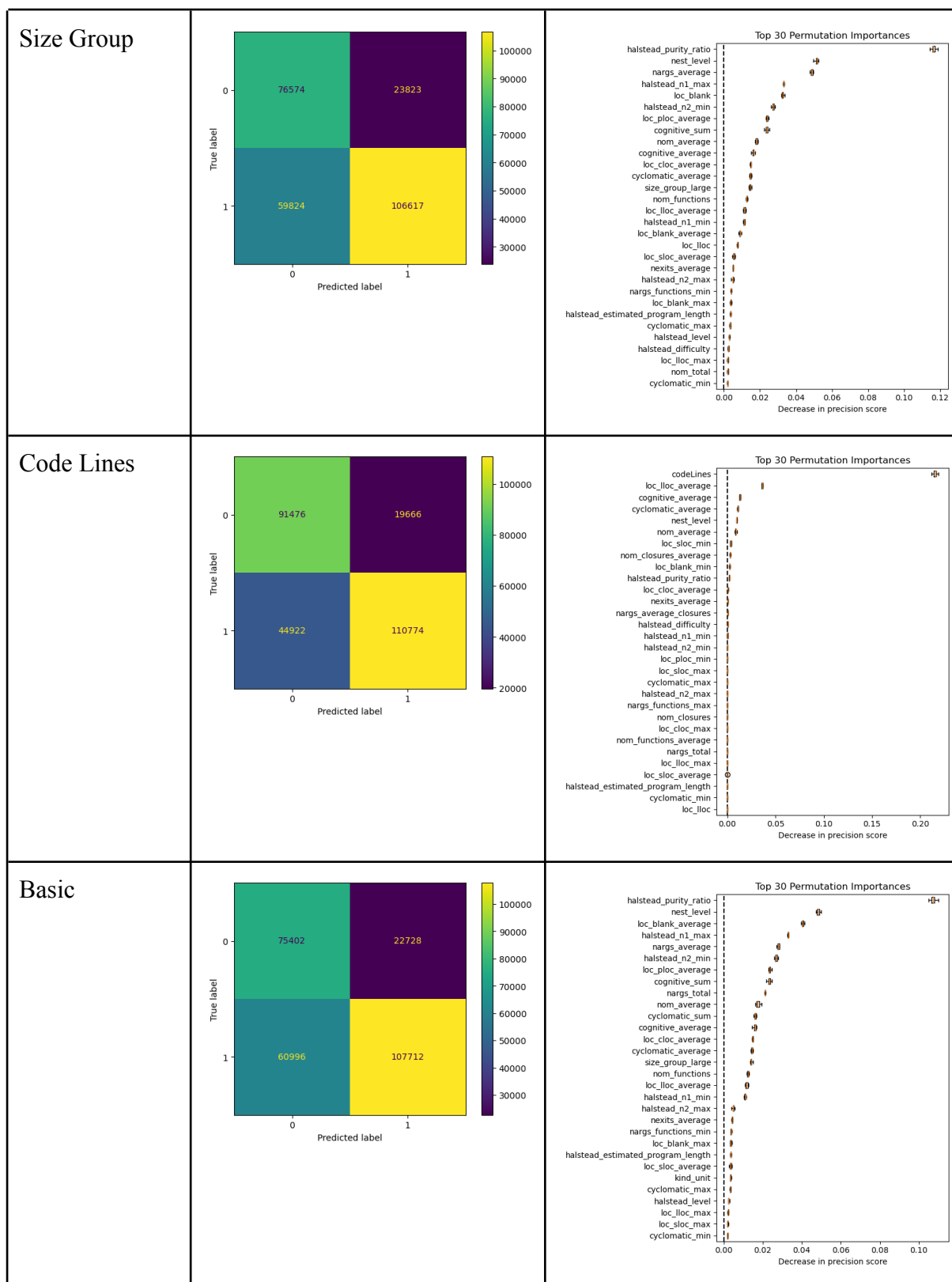


Figure. Confusion Matrix and Feature Importance for Perceptron algorithm for all data configurations for the accuracy metric.

Precision Score Results





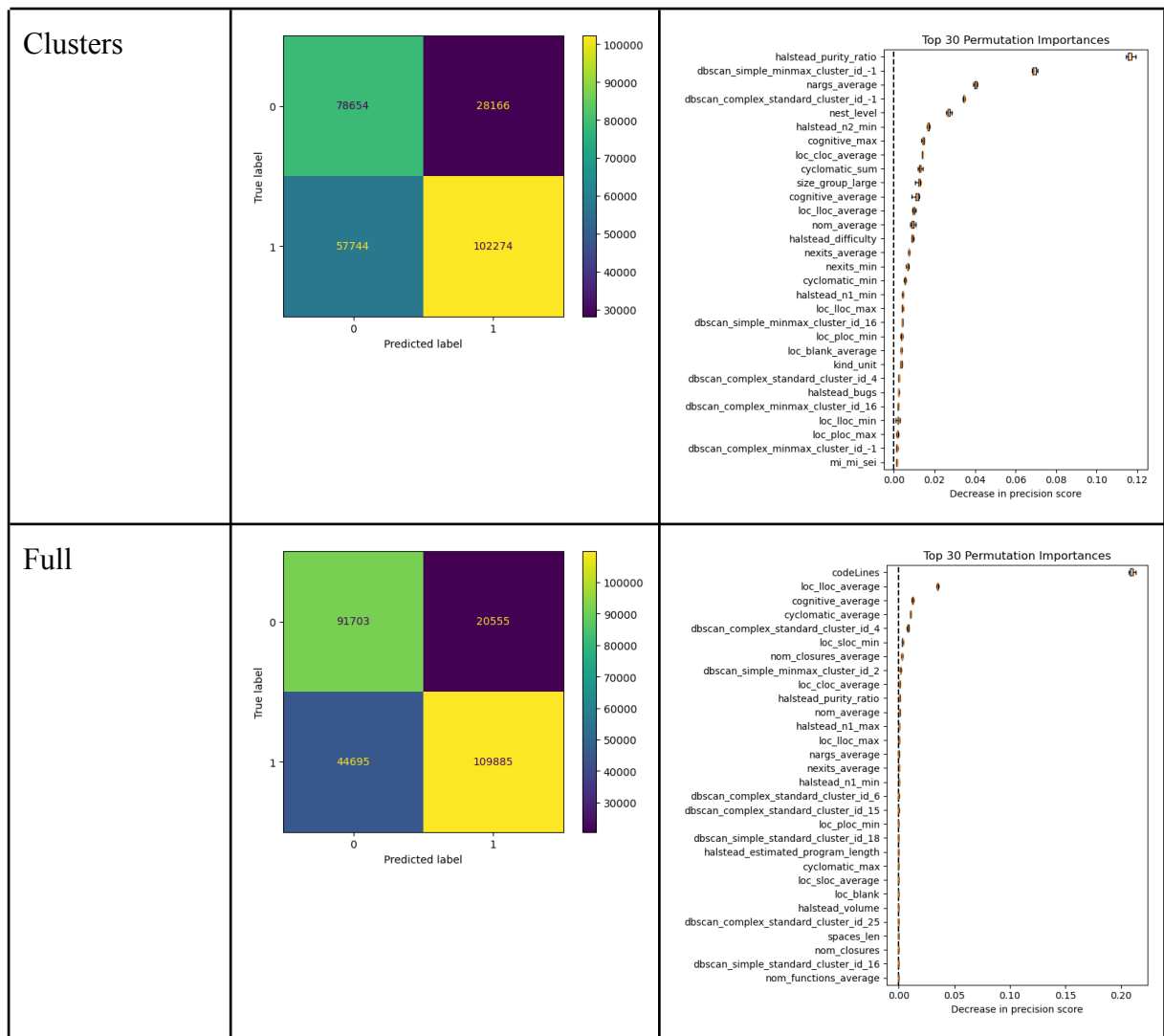
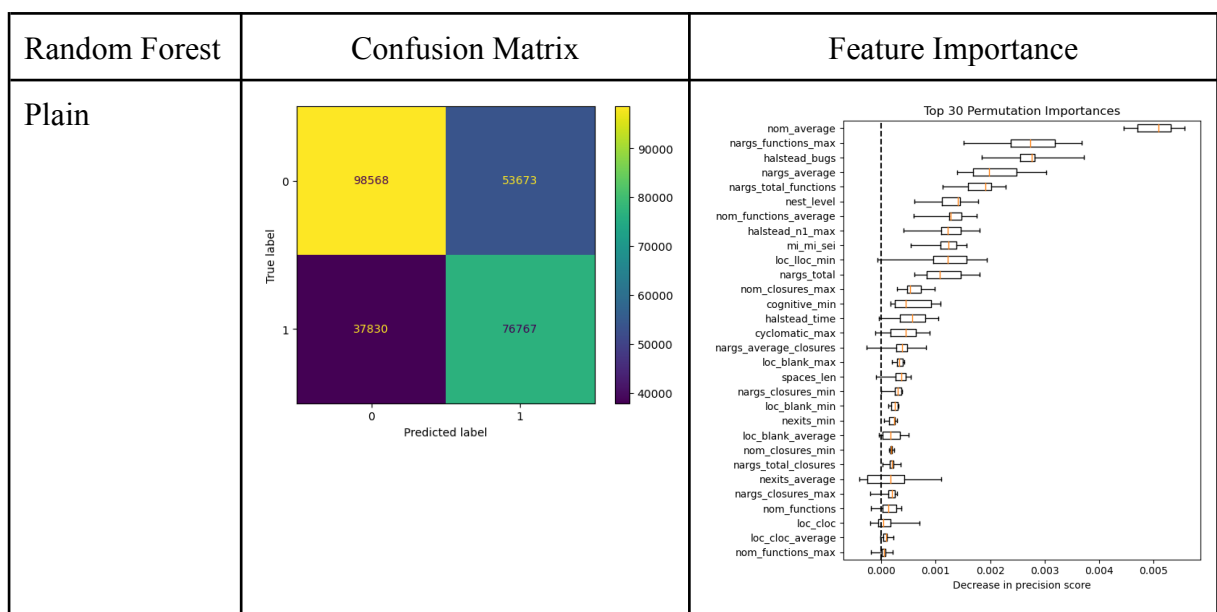
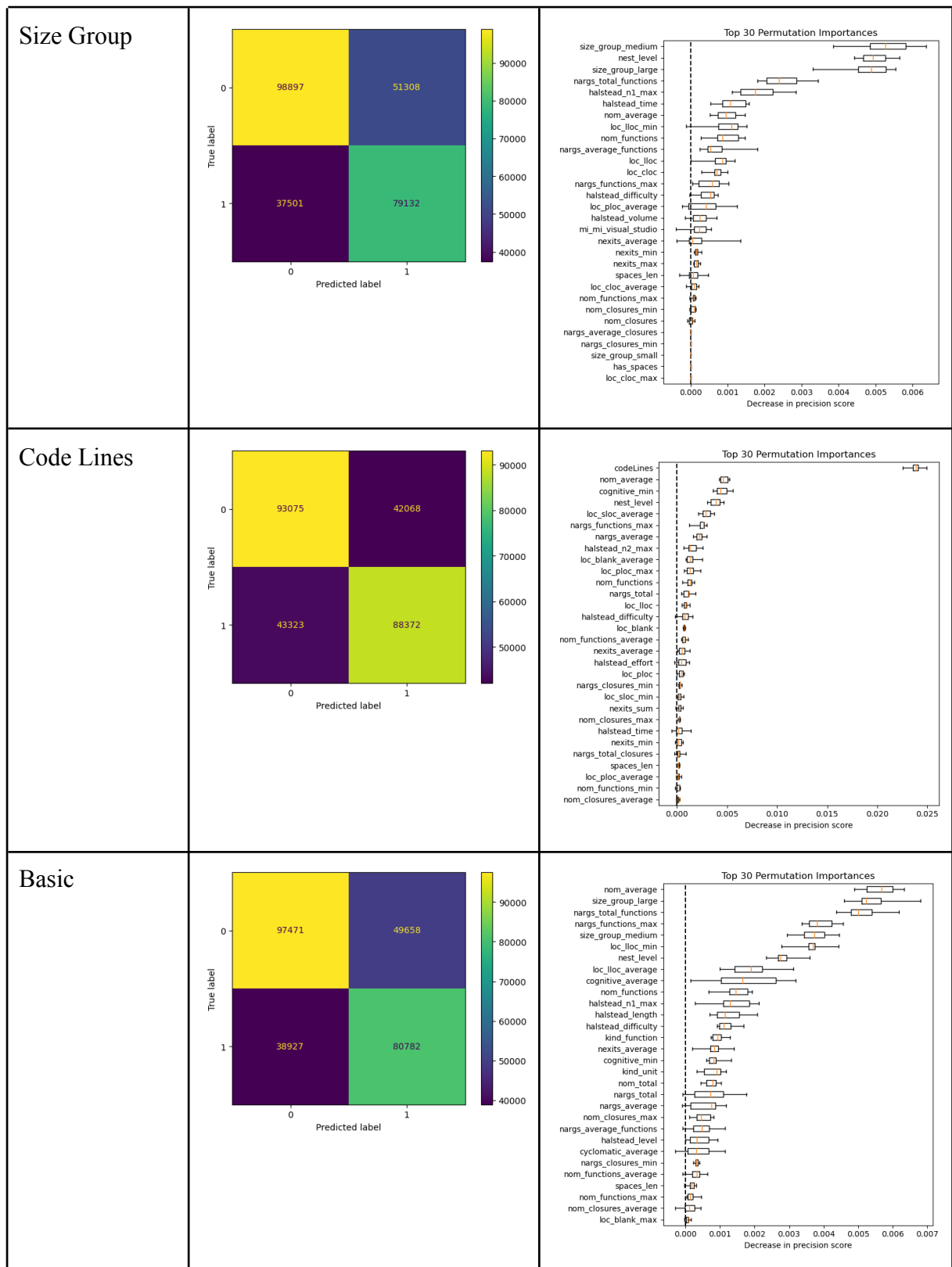


Figure. Confusion Matrix and Feature Importance for Decision Tree algorithm for all data configurations for the precision metric.





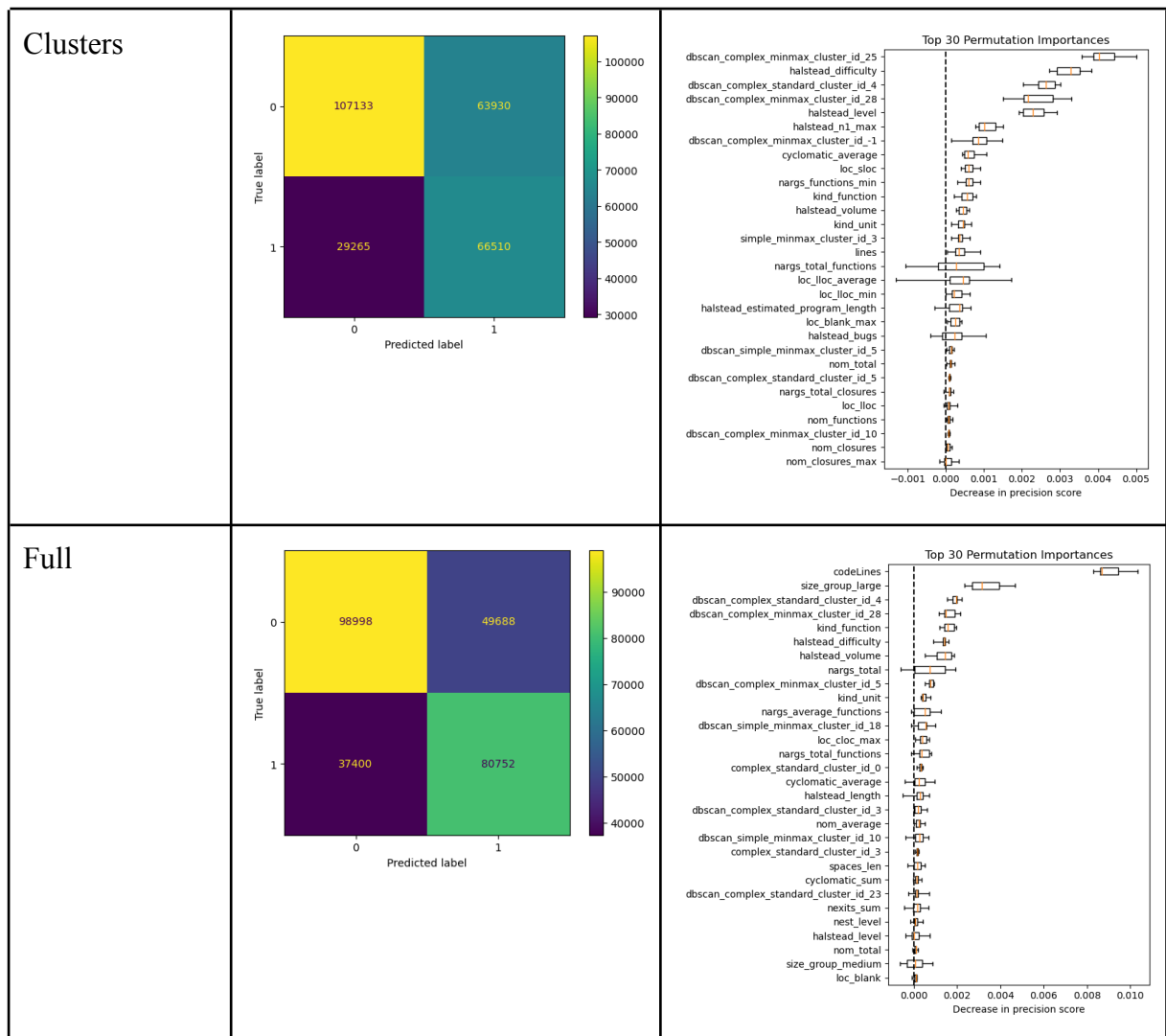
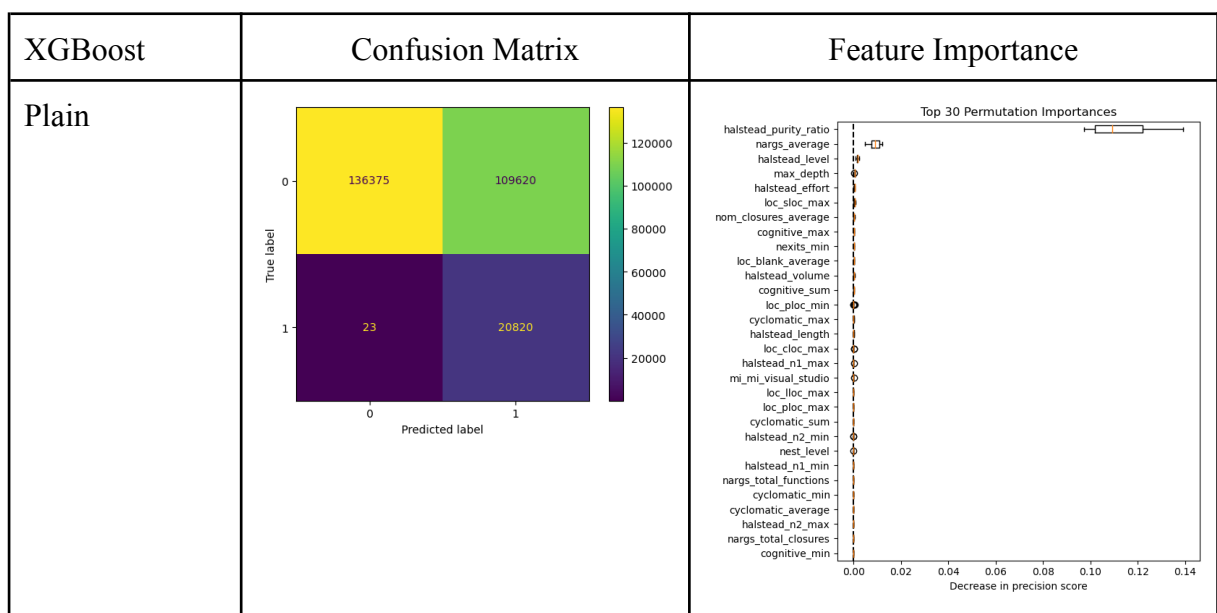
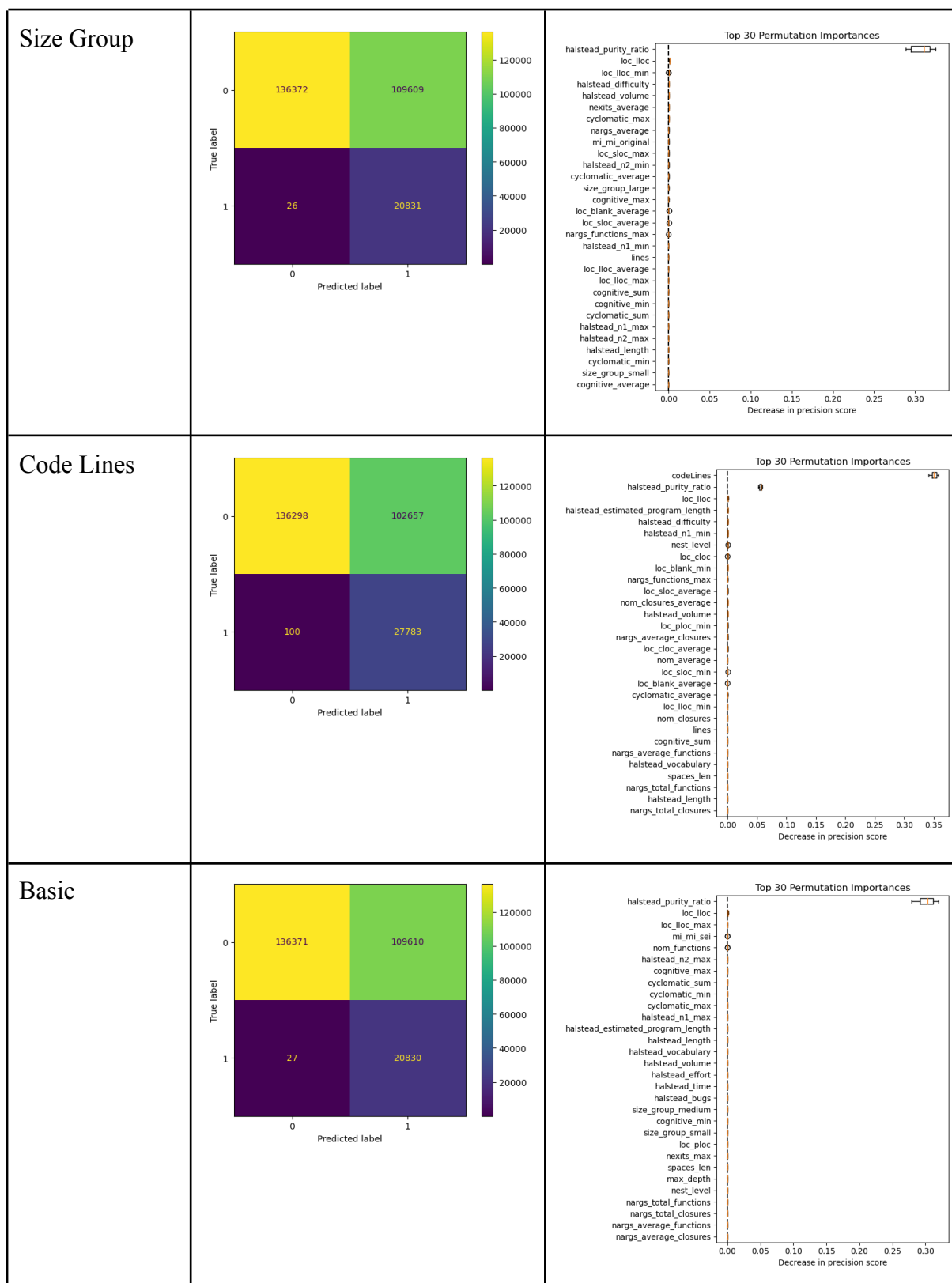


Figure. Confusion Matrix and Feature Importance for Random Forest algorithm for all data configurations for the precision metric.





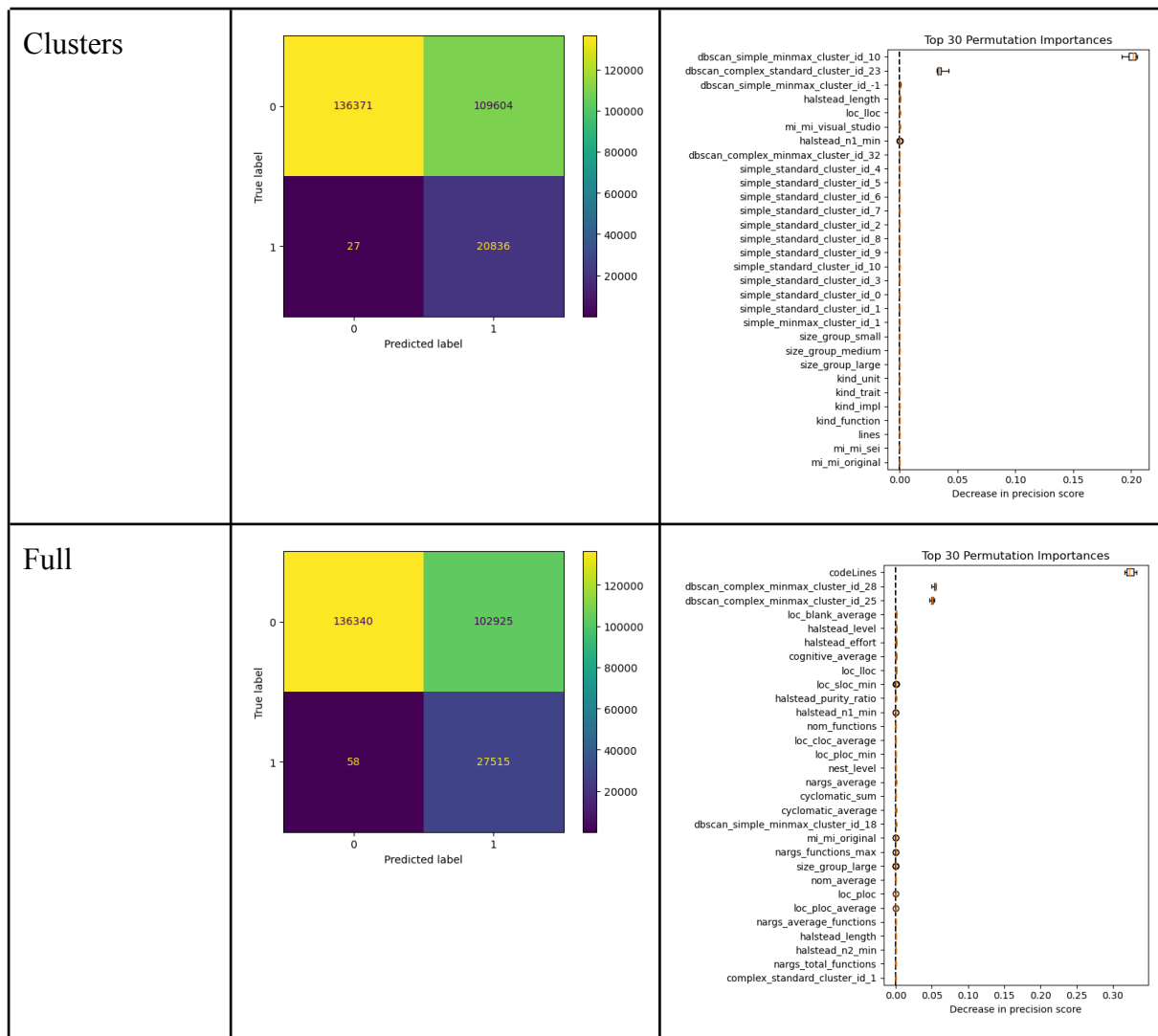
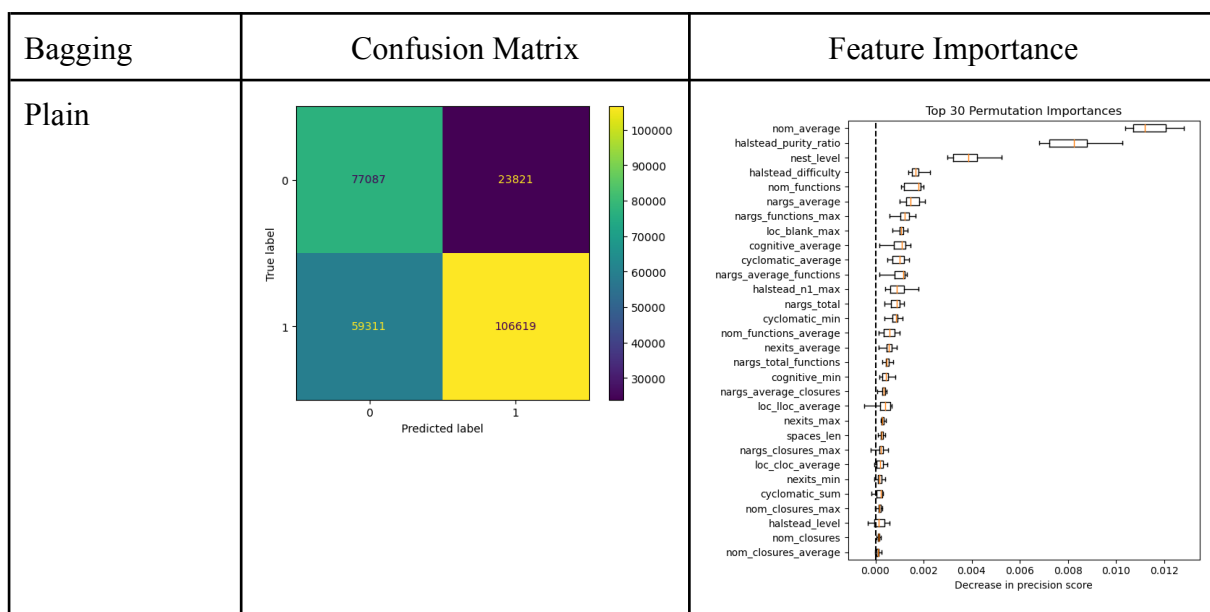
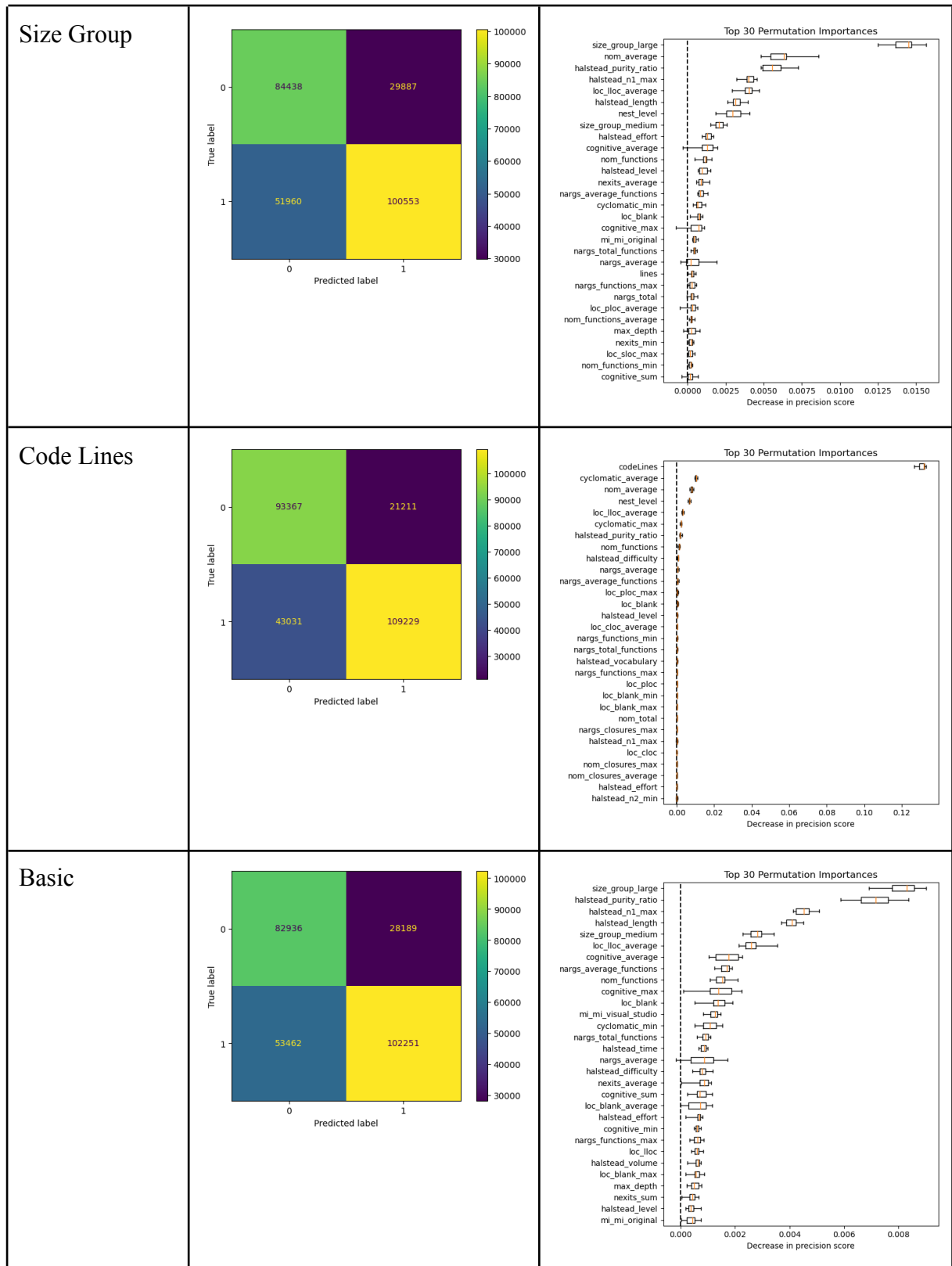


Figure. Confusion Matrix and Feature Importance for XGBoost algorithm for all data configurations for the precision metric.





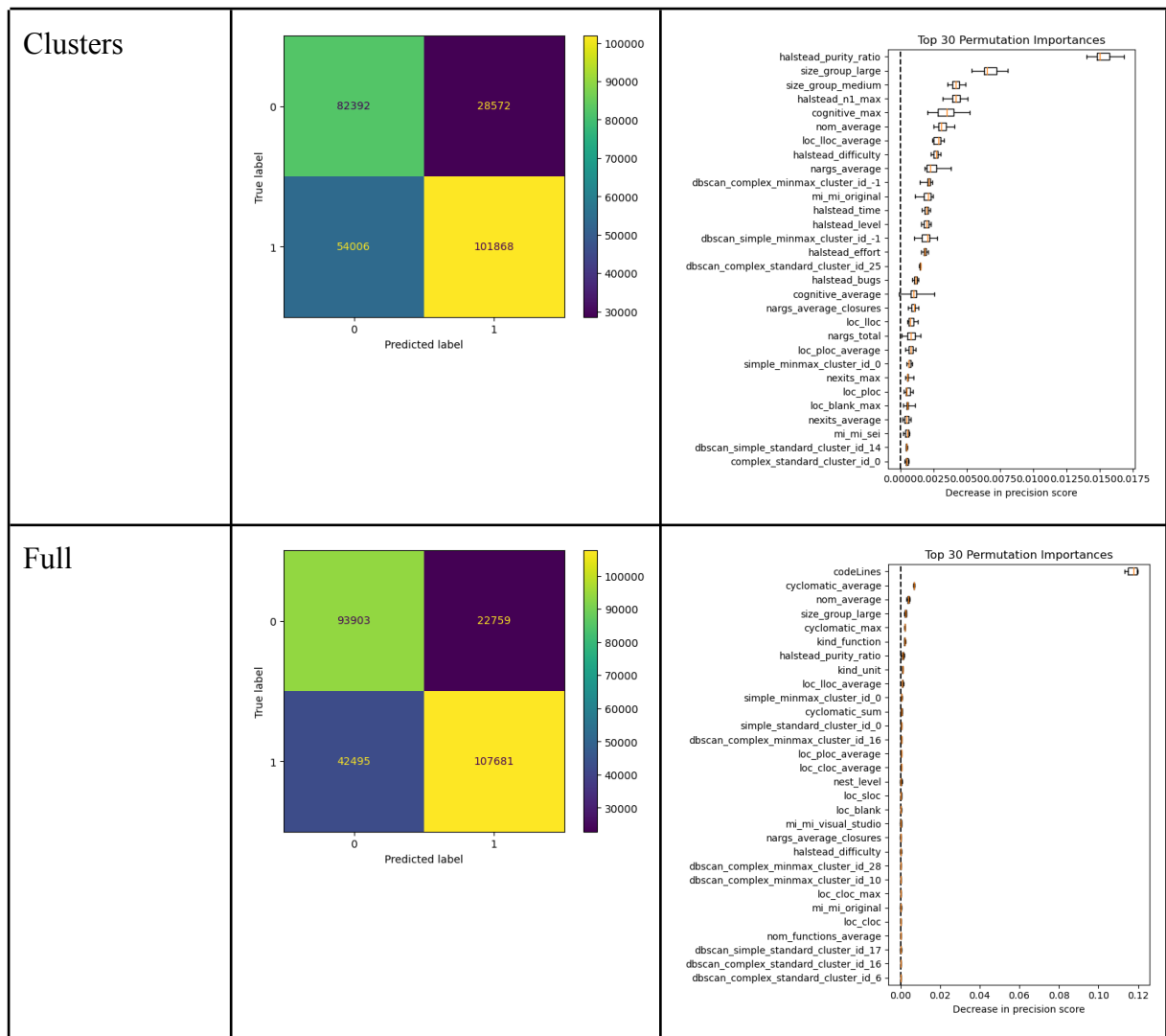
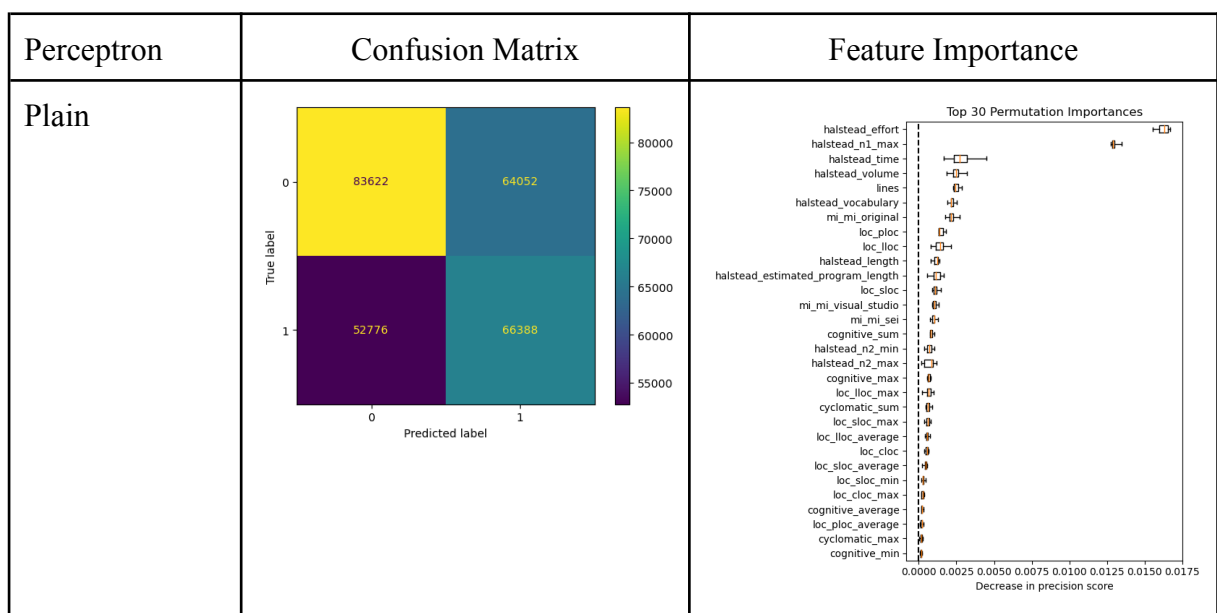
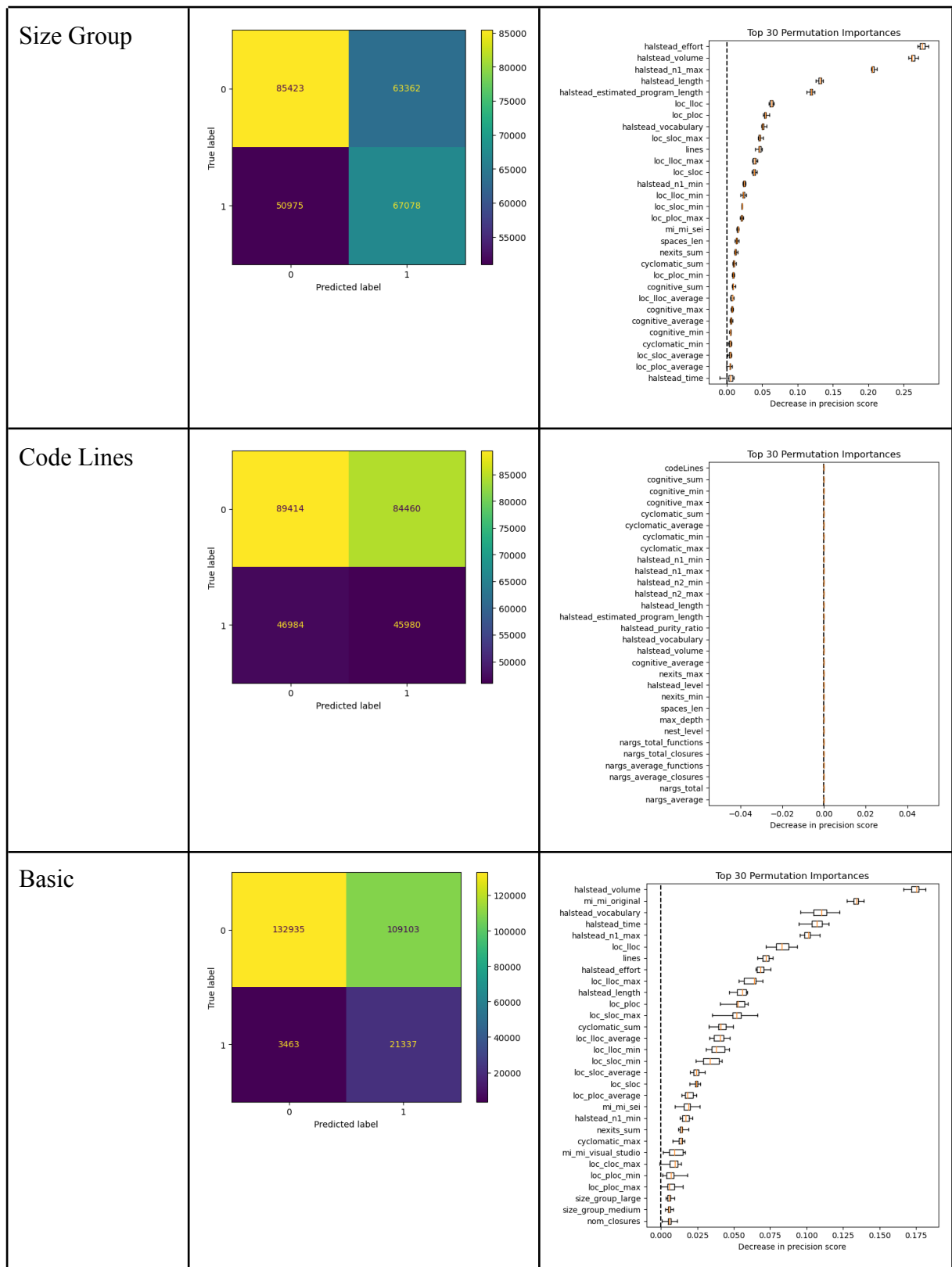


Figure. Confusion Matrix and Feature Importance for Bagging algorithm for all data configurations for the precision metric.





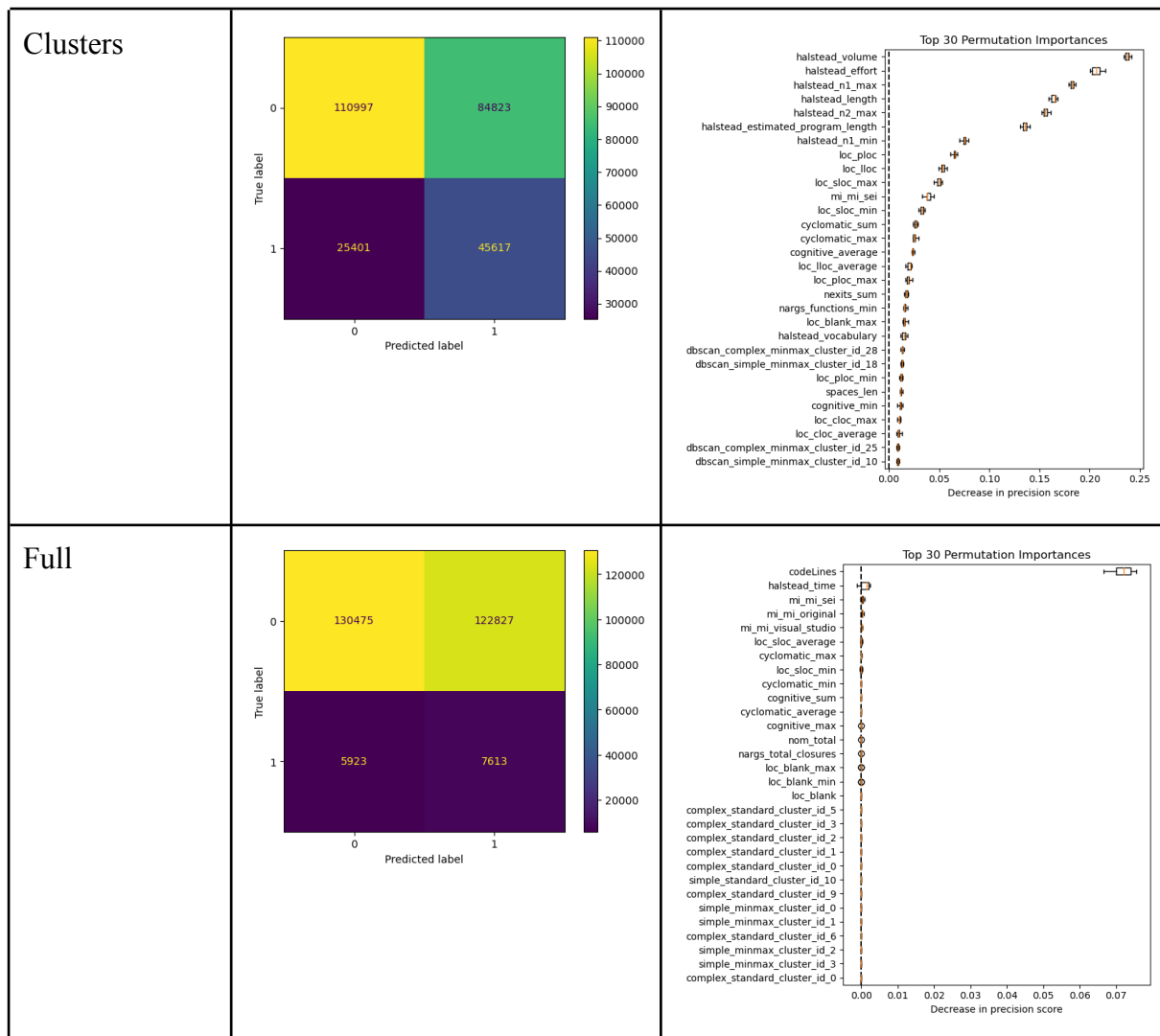
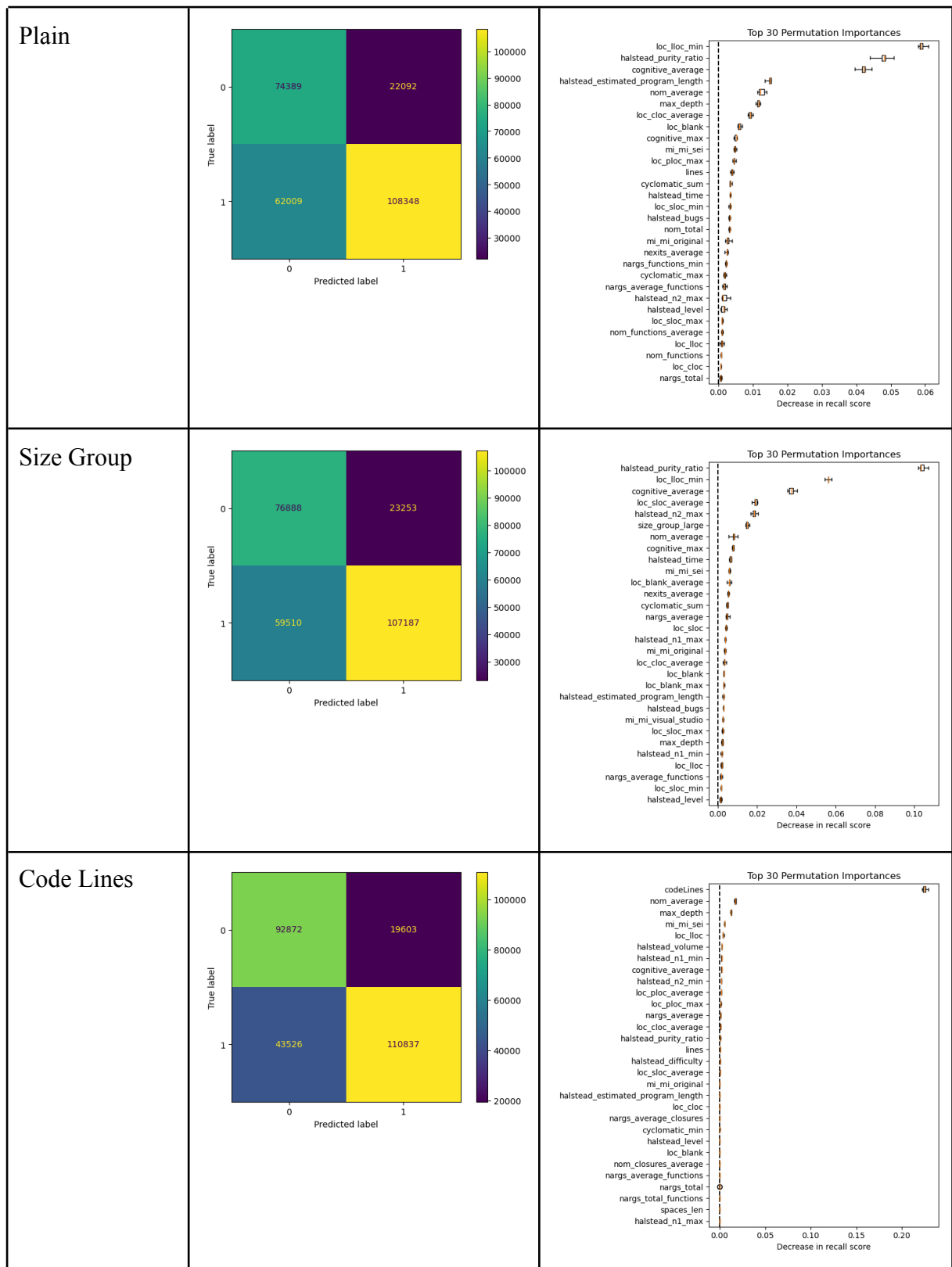


Figure. Confusion Matrix and Feature Importance for Perceptron algorithm for all data configurations for the precision metric.

Recall Score Results

Decision Tree	Confusion Matrix	Feature Importance
---------------	------------------	--------------------



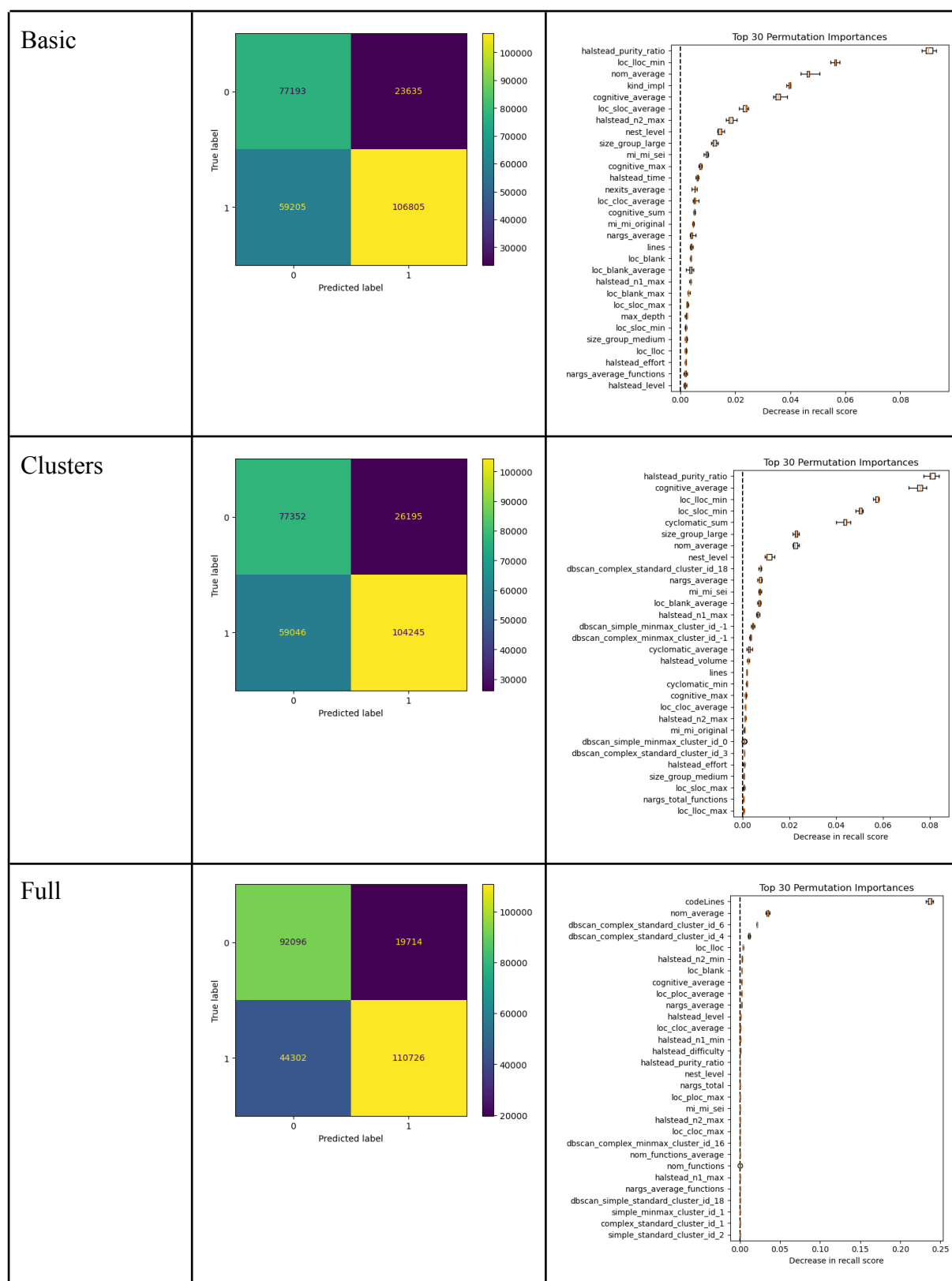
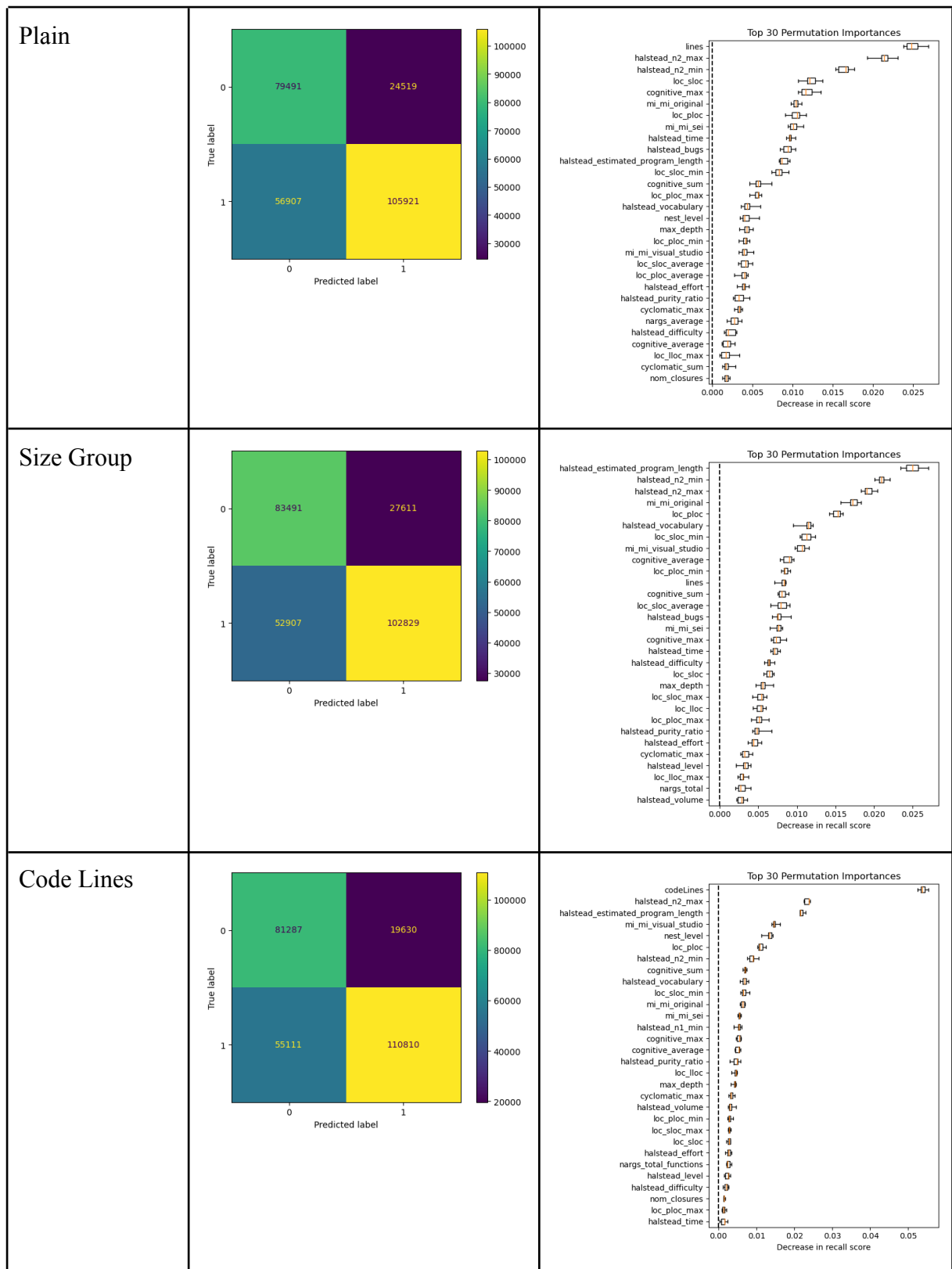


Figure. Confusion Matrix and Feature Importance for Decision Tree algorithm for all data.

Random Forest	Confusion Matrix	Feature Importance
---------------	------------------	--------------------



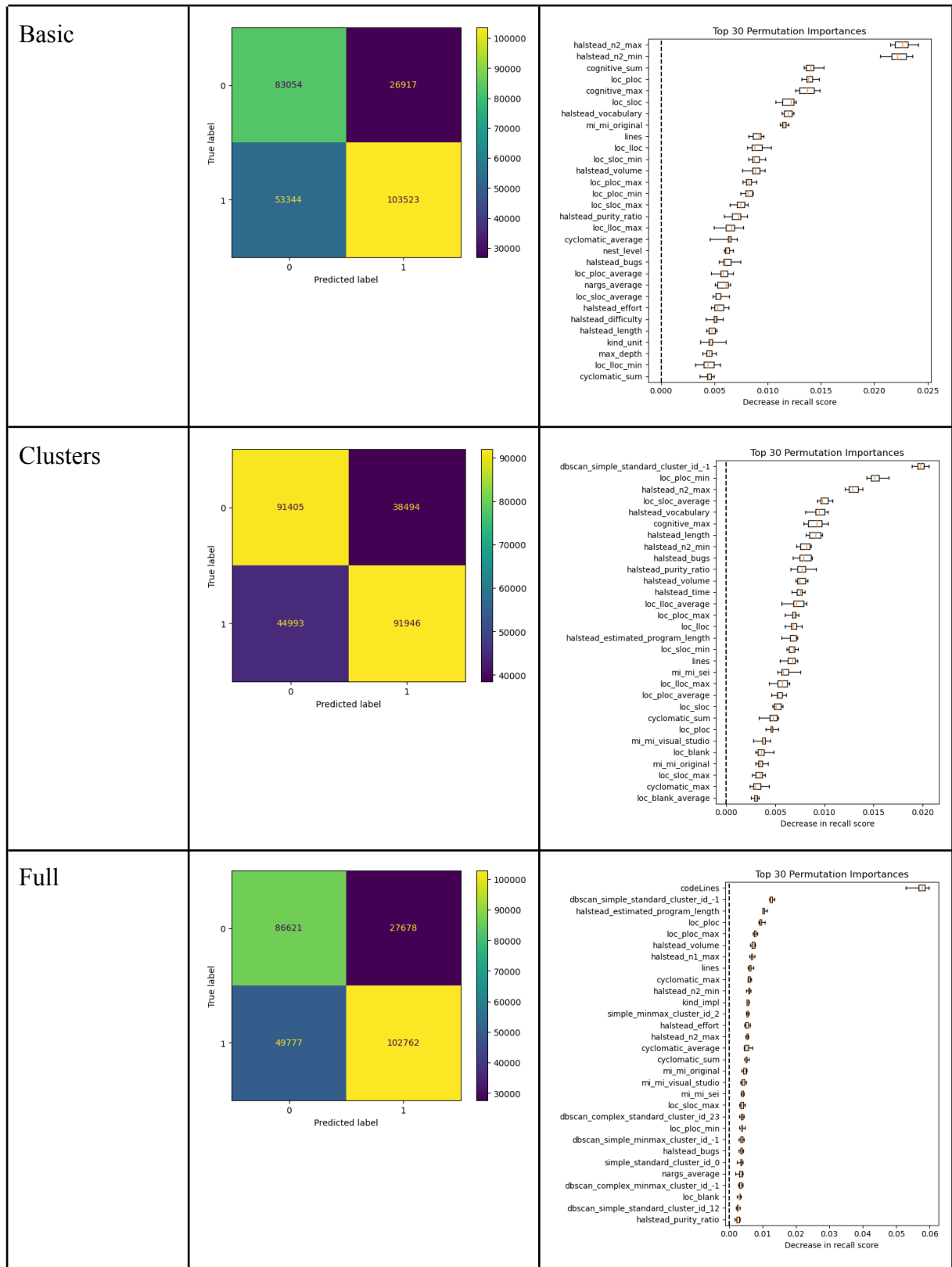
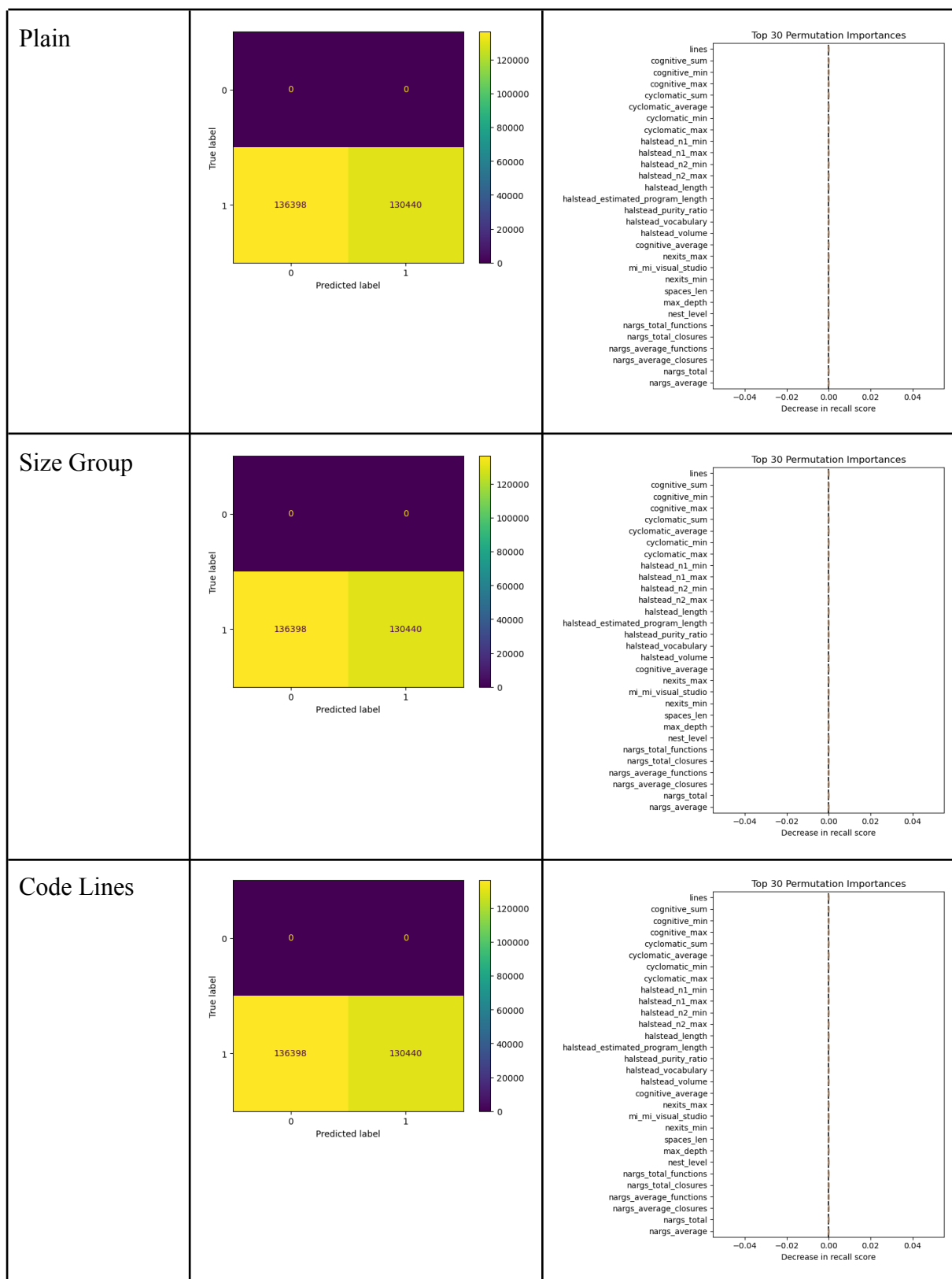


Figure. Confusion Matrix and Feature Importance for Random Forest algorithm for all data configurations for the recall metric.

XGBoost	Confusion Matrix	Feature Importance
---------	------------------	--------------------



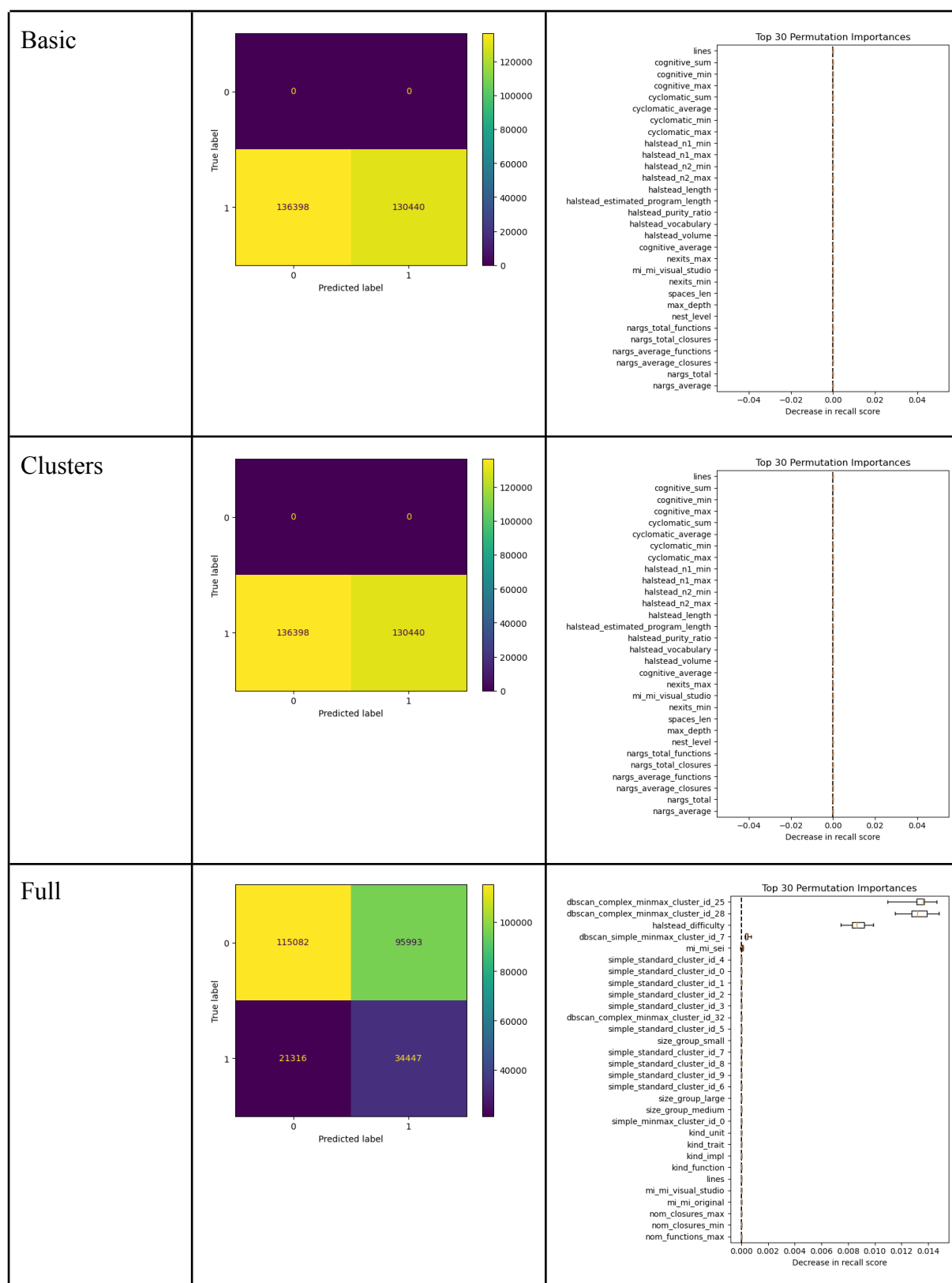
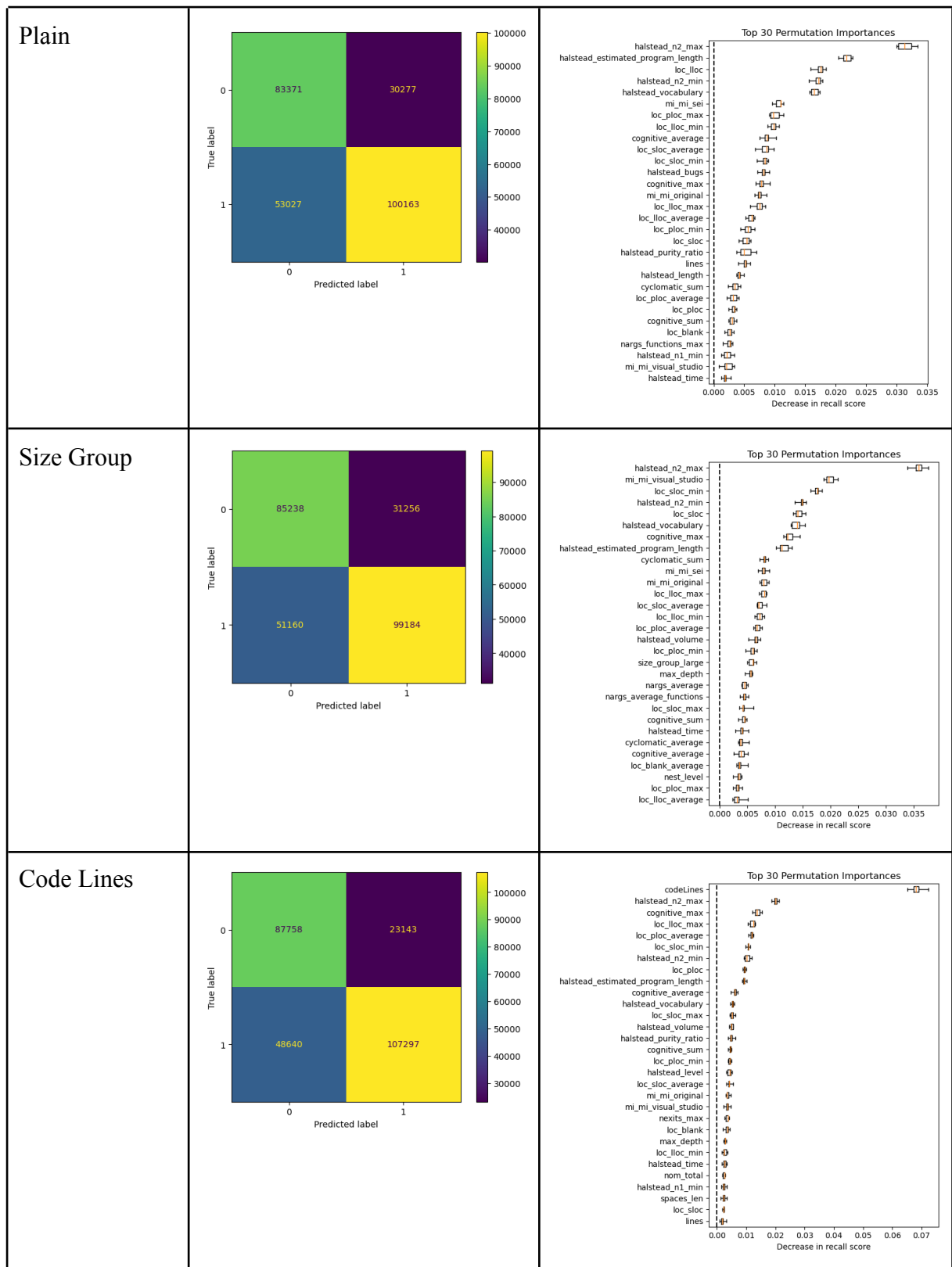


Figure. Confusion Matrix and Feature Importance for XGBoost algorithm for all data configurations for the recall metric.

Bagging	Confusion Matrix	Feature Importance
---------	------------------	--------------------



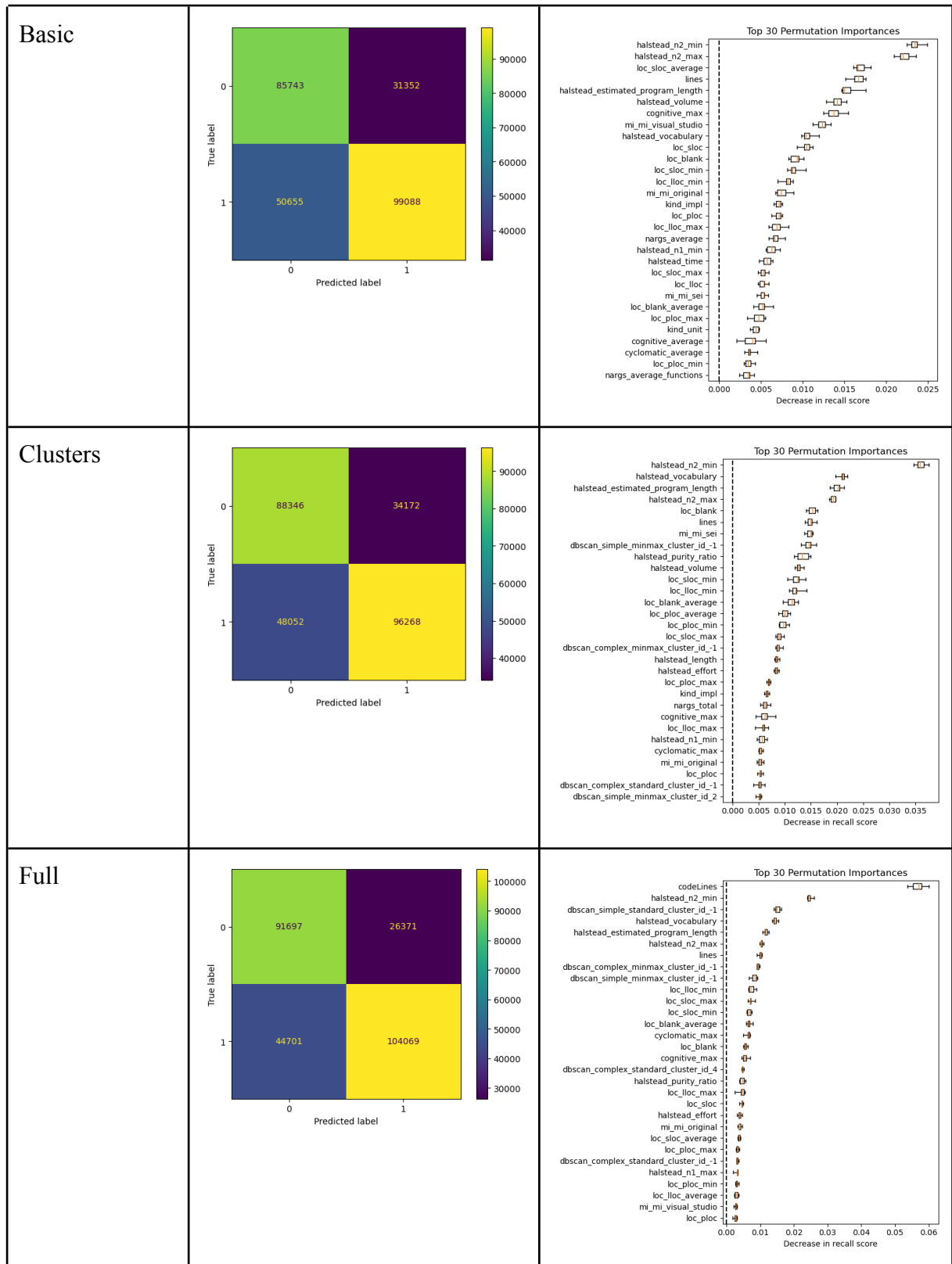
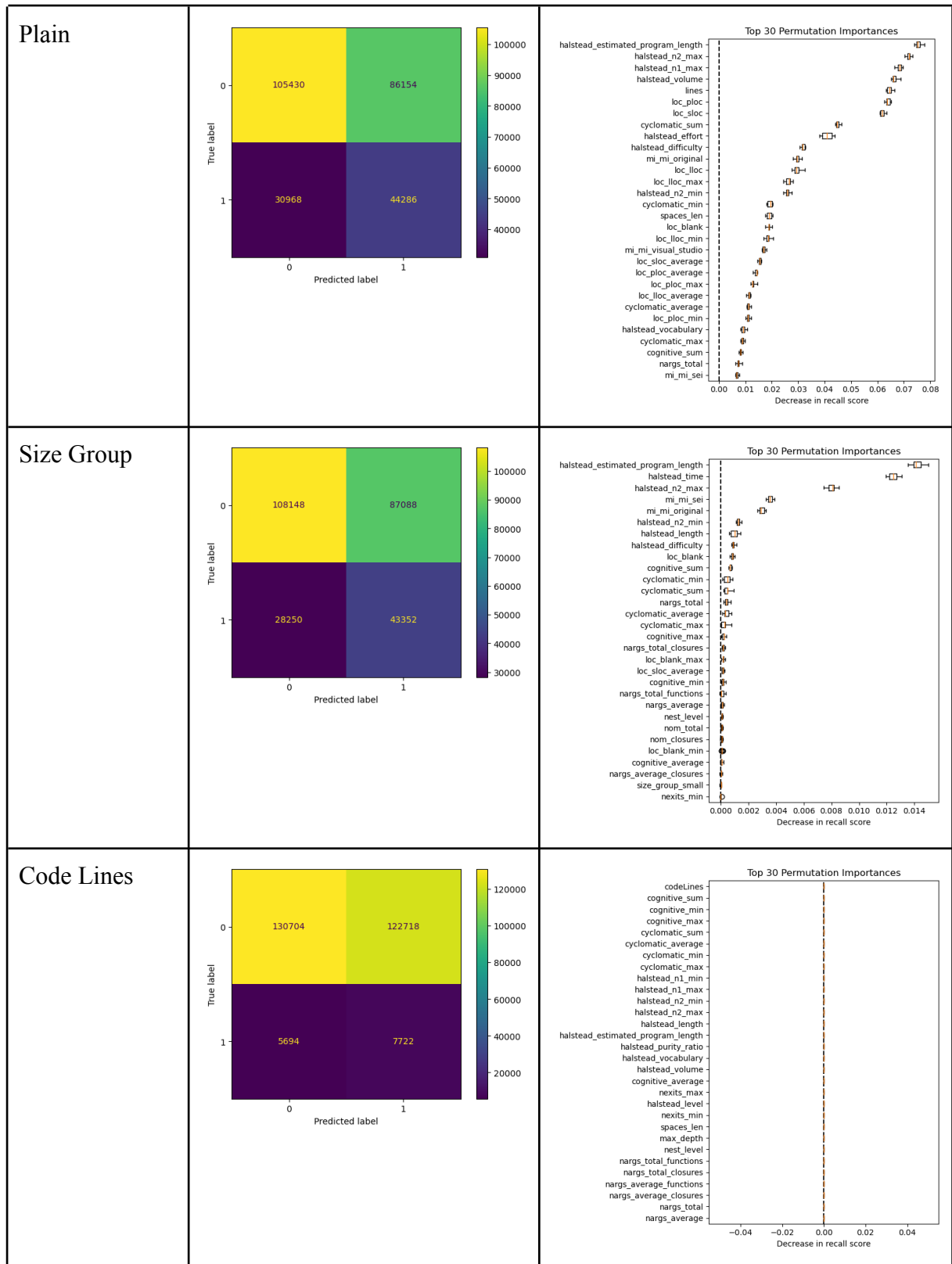


Figure. Confusion Matrix and Feature Importance for Bagging algorithm for all data configurations for the recall metric.

Perceptron	Confusion Matrix	Feature Importance
------------	------------------	--------------------



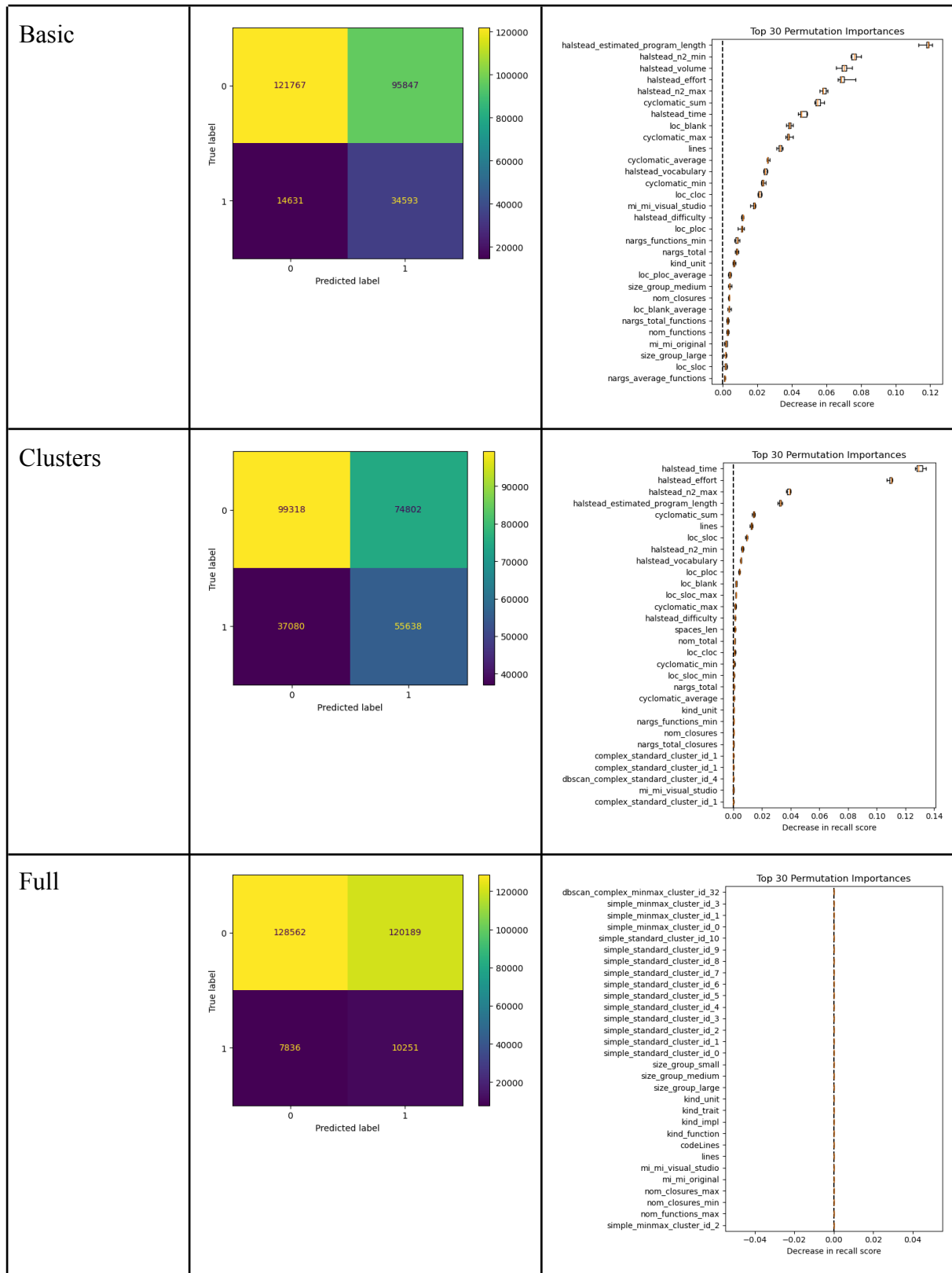


Figure. Confusion Matrix and Feature Importance for Perceptron algorithm for all data configurations for the recall metric.

F1 Score Results

Decision Tree	Confusion Matrix	Feature Importance									
Plain	<p>Confusion Matrix for Plain model:</p> <table border="1"> <thead> <tr> <th>True \ Predicted</th> <th>0</th> <th>1</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>74491</td> <td>22145</td> </tr> <tr> <th>1</th> <td>61907</td> <td>108295</td> </tr> </tbody> </table>	True \ Predicted	0	1	0	74491	22145	1	61907	108295	<p>Top 30 Permutation Importances for Plain model:</p> <ul style="list-style-type: none"> nargs_average halstead_purity_ratio loc_llcc_max loc_llcc_min nom_average cognitive_average loc_sloc_max nest_level max_depth loc_cloc_average halstead_n1_min cyclomatic_sum cyclomatic_average halstead_n2_min loc_llcc_average halstead_estimated_program_length loc_ploc_max halstead_level loc_sloc_average loc_blank halstead_n2_max nextits_average mi_mi_visual_studio cyclomatic_max loc_sloc_min nargs_functions_min cognitive_max loc_blank_max nom_total halstead_bugs
True \ Predicted	0	1									
0	74491	22145									
1	61907	108295									
Size Group	<p>Confusion Matrix for Size Group model:</p> <table border="1"> <thead> <tr> <th>True \ Predicted</th> <th>0</th> <th>1</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>76596</td> <td>22925</td> </tr> <tr> <th>1</th> <td>59802</td> <td>107515</td> </tr> </tbody> </table>	True \ Predicted	0	1	0	76596	22925	1	59802	107515	<p>Top 30 Permutation Importances for Size Group model:</p> <ul style="list-style-type: none"> halstead_purity_ratio nargs_total_functions loc_llcc_max loc_llcc_min nest_level cognitive_average nom_average nargs_average size_group_large halstead_n2_min loc_sloc_average loc_cloc_average cyclomatic_average loc_llcc_average halstead_n2_max nextits_average max_depth loc_ploc_max loc_sloc_max loc_blank loc_blank_max halstead_level cyclomatic_sum mi_mi_sei loc_blank_average cognitive_max halstead_n1_max mi_mi_visual_studio halstead_estimated_program_length mi_mi_original
True \ Predicted	0	1									
0	76596	22925									
1	59802	107515									
Code Lines	<p>Confusion Matrix for Code Lines model:</p> <table border="1"> <thead> <tr> <th>True \ Predicted</th> <th>0</th> <th>1</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>92881</td> <td>19547</td> </tr> <tr> <th>1</th> <td>43517</td> <td>110893</td> </tr> </tbody> </table>	True \ Predicted	0	1	0	92881	19547	1	43517	110893	<p>Top 30 Permutation Importances for Code Lines model:</p> <ul style="list-style-type: none"> codeLines nom_average cognitive_average cyclomatic_average cyclomatic_min max_depth loc_llcc_average nest_level loc_cloc_average loc_blank loc_llcc mi_mi_sei halstead_difficulty halstead_volume loc_ploc_average halstead_n1_min loc_sloc_average loc_sloc_min halstead_effort halstead_purity_ratio loc_ploc_max nargs_average loc_blank_min nom_closures_average lines nargs_average_closures halstead_length cyclomatic_sum loc_blank_average halstead_estimated_program_length
True \ Predicted	0	1									
0	92881	19547									
1	43517	110893									

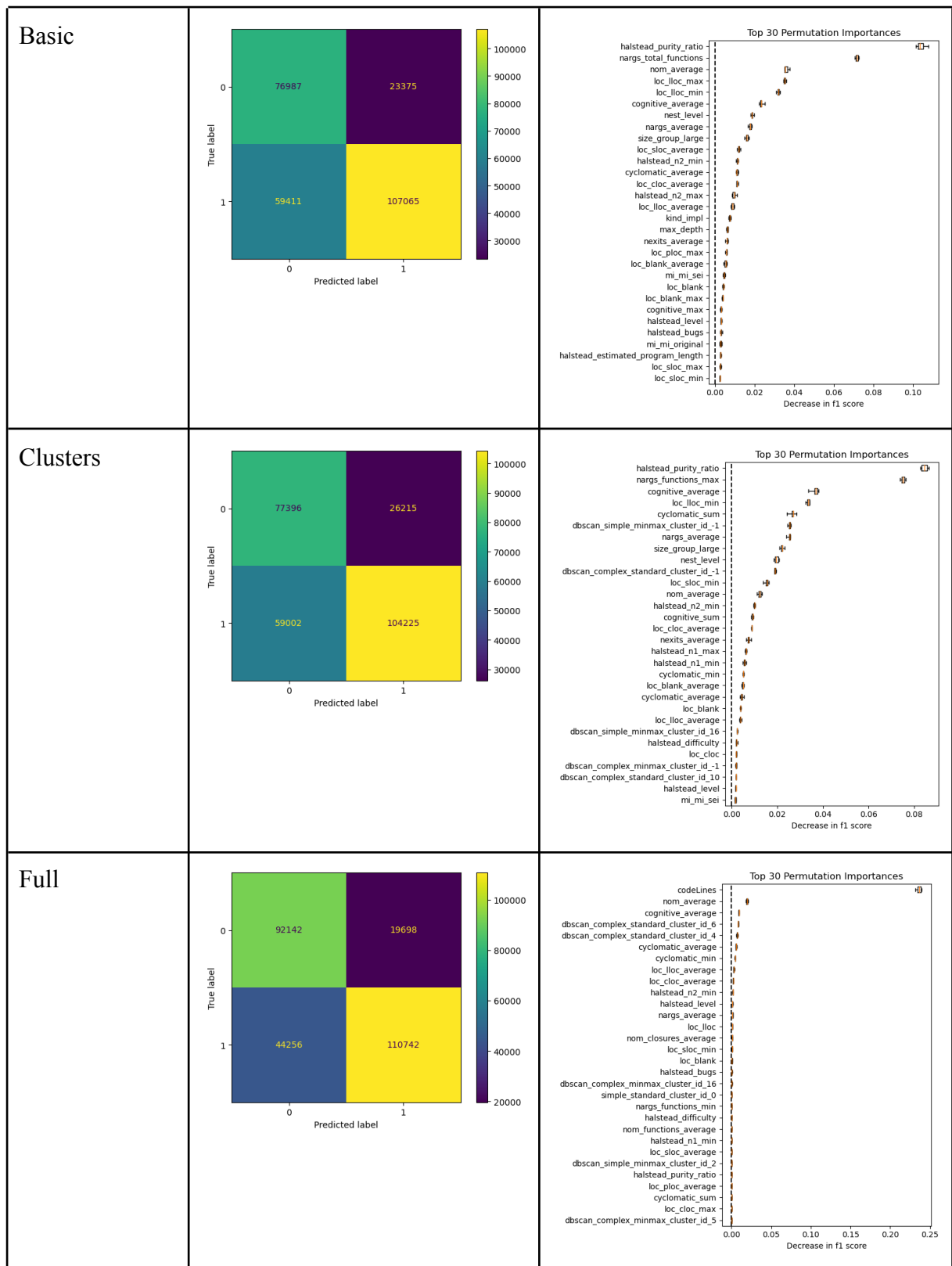
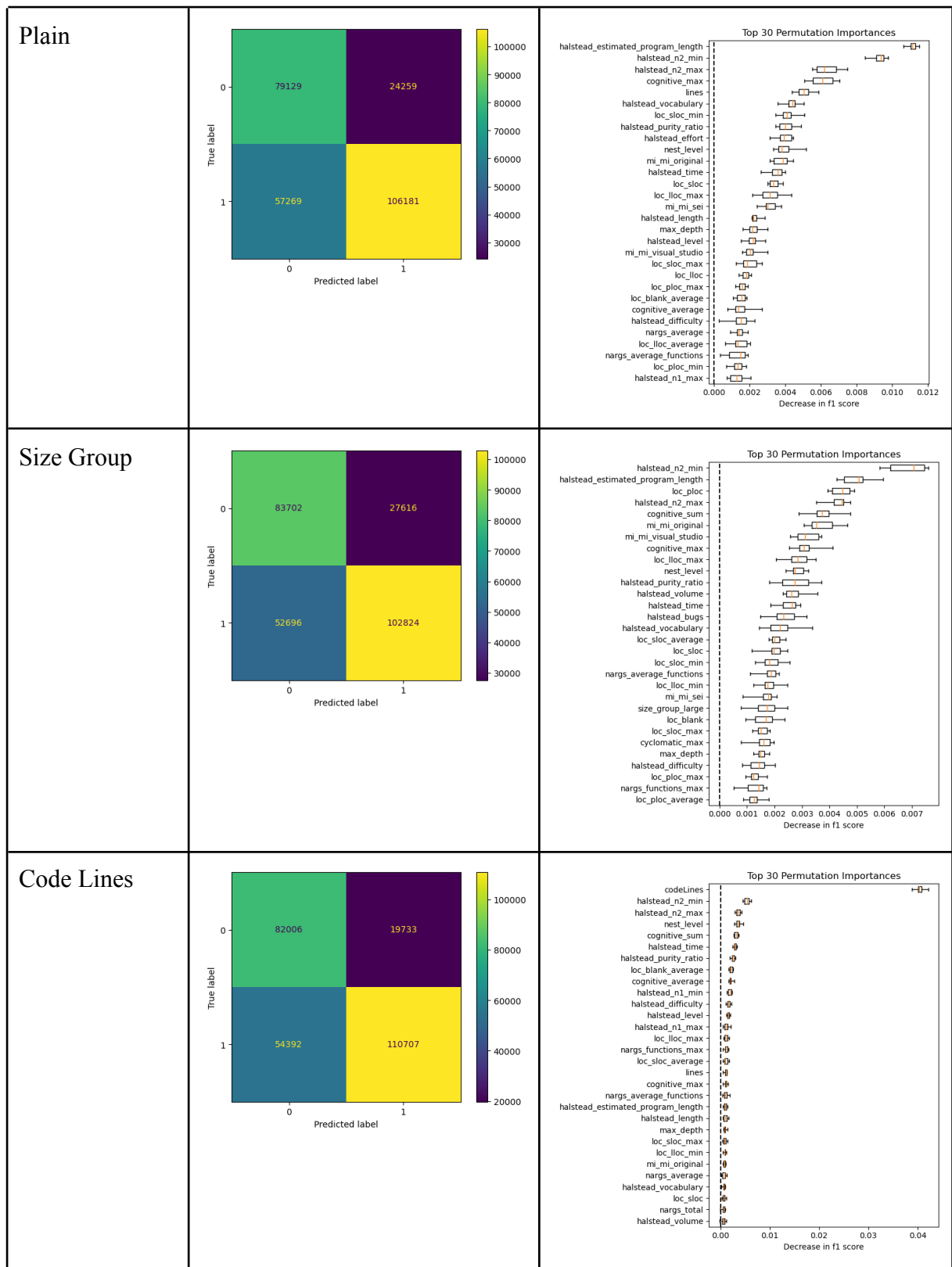


Figure. Confusion Matrix and Feature Importance for Decision Tree algorithm for all data configurations for the F1 metric.

Random Forest	Confusion Matrix	Feature Importance
---------------	------------------	--------------------



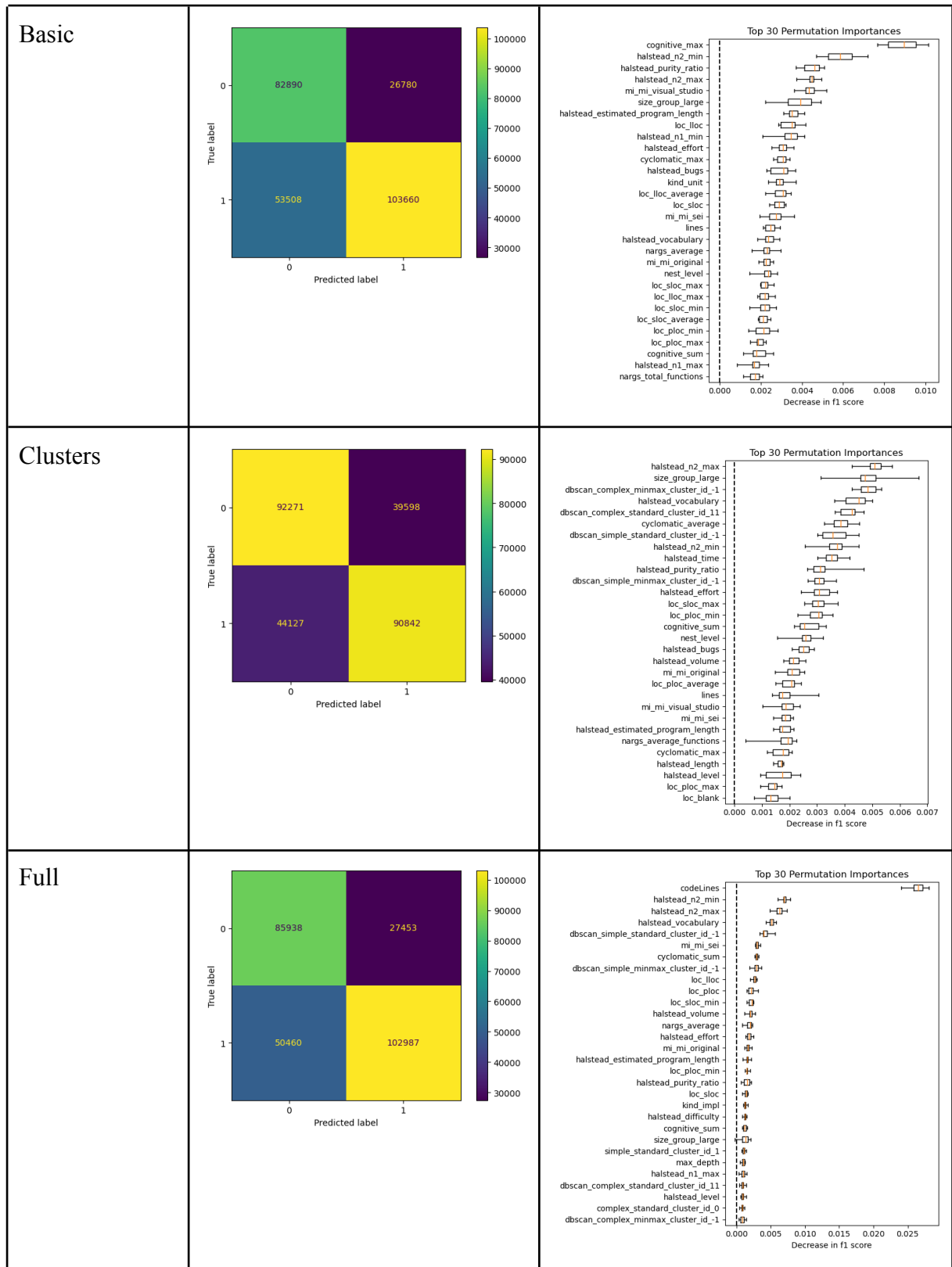
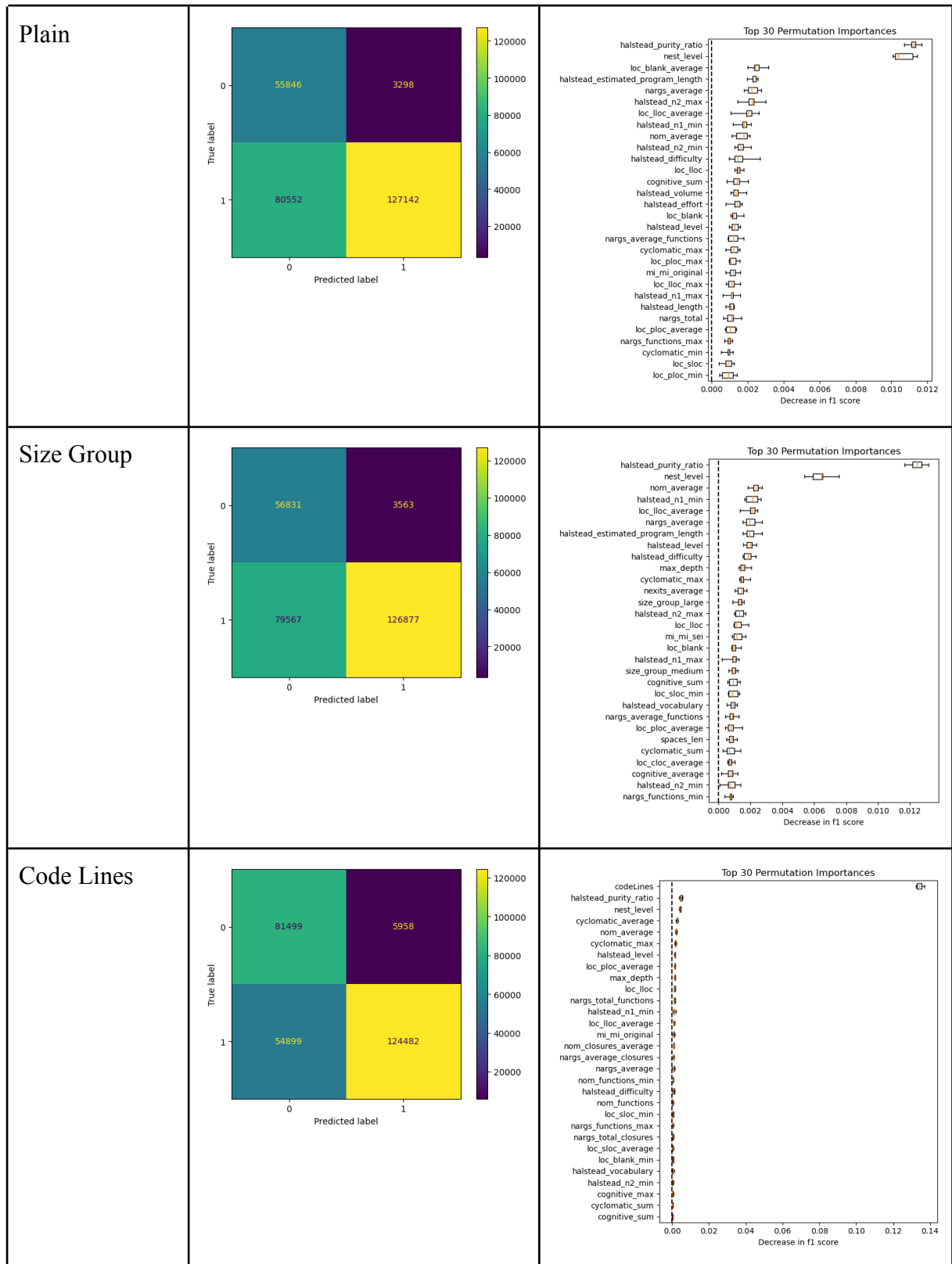


Figure. Confusion Matrix and Feature Importance for Random Forest algorithm for all data configurations for the F1 metric.

XGBoost	Confusion Matrix	Feature Importance
---------	------------------	--------------------



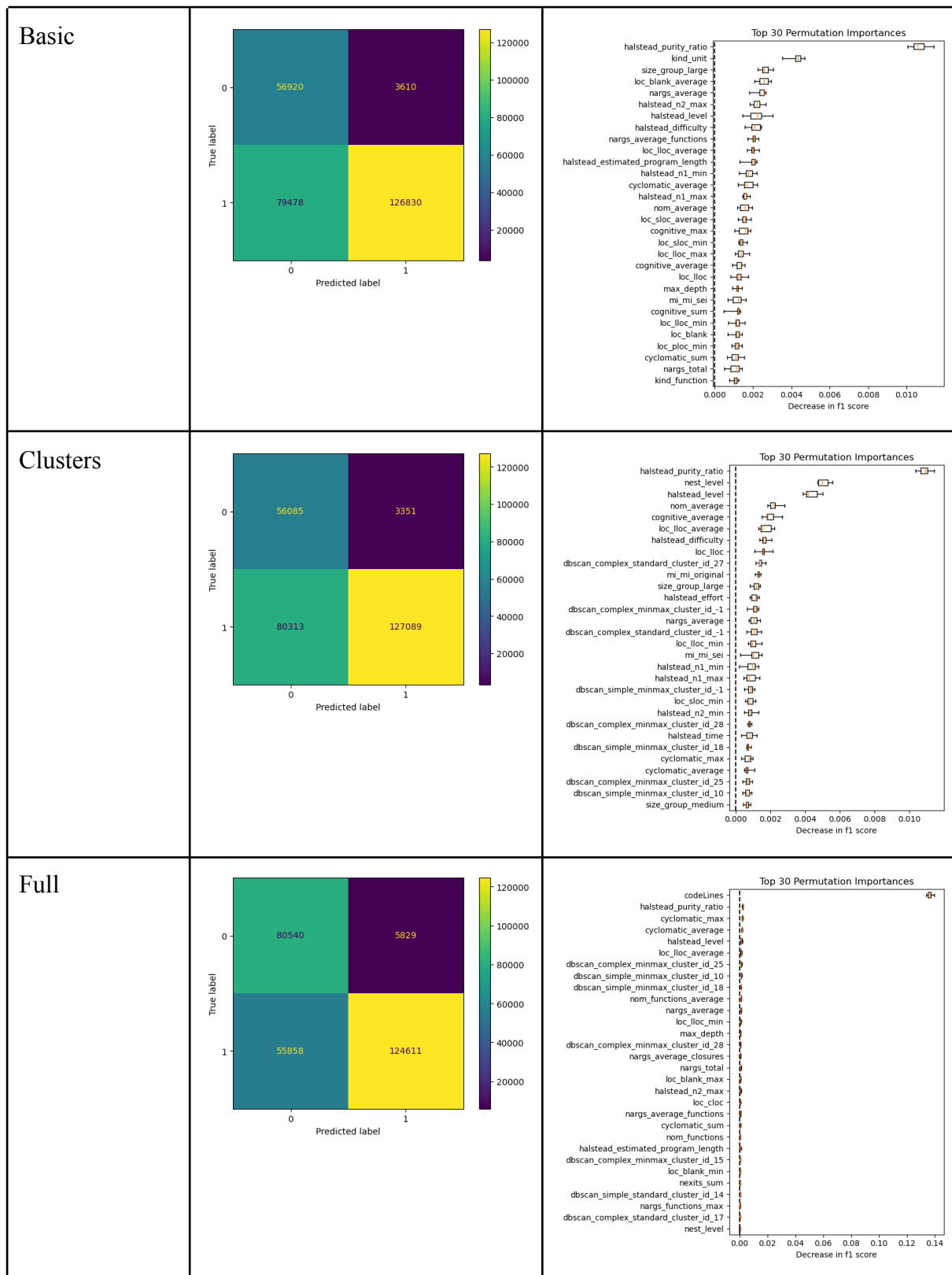
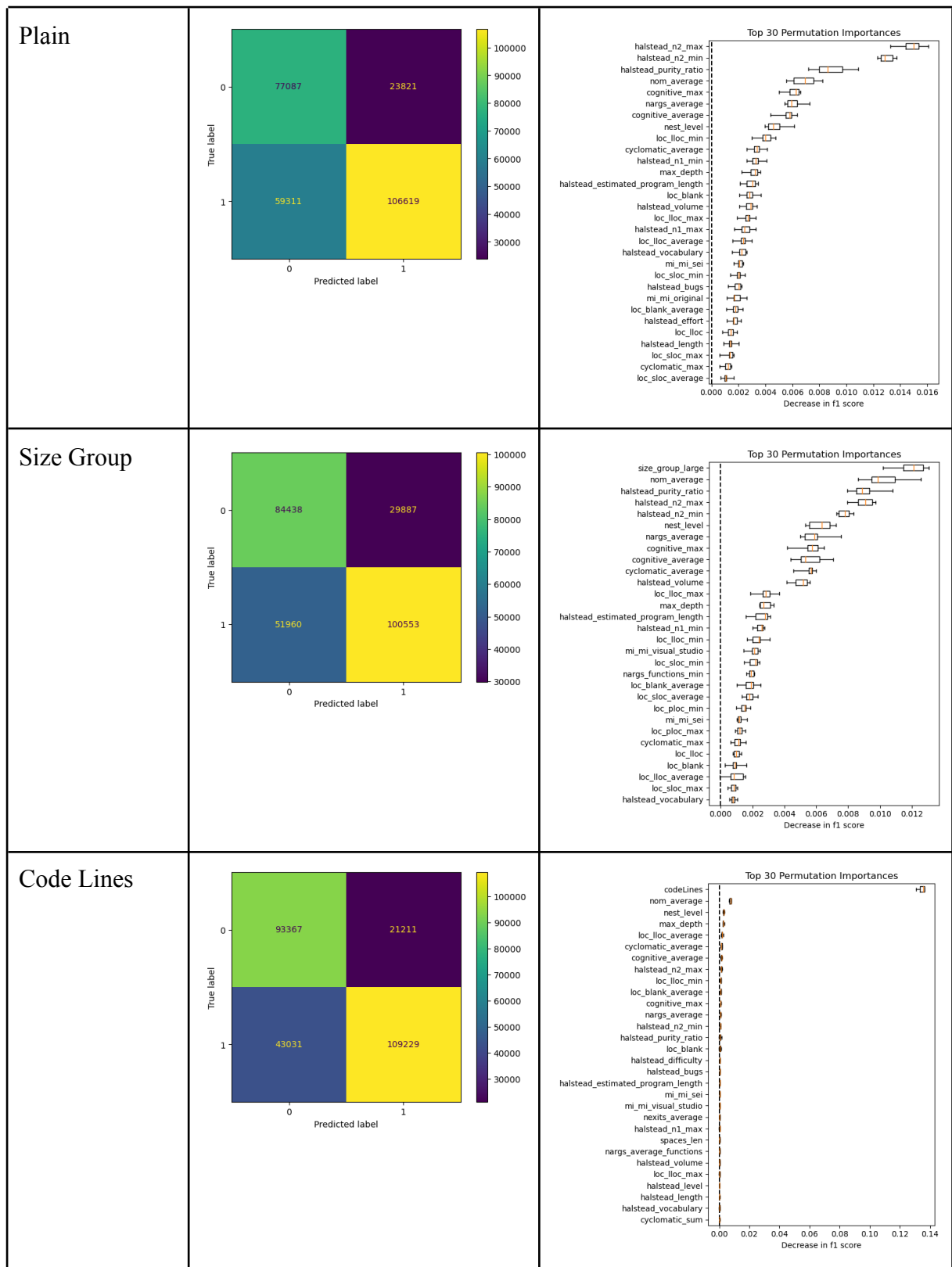


Figure. Confusion Matrix and Feature Importance for XGBoost algorithm for all data configurations for the F1 metric.

Bagging	Confusion Matrix	Feature Importance
---------	------------------	--------------------



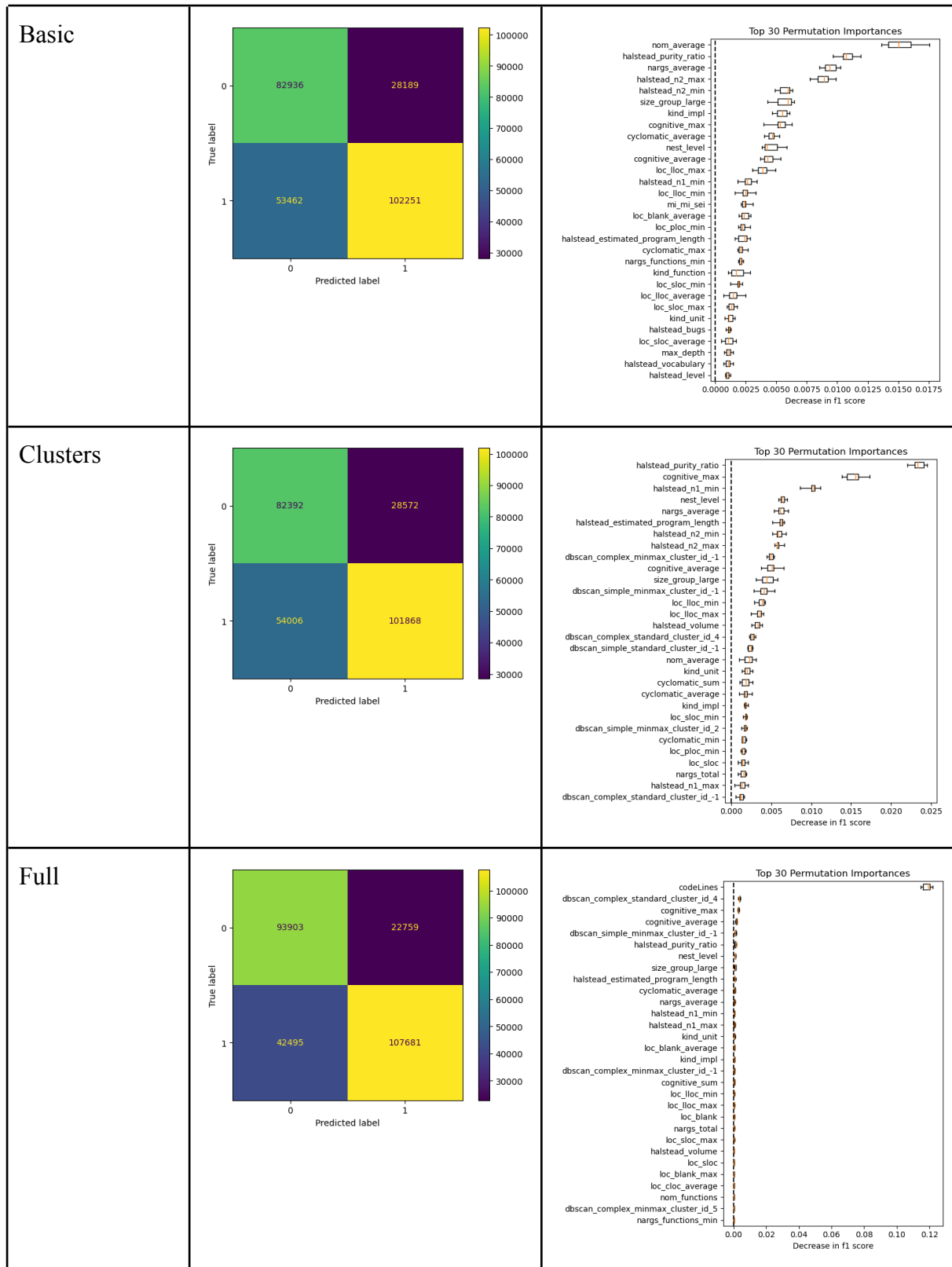
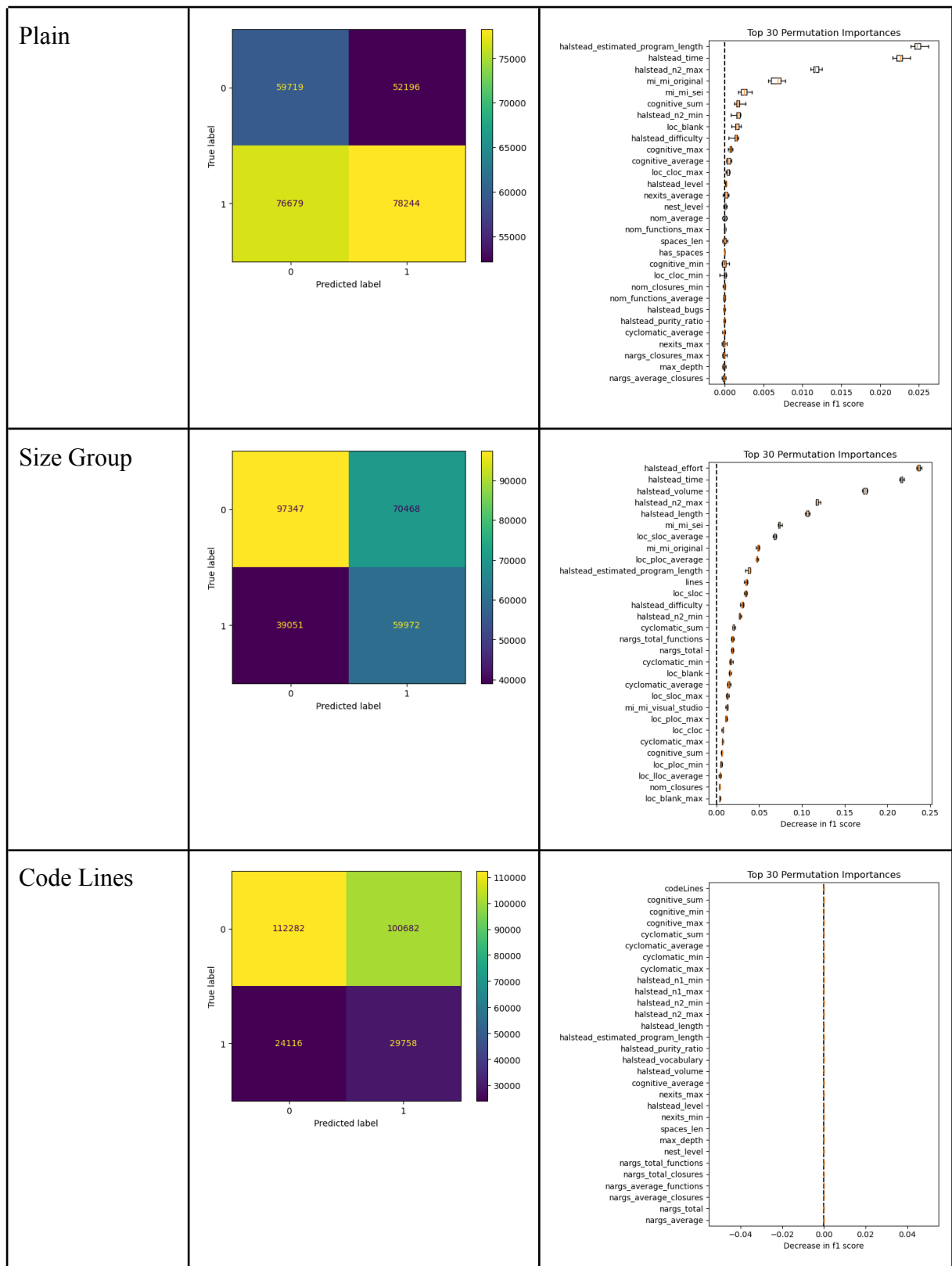


Figure. Confusion Matrix and Feature Importance for Bagging algorithm for all data configurations for the F1 metric.

Perceptron	Confusion Matrix	Feature Importance
------------	------------------	--------------------



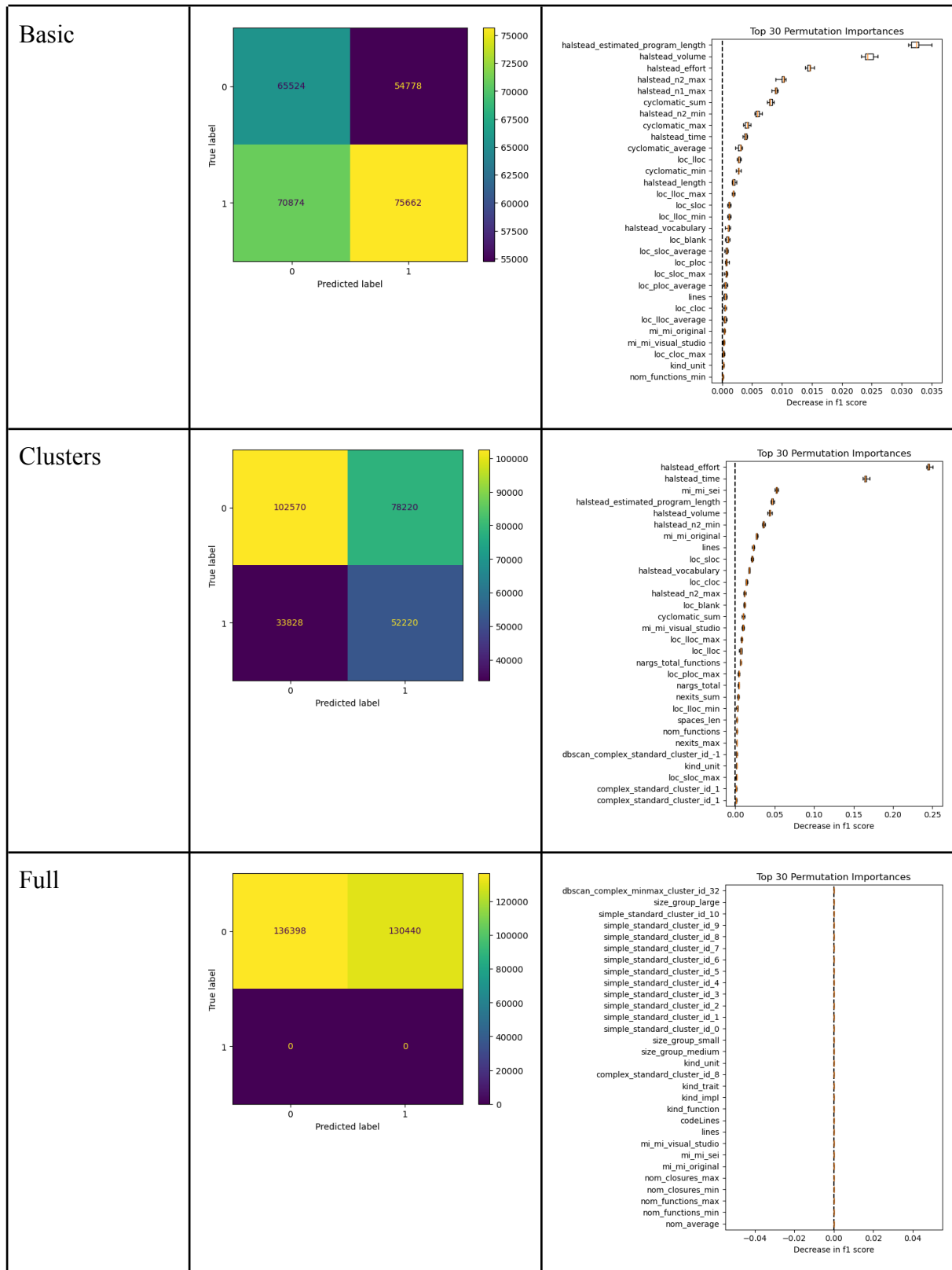
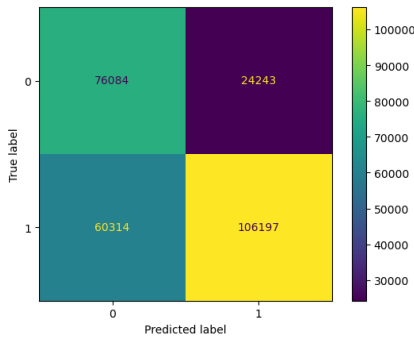
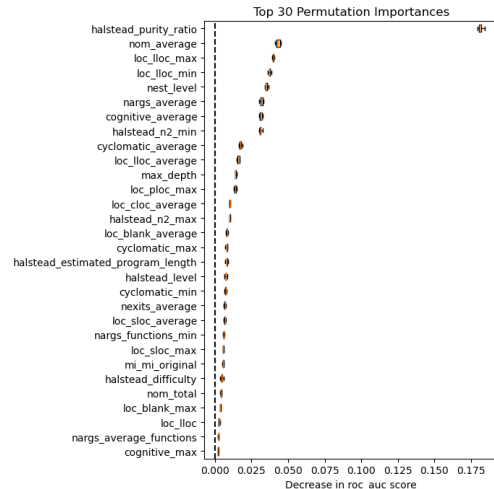
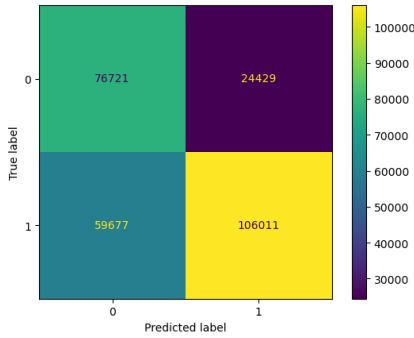
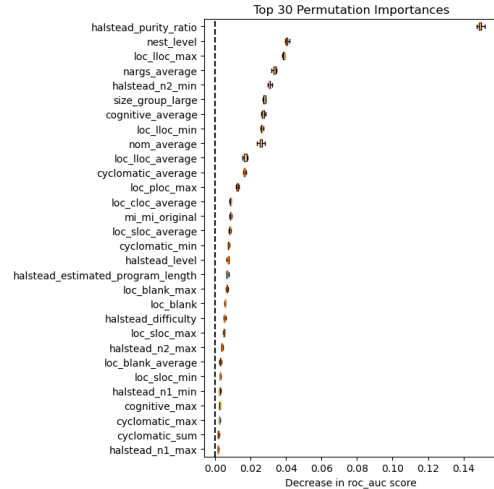
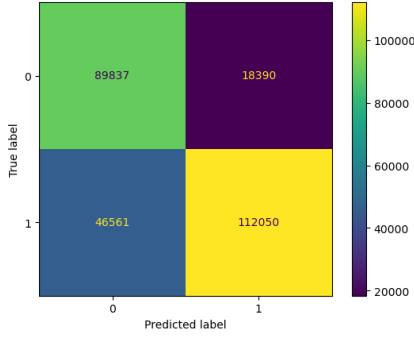
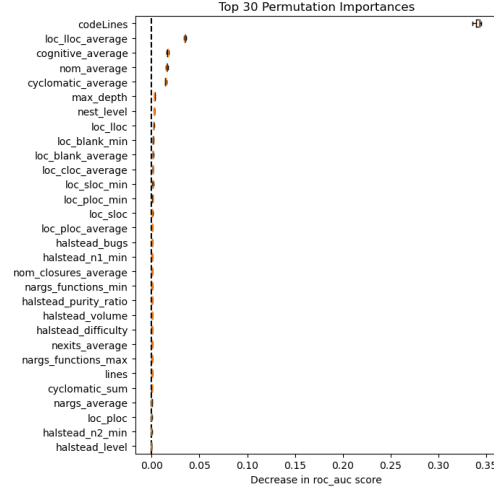


Figure. Confusion Matrix and Feature Importance for Perceptron algorithm for all data configurations for the F1 metric.

Area Under Curve Score Results

Decision Tree	Confusion Matrix	Feature Importance
Plain		
Size Group		
Code Lines		

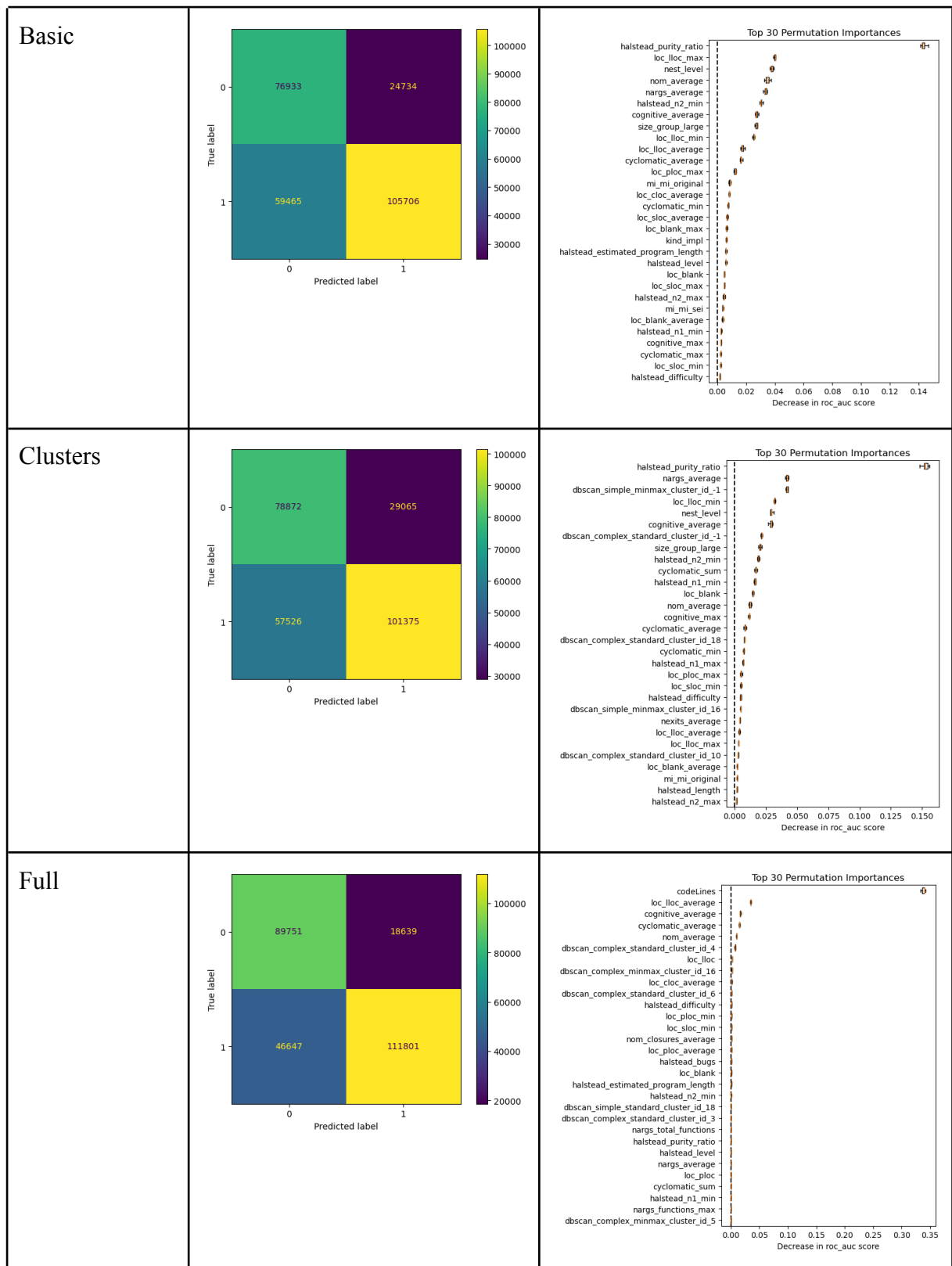
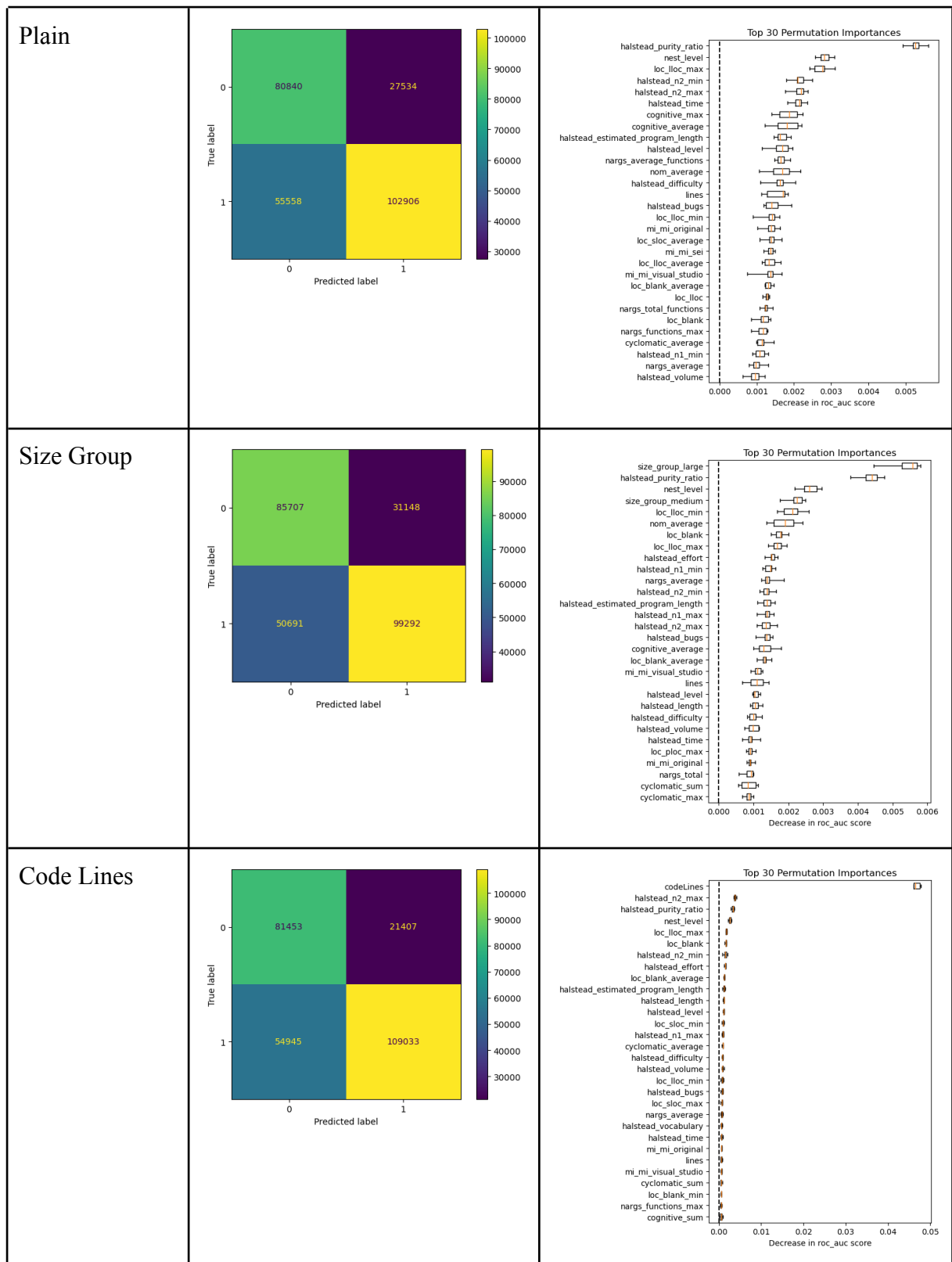


Figure. Confusion Matrix and Feature Importance for Decision Tree algorithm for all data configurations for the AUC metric.

Random Forest	Confusion Matrix	Feature Importance
---------------	------------------	--------------------



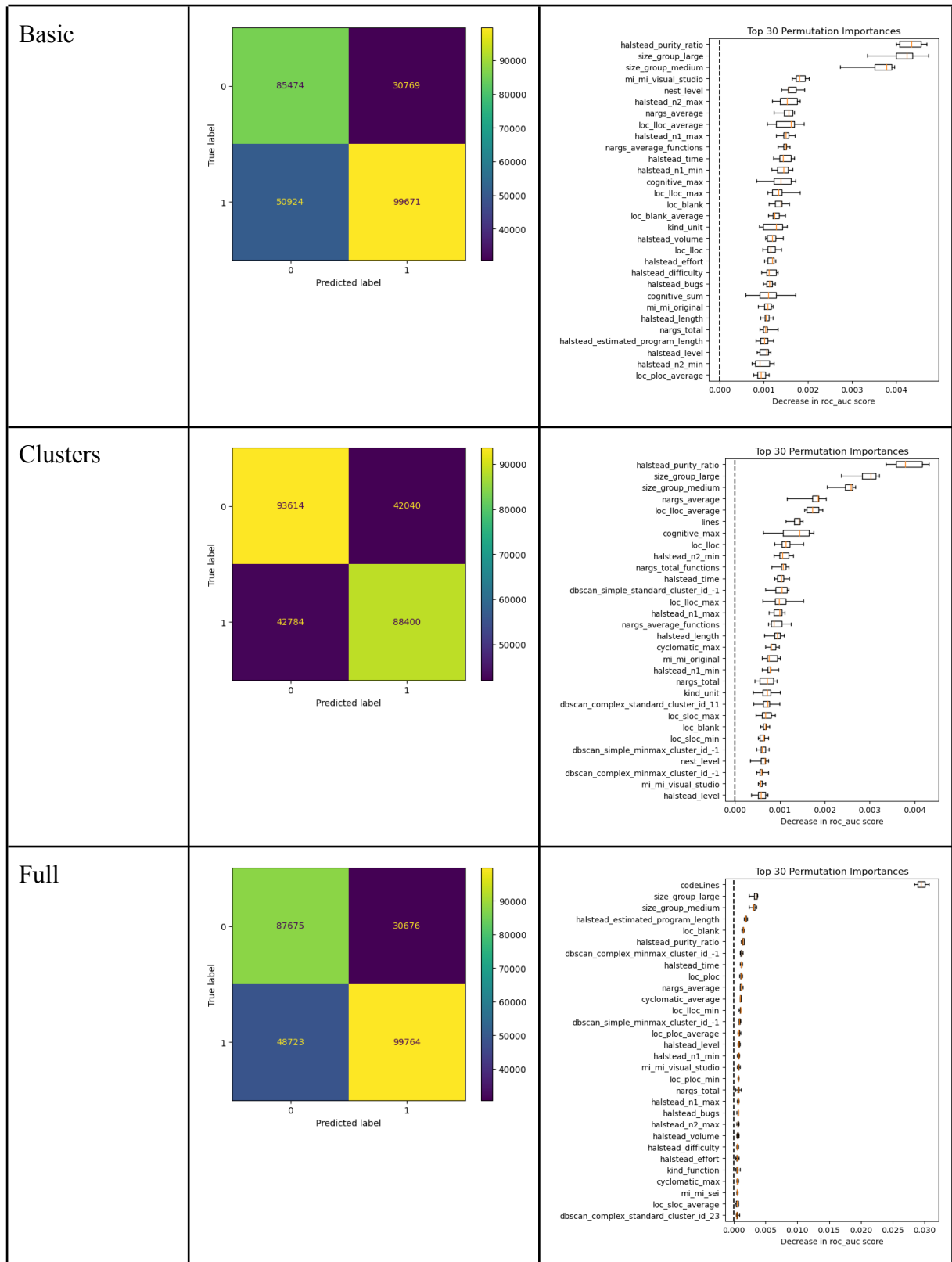
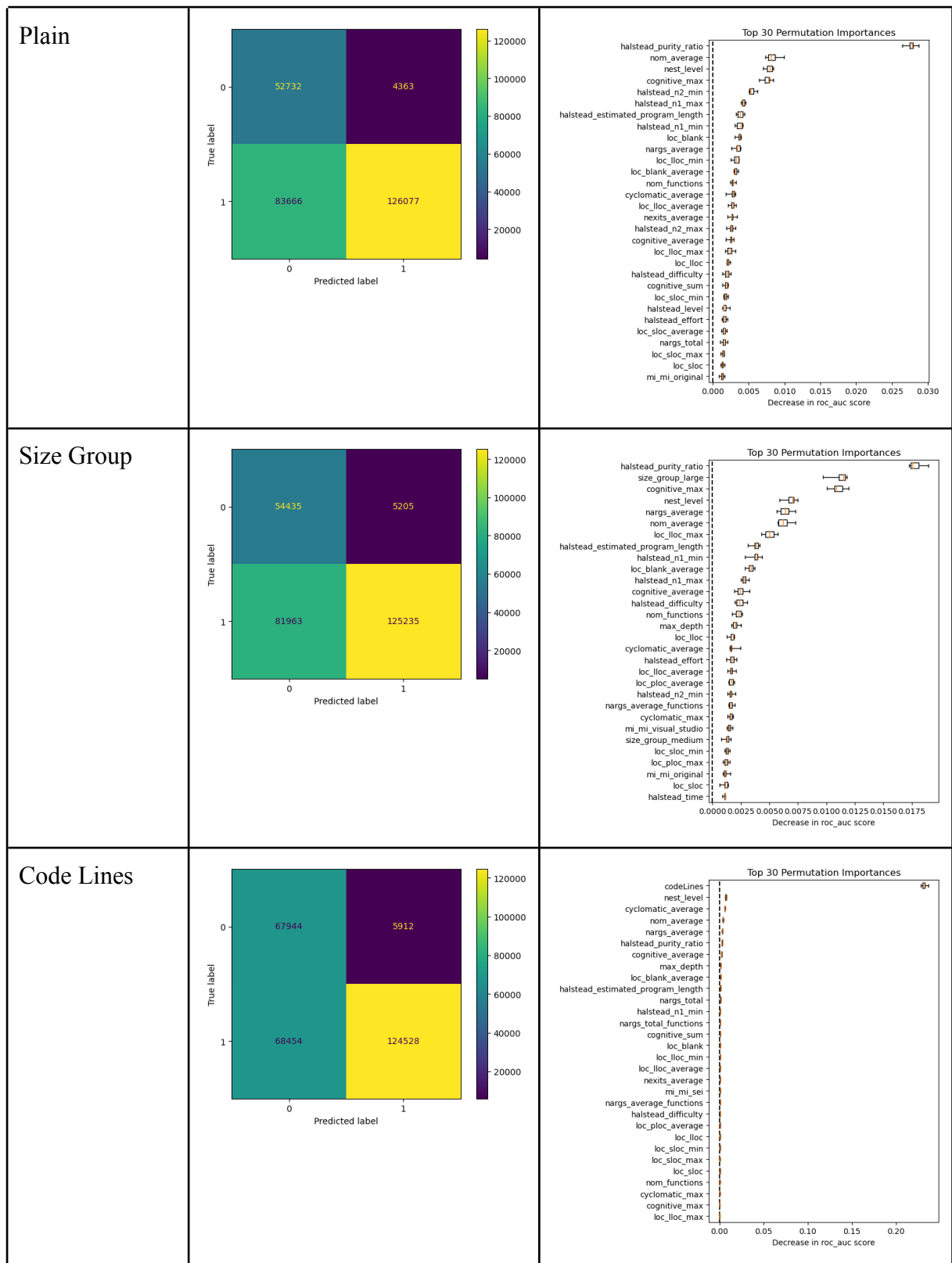


Figure. Confusion Matrix and Feature Importance for Random Forest algorithm for all data configurations for the AUC metric.

XGBoost	Confusion Matrix	Feature Importance
---------	------------------	--------------------



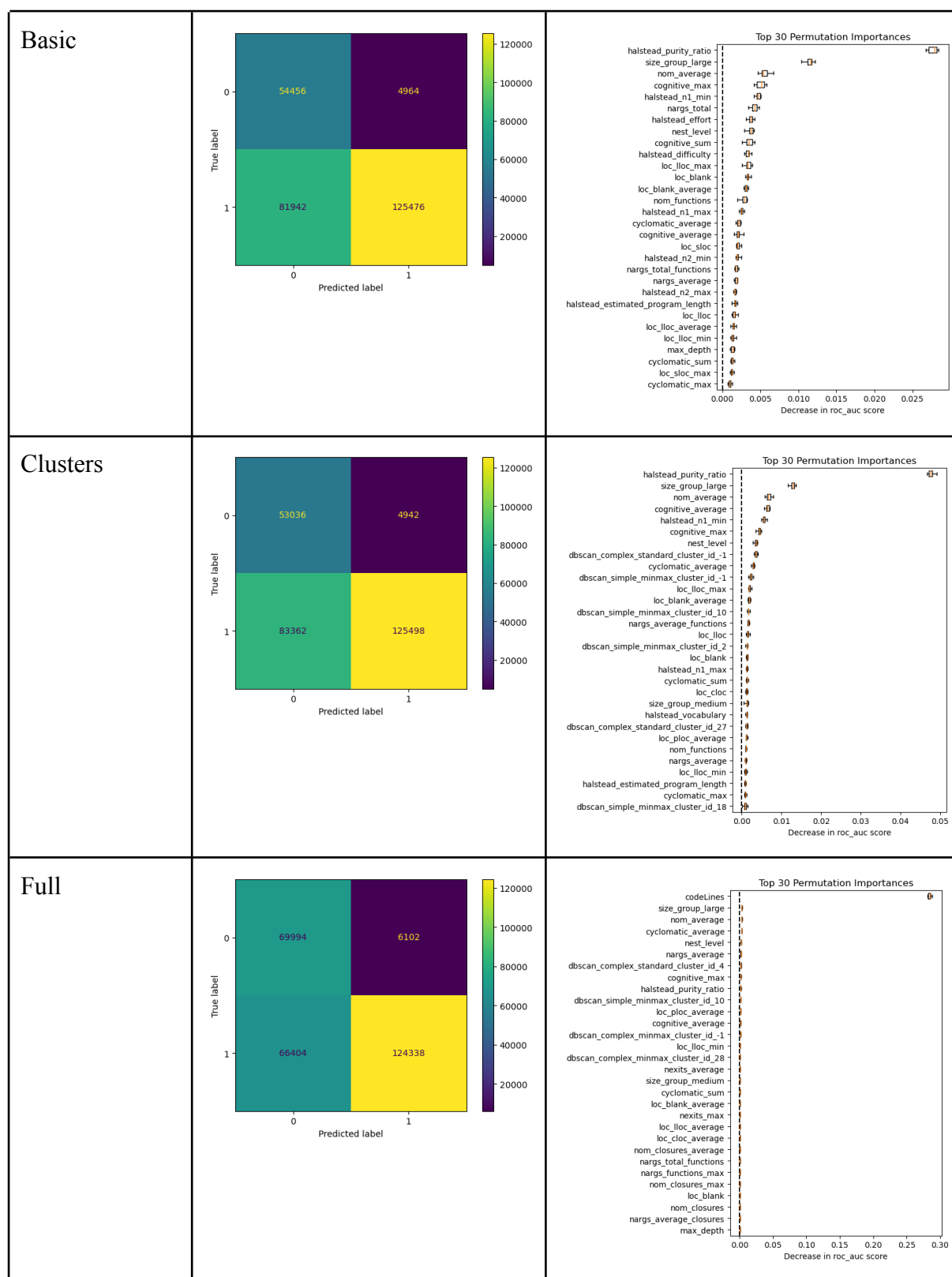
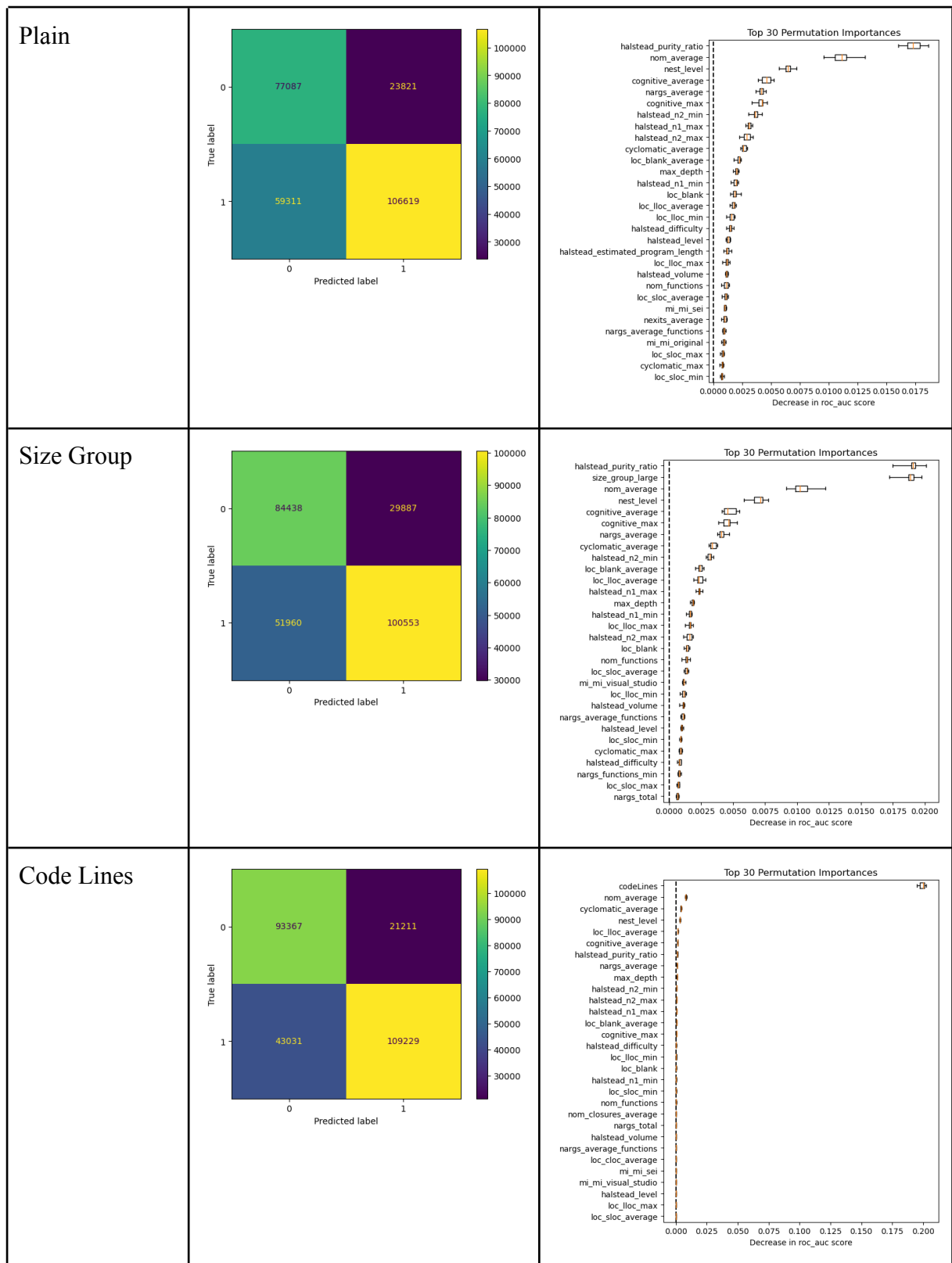


Figure. Confusion Matrix and Feature Importance for XGBoost algorithm for all data configurations for the AUC metric.

Bagging	Confusion Matrix	Feature Importance
---------	------------------	--------------------



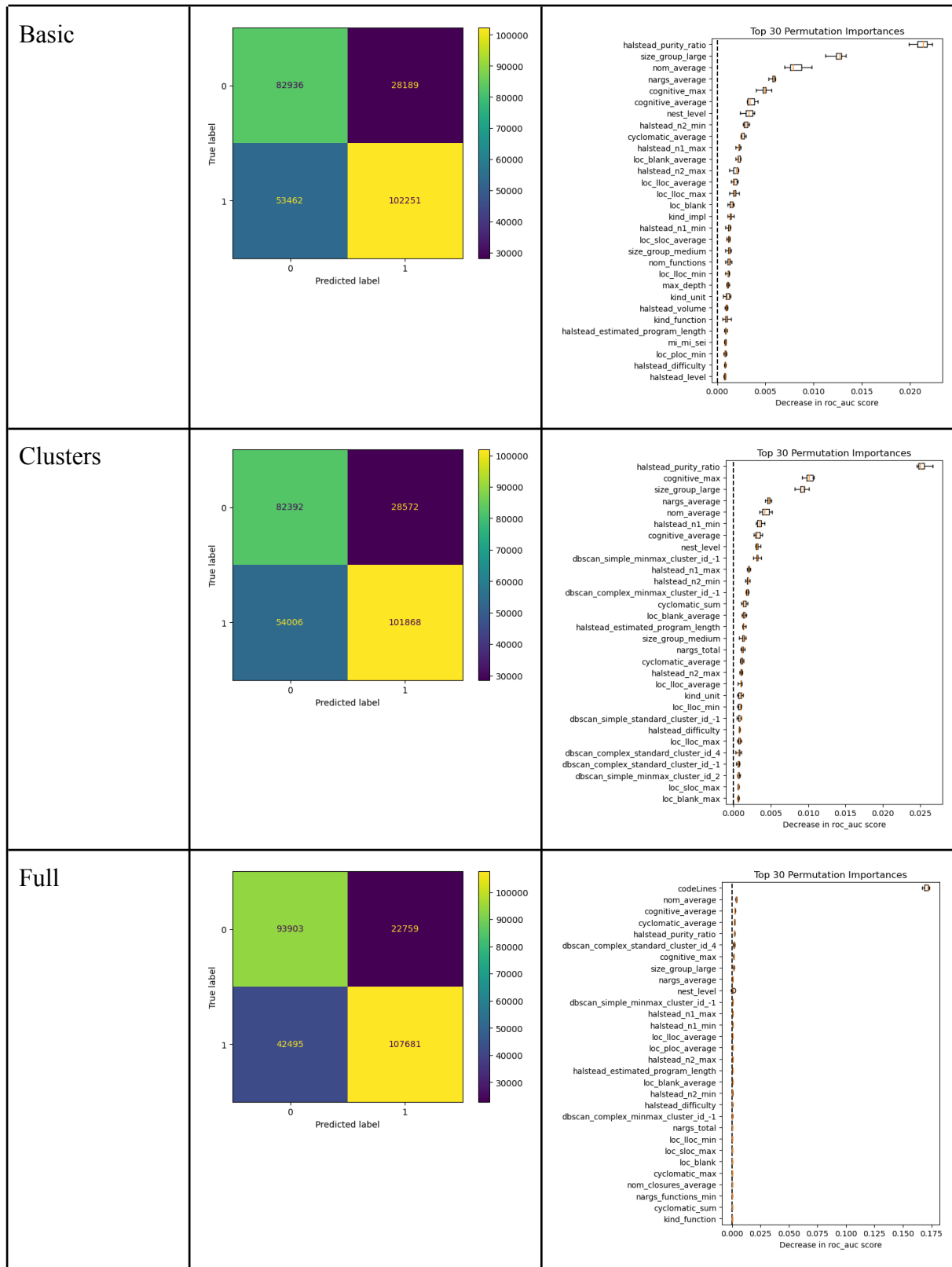


Figure. Confusion Matrix and Feature Importance for Bagging algorithm for all data configurations for the AUC metric.