NoSQL Injection Query Detection in MongoDB Using Supervised Learning-based Binary Classification Models

A Dissertation for
IMC 651 Project Work
Credits: 6
Submitted in partial fulfillment of Master's Degree in
M.Sc. (Integrated) in Data Science

By

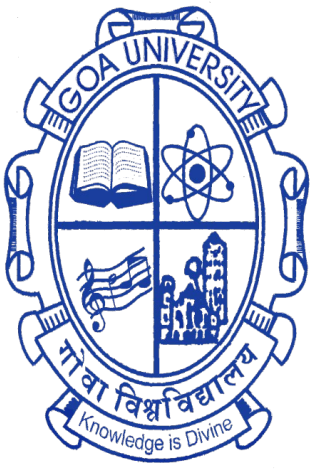
SHAUNAK SOMNATH PERNI
Seat No: (2105)
ABC ID: (678418387785)
PR No: (202100144)

MINAL NARESH SHIRODKAR
Seat No: (2108)
ABC ID: (706649472426)
PR No: (202100226)

Under the Supervision of

Shri. RAMDAS N. KARMALI

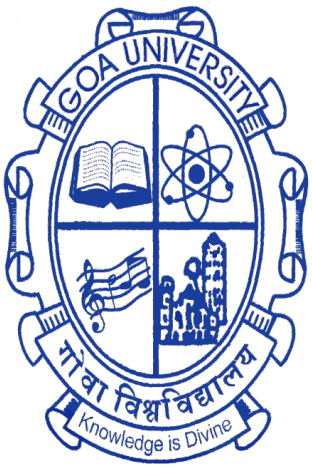
Goa Business School



GOA UNIVERSITY

APRIL 2024

GOA UNIVERSITY



GOA BUSINESS SCHOOL

CERTIFICATE OF EVALUATION

This is to certify that Mr. Shaunak Somnath Perni and Ms. Minal Naresh Shiroadkar have been evaluated for the project work titled **"NoSQL Injection Query Detection in MongoDB Using Supervised Learning-based Binary Classification Models**" in partial fulfillment of Master's Degree in M.Sc. (Integrated) in Data Science

(Examiner 1) (Examiner 2)

Place: Goa University
Date: 8th May 2024 (Dean, Goa Business School)

COMPLETION CERTIFICATE

This is to certify that the dissertation report "NoSQL Injection Query Detection in MongoDB Using Supervised Learning-based Binary Classification Models" is a bonafide work carried out by **Mr Shaunak Somnath Perni** and **Ms Minal Naresh Shirodkar** under my supervision in partial fulfillment of the requirements for the award of the degree of **M.Sc Integrated (Data Science)** in the Computer Science and Technology at the Goa Business School, Goa University.

(Shri. Ramdas N. Karmali)
Date:

(Prof. Jyoti D. Pawar) Stamp
Date:
Place: Goa University

DECLARATION BY STUDENTS

We hereby declare that the data presented in this Dissertation report entitled, "NoSQL Injection Query Detection in MongoDB Using Supervised Learning-based Binary Classification Models" is based on the results of investigations carried out by us in Computer Science and Technology at the Goa Business School, Goa University under the Supervision of Shri. Ramdas N. Karmali and the same has not been submitted elsewhere for the award of a degree or diploma by me. Further, We understand that Goa University or its authorities will not be responsible for the correctness of observations / experimental or other findings given the dissertation.

We hereby authorize the University authorities to upload this dissertation on the dissertation repository or anywhere else as the UGC regulations demand and make it available to any one as needed.

(Shaunak Somnath Perni)
Seat No: 2105

(Minal Naresh Shirodkar)
Seat No: 2108

Date:
Place: Goa University

Acknowledgments

We extend our heartfelt gratitude to the individuals whose contributions were instrumental in the completion of this research project.

We are deeply indebted to Mr. Ramdas Karmali for his invaluable guidance, unwavering support, and scholarly mentorship throughout the duration of this research project. His expertise and encouragement have been instrumental in shaping the direction and outcomes of our study.

Furthermore, we extend our thanks to the following professors, Mr. Hanumant Redkar, Mr. S. Baskar, Ms. Pradanya Bhagat, Mr. Swapnil Fadte and Mr. Jarret Fernandes, for their collaborative spirit, technical assistance, and constructive feedback. Their mentorship and guidance have been immensely beneficial in navigating the complexities of our project journey as students. We also express our gratitude to all those who provided support, encouragement, and understanding during the course of this endeavor.

Without the collective efforts and contributions of these individuals, this project would not have been possible.

Thank you.

Table of Contents

Index of Tables

Table of Figures

Terminology Used

Entity	Abbreviation
Machine Learning	ML
ML Operations	MLOps
Database	DB
Structured Query Language	SQL
Not Only SQL	NoSQL
Javascript Object Notation	JSON
Kernel Density Estimation	KDE
Support Vector Machine	SVM
[a] Fast and Lightweight AutoML Library	FLAML
Principle Component Analysis	PCA
Linear Discriminant Analysis	LDA
t-distributed Stochastic Neighbor Embedding	t-SNE

Abstract

The project built upon work done in developing machine learning models in the Cybersecurity field of injection query attack detection. Specifically, the project attempted to create a classification model that was trained on a Log file of a MongoDB database that had been attacked and attempted to identify queries that were malicious injection attack queries that allowed an unauthorized operation on the database system or were benign.

The project collected data from a local MongoDB system where a mix of injection queries and benign queries were sent to it. The system then produced a log file on which an analysis was conducted. The log file was converted to a tabular format and significant variables were identified and used to construct a training dataset.

Several configurations of the dataset were made i.e. subsets of the dataset where some variables were omitted/added either to reflect the dataset used in previous works or to test our hypothetical dataset configurations, 9 models were trained on each and their performances were recorded. A MLOps library was also used to identify the best model for each dataset.

A comparative study was then done with average model performance per dataset but it was concluded that there was no significant difference between the Top 3 datasets' average model performance. However, a difference was noticed in each of the Top models created for each dataset. Each one of the 3 datasets had an accuracy of 84% and a sensitivity of 88%. However, 2 of the 3 datasets scored higher on precision and F1 scores with them being 88% and 84.5% respectively.

1 Introduction

1.1 Database and Their Roles

In today's digital era, databases serve as the backbone of countless applications, holding vast amounts of valuable information. These information assets were critical for businesses, educational institutions, government agencies, and various other entities to manage and utilize effectively. Databases facilitated the storage, retrieval, and manipulation of structured and unstructured data, enabling seamless operations and informed decision-making.[1].

1.1.1 Informational Assets

The data stored within databases were often regarded as invaluable assets, representing the lifeblood of modern organizations. From customer profiles and transaction histories to inventory records and research findings, databases housed a wealth of information crucial for day-to-day operations and strategic planning. Protecting these assets was paramount, as any compromise could lead to severe consequences such as financial loss, reputational damage, and legal repercussions.

With the proliferation of online services and digital transactions, databases have become central repositories for sensitive data, including personally identifiable information (PII), financial records, and proprietary business data. Consequently, safeguarding databases against security threats was of utmost importance to preserve the confidentiality, integrity, and availability of these information assets.

1.2 SQL

Structured Query Language (SQL) stood as the cornerstone of relational database management systems (RDBMS), providing a standardized language for querying, updating, and managing relational databases. Developed in the 1970s, SQL evolved into a powerful tool for interacting with databases, offering a rich set of syntax and commands for performing a wide range of operations.

Over the decades, SQL underwent significant advancements and refinements, with various implementations and extensions tailored to specific database platforms. The development of SQL was driven by the growing demands of businesses and organizations for efficient data management solutions, leading to the introduction of features such as stored procedures, triggers, and user-defined functions. [2], [3].

1.2.1 Limitations

Despite its widespread adoption and versatility, SQL is not without its limitations. One notable challenge is its rigid schema structure, which necessitates predefined table structures and relationships. This inherent inflexibility can pose challenges in scenarios where the data model is dynamic or schema-less, hindering the scalability and adaptability of database systems[4].

Moreover, SQL-based databases may struggle to handle the scale and complexity of modern data requirements, particularly in scenarios involving unstructured or semi-structured data types. As organizations grapple with the influx of big data and the need for real-time analytics, the limitations of traditional SQL databases become increasingly apparent, paving the way for the emergence of alternative approaches such as NoSQL[5]

1.3 NoSQL

The rise of big data, coupled with the proliferation of web and cloud-based applications, has fueled the need for database systems capable of handling unprecedented volumes of data and supporting high-performance, distributed architectures. In response to these evolving requirements, NoSQL (Not Only SQL) databases have emerged as a compelling alternative to traditional SQL-based systems[5].

NoSQL databases depart from the relational model of SQL databases, offering a schema-less or flexible schema approach that accommodates diverse data types and structures. This flexibility is particularly well-suited for applications with dynamic or rapidly changing data schemas, such as social media platforms, IoT (Internet of Things) devices, and real-time analytics systems.[6]

The development of NoSQL databases has been driven by the desire to overcome the scalability and performance limitations inherent in SQL databases, especially when dealing with massive datasets and high-concurrency workloads. By adopting distributed architectures, horizontal scaling, and optimized data storage models, NoSQL databases offer superior scalability, fault tolerance, and performance compared to their SQL counterparts.

1.3.1 Types

NoSQL databases encompass a diverse range of data models and storage technologies, each optimized for specific use cases and data processing requirements. Common types of NoSQL databases include:

- Document-oriented databases: Store and retrieve data in the form of semi-structured documents, typically using JSON or BSON formats. Examples include MongoDB, Couchbase, and CouchDB.
- Key-value stores: Store data as key-value pairs, providing fast and efficient access to individual records. Examples include Redis, Amazon DynamoDB, and Apache Cassandra.
- Column-family stores: Organize data into columns rather than rows, enabling efficient storage and retrieval of sparse, wide-column datasets. Examples include Apache HBase, Apache Cassandra, and ScyllaDB.
- Graph databases: Model data as nodes, edges, and properties, allowing for the representation and traversal of complex relationships between entities. Examples include Neo4j, Amazon Neptune, and TigerGraph.[7]

1.4 Injection Attacks

Injection attacks represent a class of security threats that exploit vulnerabilities in software applications to execute malicious code or commands. These attacks typically involve the insertion of unauthorized or malicious input data into an application's input fields or parameters, with the intent of manipulating its behavior or compromising its security[8], [9], [4], [10].

1.4.1 Vulnerabilities

Injection attacks exploit weaknesses in input validation and sanitization mechanisms, allowing attackers to bypass security controls and interact with databases or execute arbitrary commands on the underlying systems Common types of injection attacks include SQL injection (SQLi), NoSQL injection (NoSQLi), command injection, and LDAP injection, among others.

1.4.2 Cost and Damage

The consequences of injection attacks can be severe, ranging from unauthorized access to sensitive data and data manipulation to system compromise and service disruption. In addition to the immediate impact on affected systems and users, injection attacks can also have long-term repercussions, including financial losses, reputational damage, and regulatory penalties.

1.5 Traditional Security Methods

Traditional security methods for SQL and NoSQL databases typically revolve around implementing a combination of preventive and detective measures to safeguard against potential vulnerabilities and attacks. These methods include:

- Input validation: Ensuring that user input is properly validated and sanitized to prevent injection attacks. This involves validating input data against predefined rules and sanitizing potentially dangerous characters or commands.
- Parameterized queries: Using parameterized queries or prepared statements to separate SQL code from user input, thereby reducing the risk of SQL injection attacks.
- Authentication and authorization: Implementing robust authentication and authorization mechanisms to control access to database resources and prevent unauthorized users from performing malicious actions.
- Database encryption: Encrypting sensitive data at rest and in transit to protect it from unauthorized access or interception by attackers.
- Security patches and updates: Regularly applying security patches and updates to database software and underlying operating systems to address known vulnerabilities and security flaws[10], [11], [12].

But even with such measures, attacks were still experienced as recently as October 2023.

[13], [14], [15], [16]

Month and Year	Target Organization/Service	Estimated Damage
October 2023	ICMR	$80,000
October 2023	23andMe	$54,000,000
May 2023	MoveIT	$10,000,000,000

Table 1.5.1: Selected Recent SQL Injection Attacks

As observed, even with these measures, major injection attacks were still causing a devastating amount of financial damage.

Hence, a motivation to use reactive models had been made, i.e., measures that could be trained and automatically discover patterns in injection attacks rather than be mitigated manually. Such types of measures are usually machine learning models or models that can be trained against a known list of attack queries to identify them.

1.6 Using Trained Models

The increasing sophistication and diversity of cyber threats, including injection attacks targeting both SQL and NoSQL databases, necessitate the adoption of advanced detection and prevention techniques. Machine learning models offer a promising approach to enhancing database security by leveraging the power of data-driven analytics and pattern recognition to identify and mitigate potential threats.

Incorporating trained machine learning models into database security frameworks involves several key steps:

- Data collection and preprocessing: Gathering relevant datasets containing examples of benign and malicious database queries or interactions. Preprocessing involves cleaning, transforming, and encoding the data to prepare it for model training.
- Feature engineering: Extracting informative features from the input data to capture relevant patterns and characteristics associated with benign and malicious behavior. These features may include query syntax, parameter values, user context, and temporal or spatial patterns.
- Model training: Selecting appropriate machine learning algorithms and train them on the labeled dataset to learn the underlying patterns and relationships between input features and target labels (i.e., benign or malicious). Common algorithms for classification tasks include logistic regression, decision trees, support vector machines, and neural networks.
- Model evaluation and validation: Assessing the performance of trained models using metrics such as accuracy, precision, recall, and F1-score. Validation techniques such as cross-validation or holdout validation help ensure the generalization and robustness of the models to unseen data.
- Integration and deployment: Incorporating trained models into existing database security systems or intrusion detection frameworks to enable real-time monitoring and analysis of database activity. This may involve deploying models as standalone components or integrating them with database management systems (DBMS) or network security appliances.

Trained Machine learning may provide a higher performance in militating, detecting or even protecting systems from such attacks

2 Literature Survey

2.1 Overview of SQL and NoSQL Injection Attacks

SQL injection is one of the most serious security vulnerabilities in web applications that use relational databases. As the use of web applications and online services has increased significantly over the past two decades, addressing SQL injection vulnerabilities has become paramount. SQL injection attacks can allow unauthorized access or modification of data stored in databases, compromising the integrity and confidentiality of sensitive information [10].

With the rise of NoSQL databases, such as MongoDB, for handling unstructured data and meeting the demands of modern web applications, new security challenges have emerged. Several studies [6, 12, 24, 25, 26, 27] have demonstrated that NoSQL databases are vulnerable to injection attacks, similar to traditional SQL injection attacks on relational databases. These attacks can potentially lead to unauthorized access and manipulation of sensitive data stored in NoSQL databases.

2.2 Existing Detection and Prevention Techniques

Researchers have explored various techniques for detecting and preventing both SQL and NoSQL injection attacks, ranging from traditional methods to advanced machine learning and deep learning approaches.

2.2.1 Traditional Techniques

Several detection and prevention techniques have been proposed to address SQL injection vulnerabilities. The paper by the anonymous authors [11] presents a comprehensive literature survey on SQL injection detection and prevention techniques. They emphasize the importance of securing data stored in databases, particularly in the era of cloud computing, where data storage and access are widely distributed.

Rua et al. [17] investigate the impact of poor input validation on the security of server databases. They propose a technique called CombinedDetect, which combines JavaScript and PHP coding to detect and isolate malicious SQL queries before sending them to the server. Their study highlights the risks posed by SQL injection attacks and the importance of effective input validation and multiple detection methods.

For NoSQL injection attacks, studies [29, 30] have proposed traditional techniques involving analyzing the structure of NoSQL queries, comparing the intended query structure with the runtime query structure, and employing input validation techniques. While these techniques have shown promise, they may struggle to detect unknown or zero-day attacks effectively.

2.2.2 Machine Learning and Deep Learning Techniques

With the limitations of traditional signature-based detection methods, there has been growing interest in applying machine learning techniques for detecting SQL injection attacks, including unknown or zero-day attacks [18, 19, 20, 21].

Hasan et al. [18] propose a machine learning-based heuristic algorithm for SQL injection detection. They train and test 23 different machine learning classifiers on a dataset of SQL statements, selecting the top five classifiers based on their detection accuracy and developing a GUI application using these classifiers.

Hosam et al. [19] focus on machine learning techniques for SQL injection detection, defining 13 relevant features that can be extracted from user inputs and evaluating six different machine learning algorithms. Their models achieve an accuracy of up to 99.6% and demonstrate the ability to generalize well to unseen data.

For NoSQL injection attacks, researchers have explored the use of machine learning and deep learning models. Studies such as [8, 9, 22, 23, 28] have employed various machine learning and deep learning models, including Support Vector Machines (SVMs), Random Forests, Neural Networks, and Deep Residual Networks (ResNets), for detecting NoSQL injection attacks. These models have achieved promising results, with reported accuracies ranging from 91.5% to 99.6%.

2.2.3 Deep Learning Techniques

In addition to traditional machine learning methods, deep learning techniques have also been explored for SQL injection detection. Sangeeta et al. [22] propose a method for SQL injection attack detection using ResNet, a deep residual neural network architecture. Their approach tokenizes and vectorizes input queries, which are then trained using the ResNet algorithm. The authors demonstrate that their ResNet-based model can effectively identify different types of SQL injection attacks.

2.2.4 Datasets and Tools

To facilitate the development and evaluation of SQL and NoSQL injection detection models, researchers have created datasets and tools. The MongoDB Injection Dataset [31] is a comprehensive collection of MongoDB NoSQL injection attempts and vulnerabilities, which can be used for training and testing machine learning models.

2.3 Our Approach

The proposed approach leverages supervised learning-based binary classification models, which can distinguish between benign and malicious NoSQL queries.

By utilizing log files from MongoDB servers, the research aims to create a realistic and representative dataset for training and evaluating the proposed model. This approach addresses the need for practical and scalable solutions for detecting NoSQL injection attacks in real-world scenarios.

The performance of the proposed model will be compared with previous works, particularly those employing machine learning and deep learning techniques for SQL and NoSQL injection detection. This comparison will provide insights into the effectiveness of the proposed approach and its potential contributions to the field of database security, encompassing both relational and NoSQL databases.
Through this research, the authors aim to develop a robust and efficient model for detecting NoSQL injection attacks on MongoDB servers, contributing to the ongoing efforts to enhance the security of databases and protect sensitive data from unauthorized access and manipulation.

3 Aims and Objectives

3.1 Research Questions

The research questions this project attempted to answer were:

1. What are the significant variables to discriminate a injection query from a benign query from a MongoDB log file?
2. Is it possible to create a model to classify MongoDB queries as malicious or benign based on training data extracted from log files?
3. In continuation with previous works where models were trained with only raw query text adding more variables to the dataset bring any significant change in the performance of a random model trained on such data?

3.2 Problem Statements

The problem statements were as follows:

1. “Conduct a study of a MongoDB log file to discover significant features that can be used to discriminate between a injection query and a benign query”
2. “Construct a Model to classify if a given MongoDB NoSQL query is a malicious injection query that may execute an unauthorized operation on the server, or is it a benign query”
3. “Conduct a Comparative Performance study of Classification Models trained in different configurations of a dataset that has been processed from a MongoDB log file”

3.3 Project Stages

The project took place in the following stages:

1. **Data Collection:**
   Data on MongoDB log files was collected
2. **Data Preprocessing:**
   The same collected data was cleaned for errors and missing data, integrated from the various, and transformed into a common structured form for processing
3. **Data Exploration:**
   The Data variables were explored, and a descriptive analysis and significance tests were conducted. A Separability analysis was done to determine which model algorithms can be used on the dataset. A final training dataset was formed
4. **Model Formulation:**
   Dataset configurations were created and used as training data for different sets of model algorithms and their performances recorded
5. **Evalutaion and Testing:**
   Model Performance reports were tested and evaluated and conclusions were formed

4 Data Collection

4.1 Initial Challenges

Due to the unique requirements of this project, no datasets related to MongoDB logs were found. Therefore, a labeled public training dataset of MongoDB queries was utilized, where queries were labeled as Malicious and Benign via a boolean label.[31], using this dataset we conducted our analysis.

4.2 Setup for Data Collection

To collect log-based data, we set up an empty MongoDB server on a local machine. The database was intentionally left empty, with no collections or documents inserted, to study the effect on a standard server. The specifications of the local machine are as follows

Table 4.2.1: Hardware Details for running MongoDB server

Operating System	Fedora Linux 39 (Workstation Edition)
OS Type	64-Bit
Kernel-Version	6.5
Processor	AMD Ryzen™ 7 5700U with Radeon™ Graphics × 16
Memory	8.0 GiB
Disk	1TB(HDD) + 256GB (SSD)
MongoDB Version	6.0.15
Mongosh	2.25

The Database was initialized with the following setting

```
db.setProfilingLevel(2,0.1)
```

According to the MongoDB manual[32], this sets the profiler at the highest level and records all queries that take more than 0.1 microseconds to process. Subsequently, we fired 244 queries using a script provided by the dataset and collected the log files.

4.3 Data Collection Results

According to the MongoDB documentation, a standard log entry was a simple JSON object with specific attributes related to the Log and Query and query processing[32].

At default settings on our system, the log file was generated at the following file on our system

`/var/log/mongodb/mongod.log`

4.4 Pre-emptive Cleaning

Reading the generated log, we found several entries unrelated to the queries, but as part of MongoDB's system health checks and other miscellaneous entries.

MongoDB produces the following Log entry types based on which component's process was executed[32]:

Table 4.4.1: MongoDB Log Entry Codes

ACCESS	Messages related to access control and authentication.
COMMAND	Messages related to database command execution.
CONTROL	Messages related to control activities and initialization.
ELECTION	Messages specifically related to replica set elections.
FTDC	Messages related to diagnostic data collection, including server statistics.
GEO	Messages related to parsing and verification of geospatial shapes.
INDEX	Messages related to indexing operations, such as index creation.
INITSYNC	Messages related to initial sync operation in replica sets.
JOURNAL	Messages specifically related to storage journaling activities.
NETWORK	Messages related to network activities, such as connection acceptance.
QUERY	Messages related to queries and query planner activities.
RECOVERY	Messages related to storage recovery activities.
REPL	Messages related to replica sets, including initial sync and replication.
REPL_HB	Messages specifically related to replica set heartbeats.
ROLLBACK	Messages related to rollback operations in replication.
SHARDING	Messages related to sharding activities, such as mongos startup.
STORAGE	Messages related to storage activities, including fsync processes.

TXN	Messages related to multi-document transactions.
WRITE	Messages related to write operations, such as updates.
WT	Messages related to the WiredTiger storage engine.
WTBACKUP	Messages related to backup operations by WiredTiger.
WTCHKPT	Messages related to checkpoint operations by WiredTiger.
WTCMPCT	Messages related to compaction operations by WiredTiger.
WTEVICT	Messages related to eviction operations by WiredTiger.
WTHS	Messages related to WiredTiger's history store.
WTRECOV	Messages related to recovery operations by WiredTiger.
WTRTS	Messages related to rollback to stable operations by WiredTiger.
WTSLVG	Messages related to salvage operations by WiredTiger.
WTTIER	Messages related to tiered storage operations by WiredTiger.
WTTS	Messages related to timestamps used by WiredTiger.
WTTXN	Messages related to transactions performed by WiredTiger.
WTVRFY	Messages related to verification operations by WiredTiger.
WTWRTLOG	Messages related to logging write operations by WiredTiger.
Unnamed Components	Messages not associated with a named component, using the default log level specified in the system settings.

For the scope of this project, it was found that only COMMAND log entries are to be extracted from the log files as these entries contain log details of the sent queries.

4.5 Final Collected Dataset

The final collected dataset had the following variables in the JSON format

Table 4.5.1: MongoDB Log Entry Structure

```
{
 "t": <Datetime>, // timestamp
 "s": <String>, // severity
 "c": <String>, // component
 "id": <Integer>, // unique identifier
 "ctx": <String>, // context
 "msg": <String>, // message body
 "attr": <Object> // additional attributes (optional)
 "tags": <Array of strings> // tags (optional)
 "truncated": <Object> // truncation info (if truncated)
 "size": <Object> // original size of entry (if truncated)
}
```

[32]

5 Data Processing

5.1 Conversion to Tabular Data

Using the pandas library[33] we expanded and converted the data into a tabular format. However, we had to expand the data columns in several passes due to nested keys and values in the JSON structure.

The outcome of this process was to give us a tabular format of the raw JSON data that can be then processed by any processing algorithm

5.2 Cleaning

For each pass, we would also decide whether to omit or accept a variable for further passes. This is because we observed some variables to be constant since we tested the queries on an empty database.
These variables had to be omitted as they could not provide any insights since they were constant.

5.2.1 Pass 1

Table 5.2.1: Data Structure for Cleaning Pass 1

Variable Name	Data Type	Description
t	DateTime	Timestamp
s	string	Short severity code
c	string	Full Component String
id	string	Unique Identifier for the Log statement
ctx	string	The thread that caused the long statement
msg	string	Log output message
attr	Object	Key value pairs containing various attributes to the query sent
Tags	string[]	MongoDB tags attributed to the string
truncated	Object	Information if any attr key-value pair has been truncated
size	Object	Original Size of the log entry if it was truncated

5.2.2 Pass 1 - Filtering

Out of the 10 variables initially considered, only "Attr" (Attributes) was selected. The "Attr" column encompasses diverse information, including query type, targeted collection, sender's IP and port, query duration, and more.

The other variables were rejected due to having no relationship to the query sent and only to the log entry. Additionally, the "Timestamp" query was not considered as all queries were processed at the same time due to the script firing the queries at the database, making the variable a constant and not significant enough to be used.

5.2.3 Pass 2

Table 5.2.2: Data Structure for Open ATTR column

Variable Name	Data Type	Description[32]
ns	string	Namespace/Collection where the query is executed
command	Object	Parameter and type of Query sent to the database
remote	string	IP and port address of the sender
protocol	string	Protocol of the user
plan summary	string	What Plan was taken up by the system to execute the query
planningTimeMicro	int	The time it took to create a plan for the query in microseconds
keyExamined	int	Amount of keys examined
docsExamined	int	Amount of Documents Examined
nBatches	int	How many batches did it take for the operation
cursorExhaustred	bool	Unknown
numYields	int	Unknown
nreturned	int	Number of results returned
queryFramework	string	The framework used for the query
reslen	int	Unknown
Locks	object	Unknown
storage	object	Unknown

cpuNanos	int	CPU time to process query in nanoseconds
durationMills	int	Duration of the query in milliseconds

5.2.4 Pass 2 - Filtering

We decided to select only a few variables from this dataset as it was found that most variables were constant in the table due to the artificial environment from which the data was collected.
The variables that were decided to be taken due to having some degree of variance were:

- “command”: This column encapsulates crucial details of the query, such as the query type, targeted collection, and the database aimed at.
- “planningTimeMicro”: This was the Planning time taken by the system to execute the query
- “cpuNanos”: Similarly to planningTimeMicro, this represents the amount of time taken by the CPU of the system to process the query

5.2.5 Pass 3

Upon the expansion of the Command column, we found the following variables

Table 5.2.3: Data Structure for Opened Command Column

Variable Name	Data Type	Description
find	string	This is the collection argument to the "Find" query
filter	Object	The filter used for the "Find" query
lsid	Object	This contains the uuid of the query sent
$db	string	Target Database

Only the "filter" variable (now filter_str) was deemed essential since it was the only variable with some variance the rest of the variables were constant

5.3 Integration

Using the preprocessed data, the selected columns were now joined with the query dataset's "label" column. The join criteria were the "filter" column from our dataset and the "text" key from the query dataset.

5.4 Final Cleaned and Integrated Data

The final cleaned raw data is as follows

Table 5.4.1: Data Structure after Cleaning

Name	Data Type	Description
filter_str	Text	This is the extracted raw filter of the query sent
cpuNanos	Float	Time Taken by CPU to process Query
planningTimeMicros	Float	Time Taken for MongoDB to decide on a plan for the Query
Label	Boolean	The label set for queries 0 = Benign Query 1 = Malicious Query

6 Data Exploration

Using the cleaned dataset, we conducted a series of analyses on the data to gain insights. Specifically, we conducted the following types of analysis:

1. Descriptive Analysis:
   This involved analyzing the variables we have in the dataset. This includes calculations of descriptive statistical metrics, visualizations of the distributions, etc.
2. Feature Engineering
   We created new variables by using existing data to add more factors to the training dataset.
3. Significance Test
   This is where we tested the statistical significance of each variable when it comes to training for the classification models. Only the variables that significantly passed the tests were considered for the training dataset.
4. Separability Analysis
   This involved the analysis of the type of separability our data has. The outcome of this analysis affects the type of classification models we can use.

The outcome of this process was an output of a training dataset that can be used in any machine-learning classification task of MongoDB injection queries.

6.1 Descriptive Analysis

Individual variables were descriptively analyzed, considering the three types of data formats: Text, Boolean, and Numeric. The analysis was done differently for each of them to capture their unique characteristics and distributions.

6.1.1 Text

For this, the variable “Filter” was analyzed this includes:

1. Word Frequency Analysis
2. Visualization of the Word Frequencies

To conduct the word frequency analysis, a string tokenizer was used. However, due to the structure of MongoDB filters, a natural language tokenizer could not be used. Hence, for this analysis, a custom-made tokenizer was developed and used.

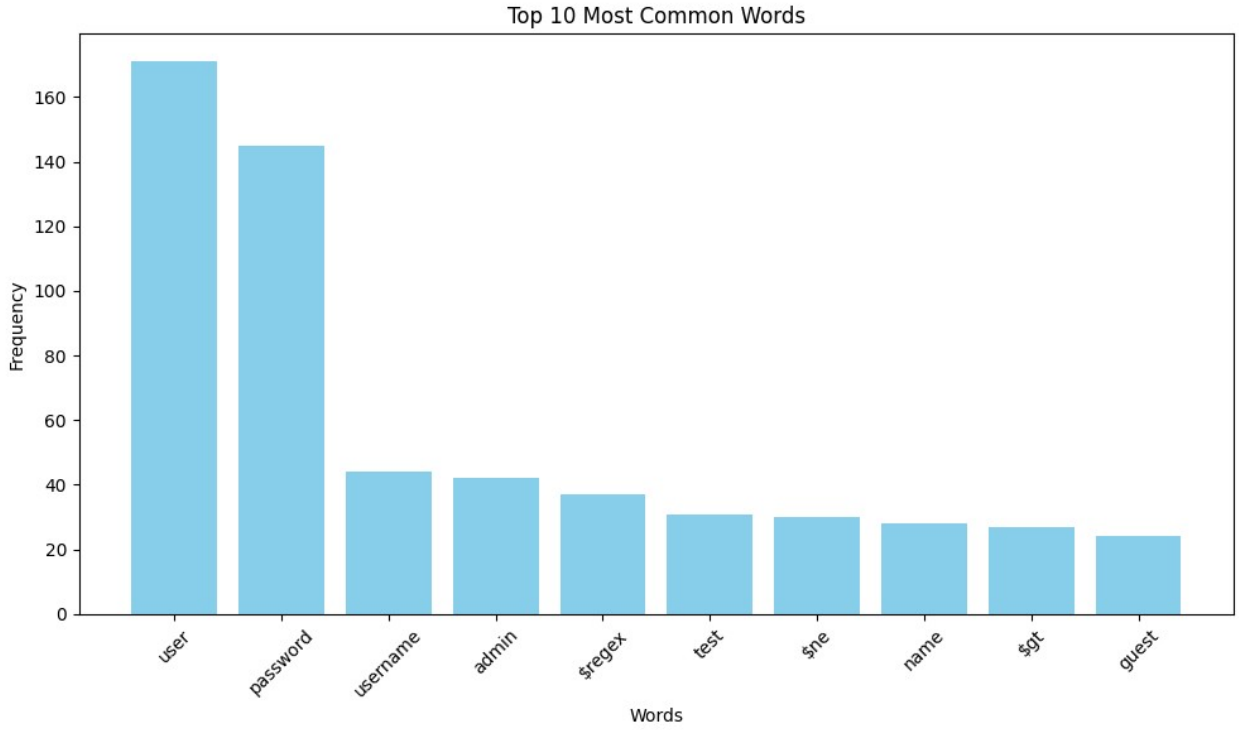


Figure 6.1.1: Top 10 Word Frequencies for Filter Variable

Table 6.1.1: Word Frequencies for Filter Variable

Word	Frequency
user	171
{}	18
password	145
username	44
$regex	37
^{}	4
$ne	30
$gt	27
hacker	19
&ne	4
alice	4
123456	12
^test	10
testuser	10
guest	24
john	11
qwerty	9
category	12
electronics	12
price	8
name	28
^S	4
selector	4
department	8
engineering	4
_id	4
1	4
John Doe	5
HR	4
salary	4
pass	6

admin	42
$username	1
$password	1
id	1
value	1
John	1
age	5
$where	4
this.age >= {}	1
test	31
type	8
roles	8
_admin	1
.*{}.*	1
$nin	1
$in	18
root	12
^{}$	4
$options	5
i	5
$mod	1
this.password.length > 10	1
$size	12
this.age <= {}	1
$type	6
this.password.length <= 5	1
Alice	1
{}|admin	1
password123	4
testuser123	1
Bob	1

6.1.2 Sampled Text

We also divided the dataset into 2 samples and conducted a descriptive analysis. They were divided according to the value of the "label" column, i.e., filter column rows for label = 0 and the same for label = 1. This allowed us to analyze the characteristics and frequencies of the data separately for each label category.

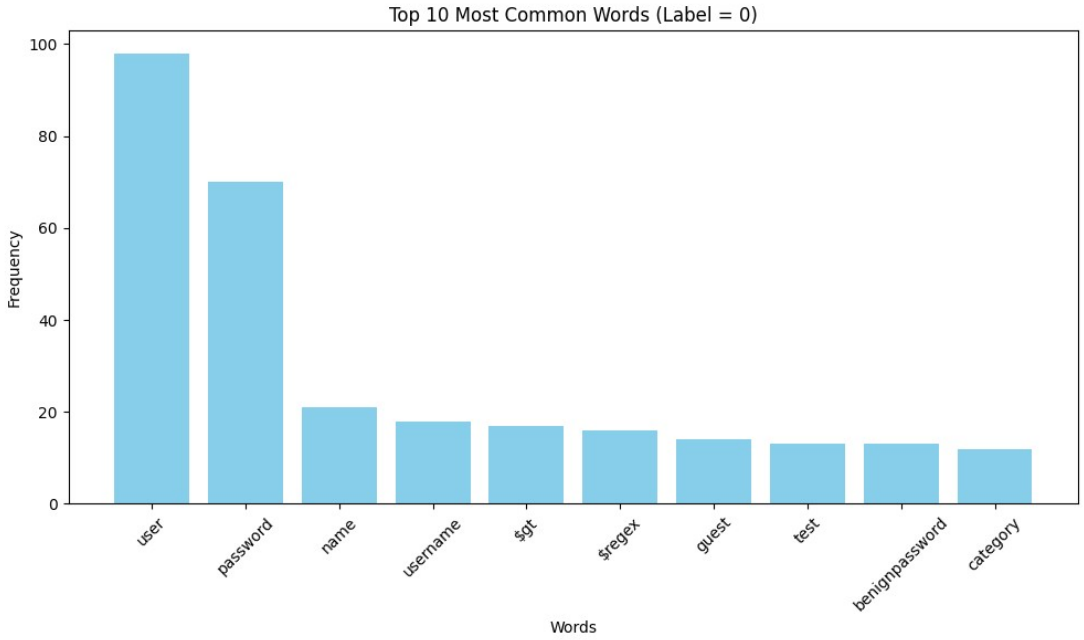


Figure 6.1.2: Top 10 Word Frequencies for Filter Variable where Label = 0

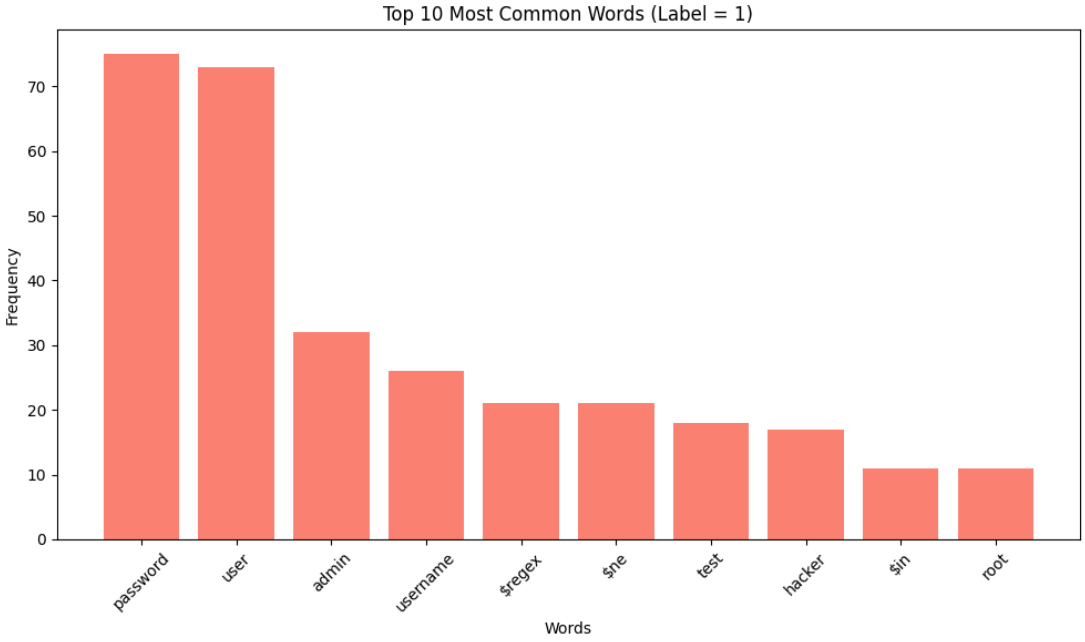


Figure 6.1.3: Top 10 Word Frequencies for Filter Variable where Label = 1

6.1.3 Numerical Features

This is the analysis of the Numerical Features, i.e., planningTimeMicros and cpuNanos. This includes:

1. Descriptive Statistical Metrics of the feature:
    a. Mean
    b. Median
    c. Standard deviation
    d. Minimum
    e. Maximum
    f. Quartiles
2. Visualization of the KDE plot for continuous features or Barplot for boolean:
    a. KDE (Kernel Density Estimate) plot for continuous features to visualize their distribution.

6.1.3.a planningTimeMicros

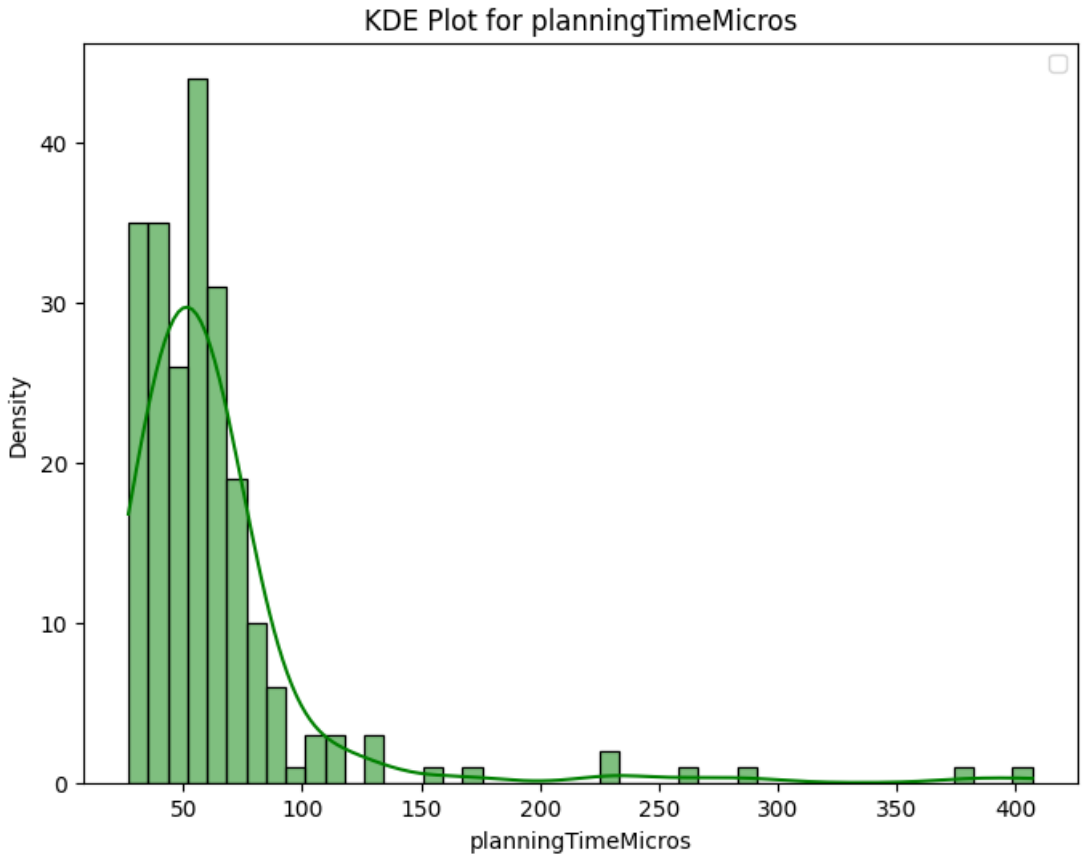


Figure 6.1.4: KDE Distribution plot of planningTimeMicros

Table 6.1.2: Descriptive Statistics for planningTimeMicros

Metric	**Value**
Count	244
Mean	99642.1696
Standard Deviation	44007.5328
Min	49784
25%	71093
50%	88842
75%	114685.25
Max	378420.0

6.1.3.b cpuNanos

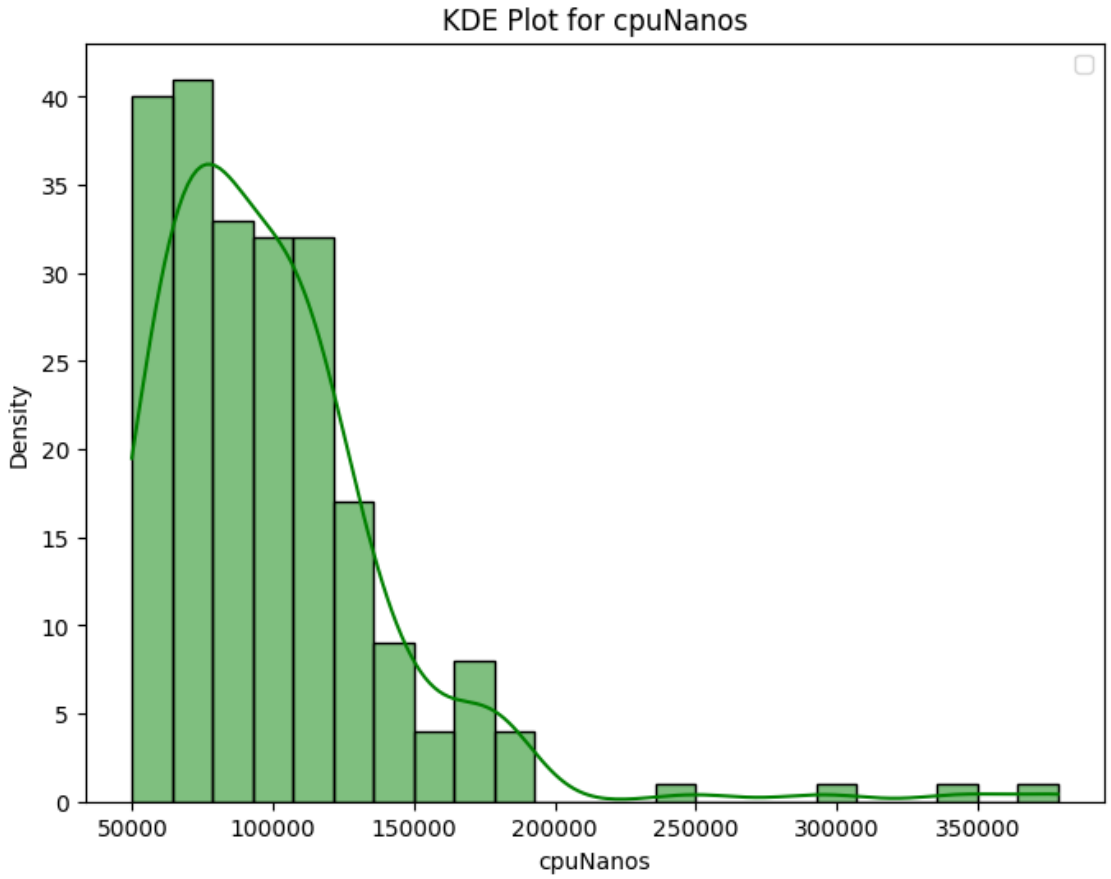


Figure 6.1.5: KDE Distribution plot of cpuNanos

Table 6.1.3.a.1: Descriptive Statistics of cpuNanos

Metric	**Value**
Count	244
Mean	63.125
Standard Deviation	46.78816
Min	27.0
25%	41.75
50%	55.5
75%	67.0
Max	407.0

6.1.4 Target Variable

6.1.4.a Label

Figure 6.6 Distribution plot for Label

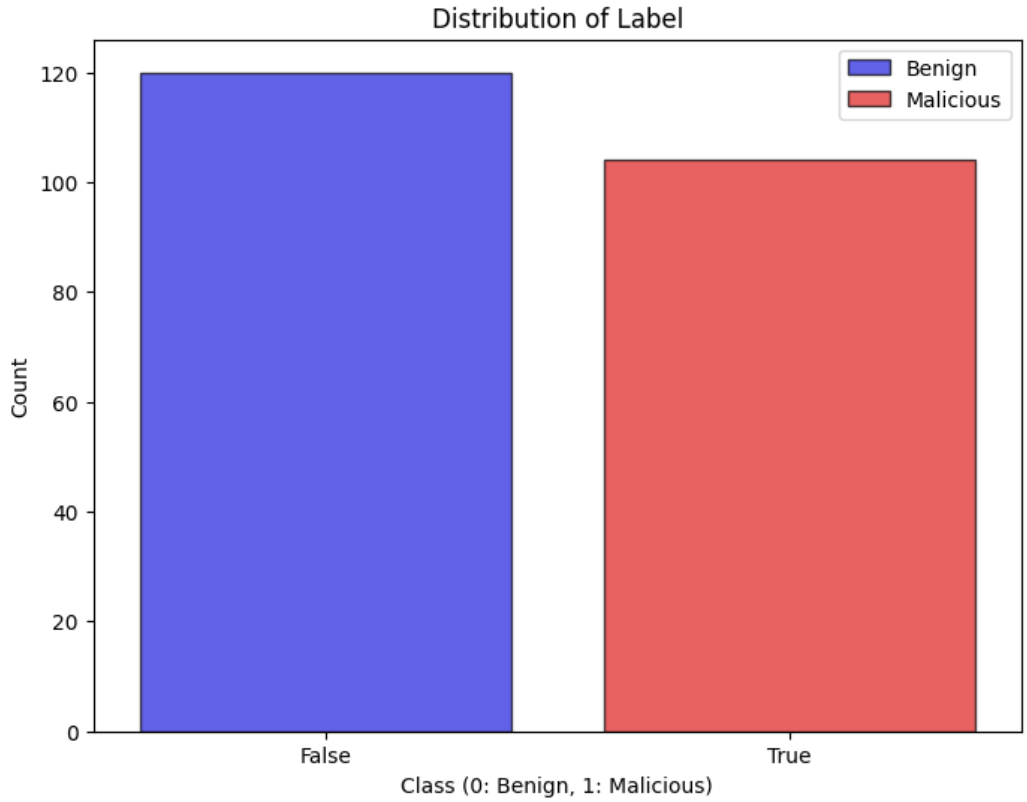


Figure 6.1.6: Distribution Plot for Label

Table 6.1.3: Descriptive Statistics of Label

Metric	Value
Count	224
Unique	2
Top	False
Frequency	120
0	120
1	104

6.2 Feature Engineering

To add more features to the same dataset, we found that we could calculate based on the filter variable. We observed that we could create three types of features:

1. Text Features
   Such features could be generated using some text operations such as omission of words or text transformation
2. Numerical
   These features were just the frequency counts or count of characters in the Text
3. Dummy
   These features could indicate the presence of particular significant words in the text.

For each generated feature, we also conducted a descriptive analysis as done previously, including calculating descriptive statistical metrics and visualizing their distributions.

6.2.1 Engineered Text Features

Upon inspection of the “filter” variable's word frequencies, we discovered that the highest frequencies were held by words that were referencing a key/document. We believe that this may cause some amount of noise to be produced as an injection query is malicious due to the operations it conducts and not necessarily because of the column/key/document it has targeted.

Hence, we wanted to further tokenize the filter to only include the following:

- Operators
- Instructions
- Keywords

We sourced the value of the conditions (i.e., the words that need to be considered by the tokenizing operations) by referencing the MongoDB documentation and selecting the corresponding words that are defined in the documentation.

The final list of words to be considered has been divided into several categories and has been presented as follows. (Please note that only words that exist in our dataset have been considered for this tokenizing algorithm. For a different dataset, a similar logic would have to be used to generate the condition values and then conduct the tokenization.)

Words to be considered in the tokenization algorithm

Table 6.2.1: Category of operators found in MongoDB found in Dataset

Category	**Description[32]**	**Words**
Comparison Operators	These are Symbols/Operators which allow basic logical statements	<, =, < $gt $eq $in $nin $ne
Where Operator	This is the $where Operator which allows the execution of custom Javascript code	$where
Regex Operator	These are operators that execute Regular Expression Operations	$regex $elemSelect
Option Operator	This is an extension to the $regex Operator which allows further configurations	$option
Math Operator	Operators who execute mathematical operations	$mod
Size Operator	Operators/keywords that return the size of the argument	$size length
This Operator	The “This” keyword	this

This new transformed text column was named “Denamed”

6.2.1.a Descriptive Statistics of Denamed

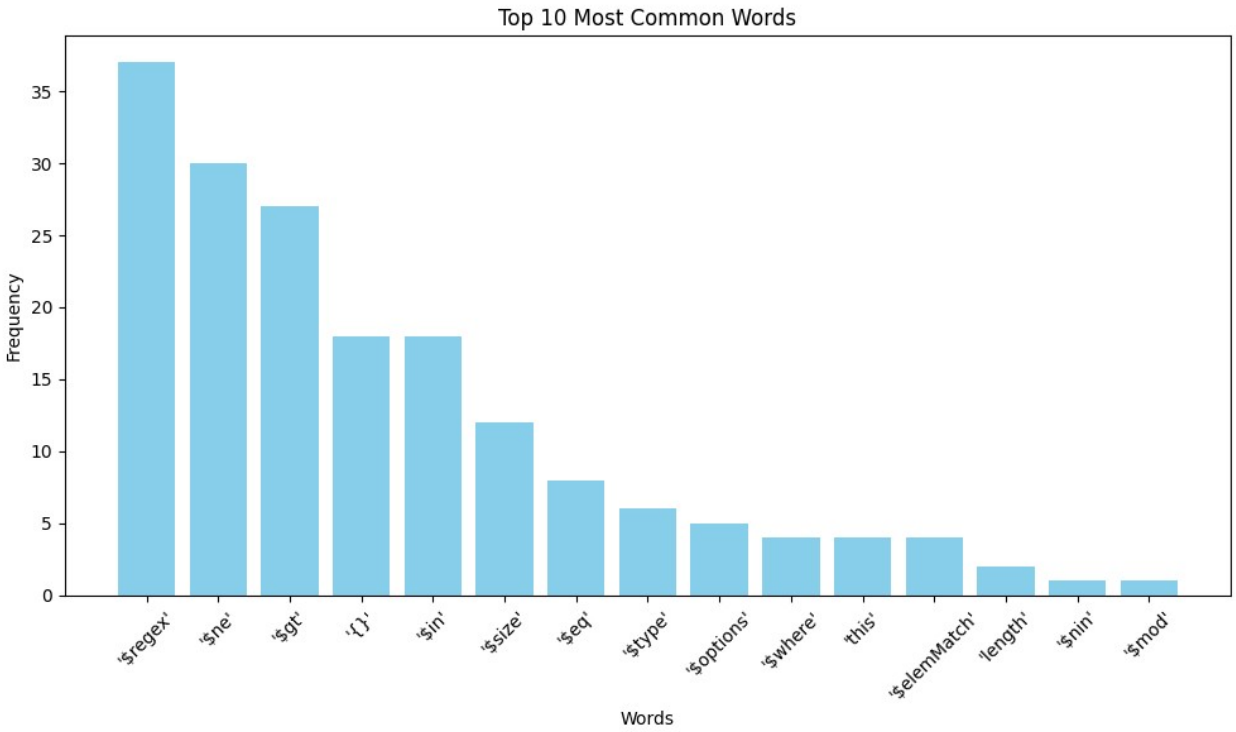


Figure 6.2.1: Word Frequencies in denamed Column

6.2.1.b Frequency Table

Table 6.2.2: Word Frequencies in denamed

Word	Frequency
{}'	18
$regex'	37
$ne'	30
$gt'	27
$where'	4
this'	4
$nin'	1
$in'	18
$options'	5
$mod'	1
length'	2
$size'	12
$type'	6
$elemMatch'	4
$eq'	8

6.2.2 Engineered Numerical Features

We were also interested in the size of the filter's string, which could provide additional insights into the complexity or length of the query being analyzed.

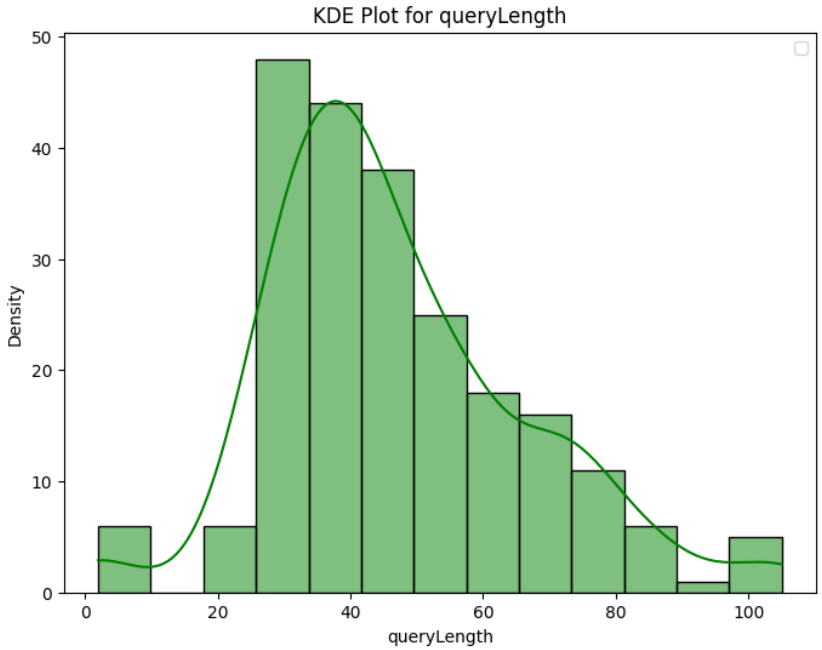


Figure 6.2.2: KDE plot of Query Length

Table 6.2.3: Descriptive Statistics of Query Length

Metric	**Value**
Count	244
Mean	46.77
Standard Deviation	19.57
Min	2
25%	33
50%	43
75%	58
Max	105

6.2.3 Engineered Dummy Variables

Based on our observations from the Engineered Text Variables (i.e., the denamed column), we decided to create dummy variables for each type of category of operator found in each instance of the filter variable.

The following variables were created for each category with the boolean type:

Table 6.8 Variable names for each MongoDB Operator Category

Category	**Variable**
Comparison Operators	logicOperator
Where Operator	whereDetected
Regex Operator	regexDetected
Option Operator	matcherOption
Math Operator	MathOperator
Size Operator	Size
This Operator	thisPointer

6.2.3.a Descriptive Statistics of Dummy Features

from left to right, top to bottom
whereDetected, regexDetected, logicOperator, MathOperator, Size, thisPointer, matcherOperator, label

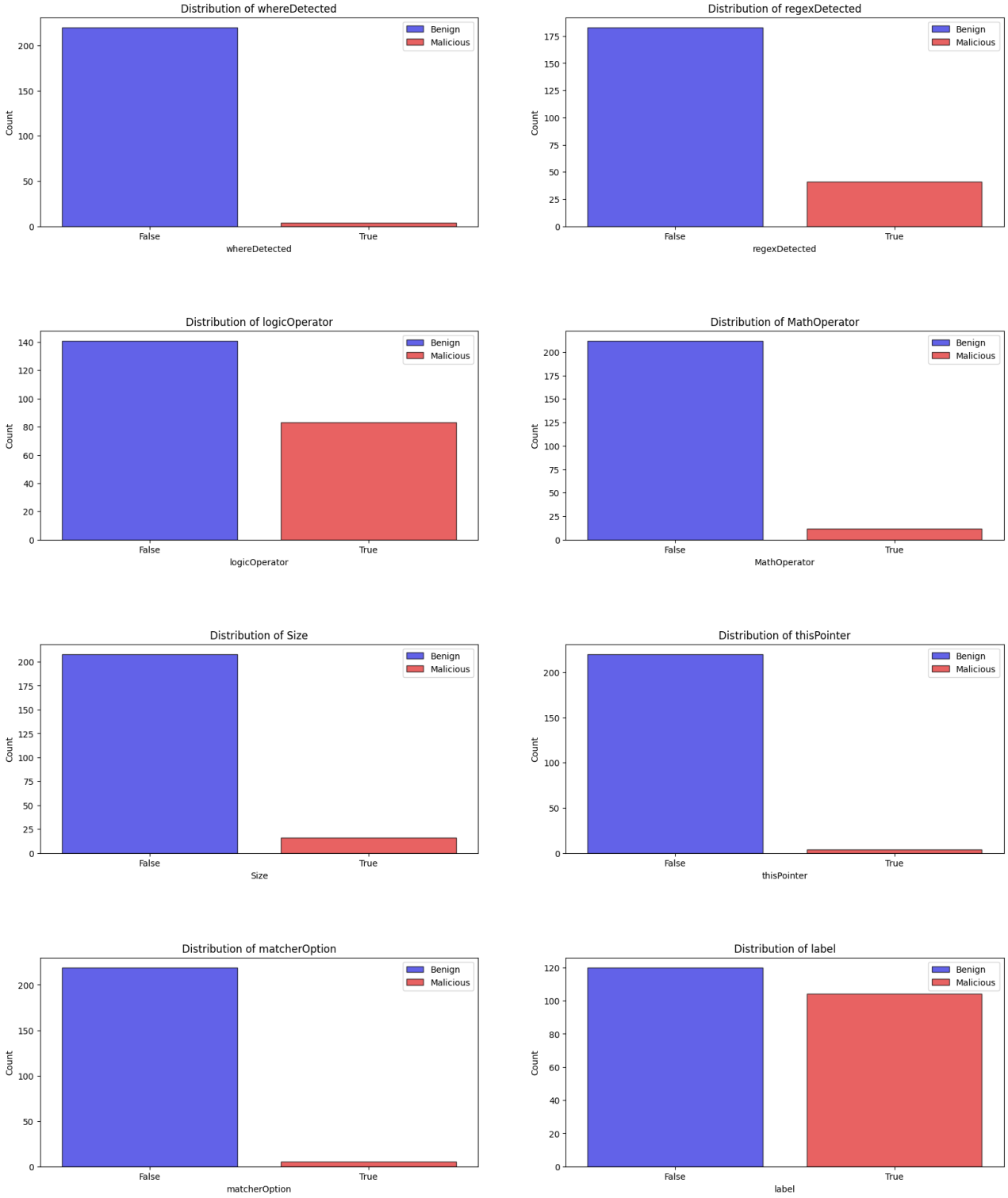


Figure 6.3.1: Figure 6.9 Burnolli plots of Dummy Features

Table 6.3.1: Descriptive Statistics of Each Engineered Dummy Variable

Variable	Count	Unique	Top	Frequency
logicOperator	244	(0, 1)	False	141
whereDetected	244	(0, 1)	False	220
regexDetected	244	(0, 1)	False	183
matcherOption	244	(0, 1)	False	219
MathOperator	244	(0, 1)	False	212
Size	244	(0, 1)	False	208
thisPointer	244	(0, 1)	False	220

6.4 Significance Testing

We conducted significance tests on each of the variables, both source data (planningTimeMicros, cpuNanos) and engineered (queryLength, whereDetected, etc.). We grouped the variables into 2 categories: Numerical and Dummy, and conducted different sequences of tests accordingly.

6.4.1 Testing Sequence

6.4.1.a Numerical

In the numerical test sequence, we aimed to test the hypothesis that the numerical variable's sample means (where the samples are created by dividing the base dataset into 2 samples depending on the value of "label") are the same. Hence, both the T-Test or the Mann-Whiteny Test were considered since we do not have any evidence to suggest that the population that these samples come from is normally distributed or not. Therefore, we applied the following tests in sequence:

1. Kologoromov-Simirinov Test for Normality
   We applied the KS test to test the hypothesis if the data is normally distributed or not
2. T-Test and Mann-Whitney Test
   These were applied to determine if the sample means are different, if they are then it means the variable plays a significant role in differentiating an injection query from a benign query, Depending on the KS test, if found true then the T-Test results were held more significant or vice versa
3. Visualization of Sample
   We also do an overlapped KDE plot of the sample distributions to gain other visual insights

The Level of significance was taken at 1%

6.4.1.b Dummy

For the dummy variables, we applied the chi-square test. This test was used to test the hypothesis that there is no association between the Dummy variables and the Label variable.
The Level of significance was taken at 10%

6.4.2 Numerical Variable Significance Testing Results

Test Results

Table 6.10 Significance Test Results of Each Numerical Feature

Variable	KS P value	T P Value	MW P Value	Significant?
querylength	0.097899	0.132023	0.178336	FALSE
cpuNanos	0.025985	0.024056	0.011228	FALSE
planningTimeMicros	0.001472	0.002851	0.006628	TRUE

Interpretation

We find that:

- QueryLength is normally distributed, and the means of the samples are not significantly different according to the T-Test.
- cpuNanos is normally distributed, and the means of the samples are not significantly different according to the T-Test.
- planningTimeMicros is not normally distributed, but the means of the samples are significantly different according to the Mann-Whitney Test.

6.4.2.a Overlapped KDE Plots

queryLength

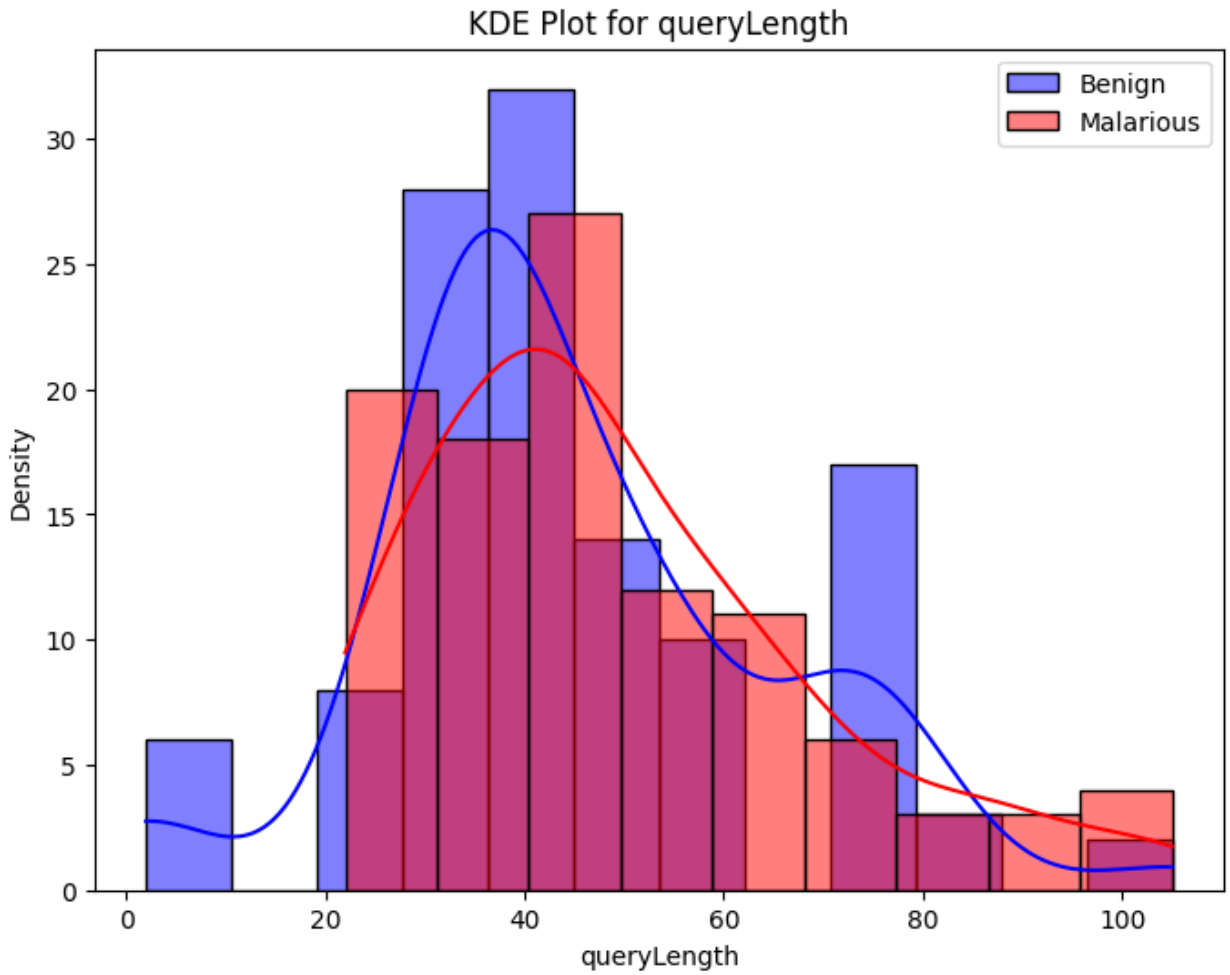


Figure 6.4.1: Overlapped KDE Plot of Query Length

cpuNanos

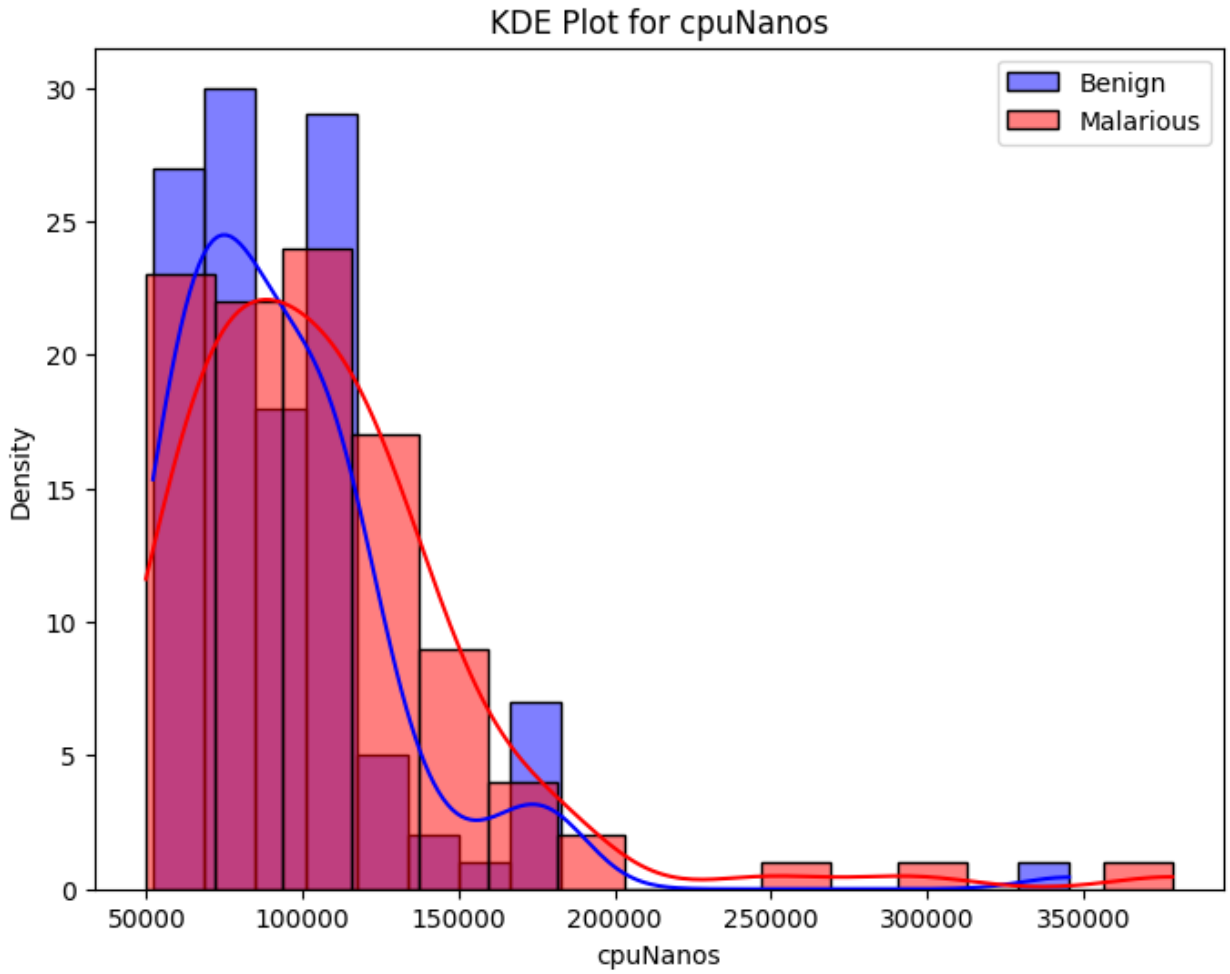


Figure 6.4.2: Overlapped KDE Plot of cpuNanos

planningTimeMicros

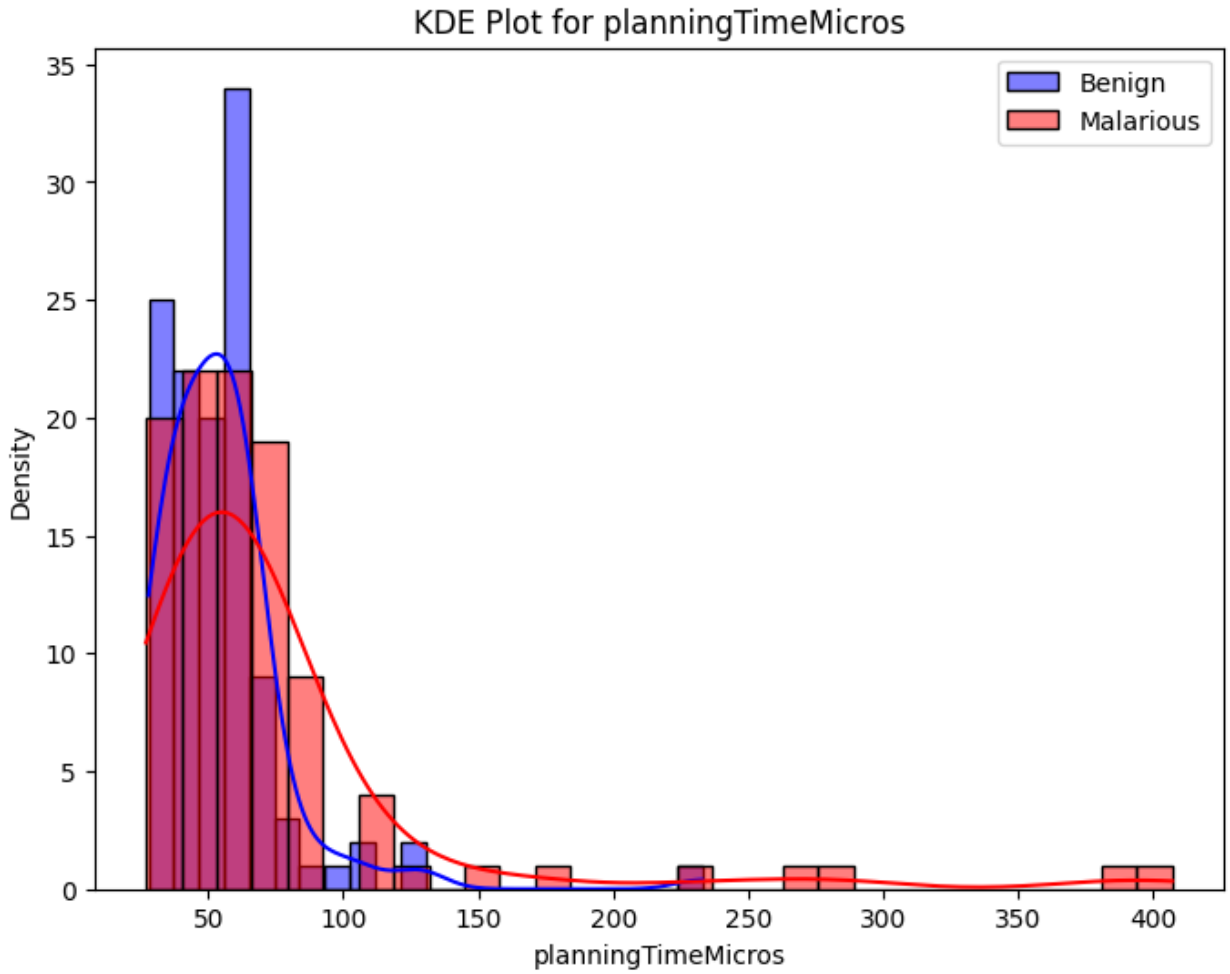


Figure 6.4.3: Overlapped KDE Plot of planningTimeMicros

6.4.3 Dummy Variable Significance Testing Results

Significant Test Results

Table 6.4.1: Significance Tests result for each Dummy Variable

Variable	Chi-sq P-value	Significance
whereDetected	0.09651	TRUE
regexDetected	0.05834	TRUE
logicOperator	0.01289	TRUE
MathOperator	0.5806	FALSE
Size	0.1100	FALSE
thisPointer	0.0965	TRUE
matcherOption	0.0481	TRUE

Interpretation

We find that:

- MathOperator and Size are the only Variables that are not associated with the label variable

6.4.4 Significance Results

After conducting the Significant test we have now created a more filtered dataset with the most significant variables taken

- filter
- dename
- planningTimeMicros
- whereDetected
- regexDetected
- logicOperator
- matcherOption
- otherOperator

6.5 Separability Analysis

The Separability analysis was conducted. This analysis was done to test if the data was Linearly Separable or non-linearly Separable[34]. To determine this, some transformations were applied to the data and visualized. If there were significant distinct groups in the data of each sample, it was inferred that the data was separable, depending on which transformation algorithm was applied. It was classified as either linearly or non-linearly separable.

6.5.1 Linear

For this test, the PCA and LDA algorithms were applied and visualized[35]. The PCA algorithm separated the groups along a vector, where the clusters of data were separated along the cardinal axis.
[36]

However, if this failed, the LDA analysis was applied. LDA applies the PCA transformation but also transforms the cardinal axis perpendicular to the vector.[37]

6.5.1.a PCA

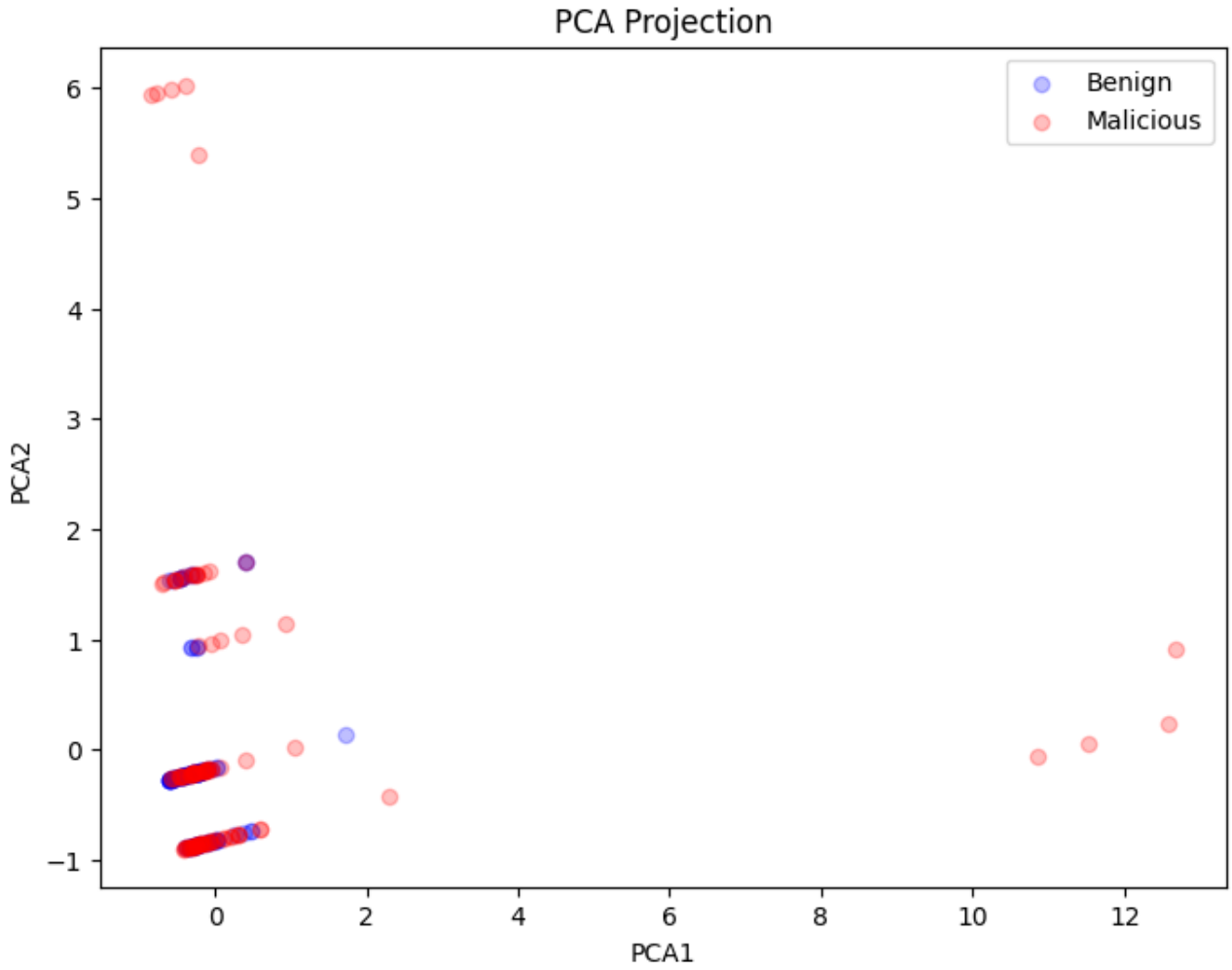


Figure 6.5.1: PCA Projection Graph

6.5.1.b LDA

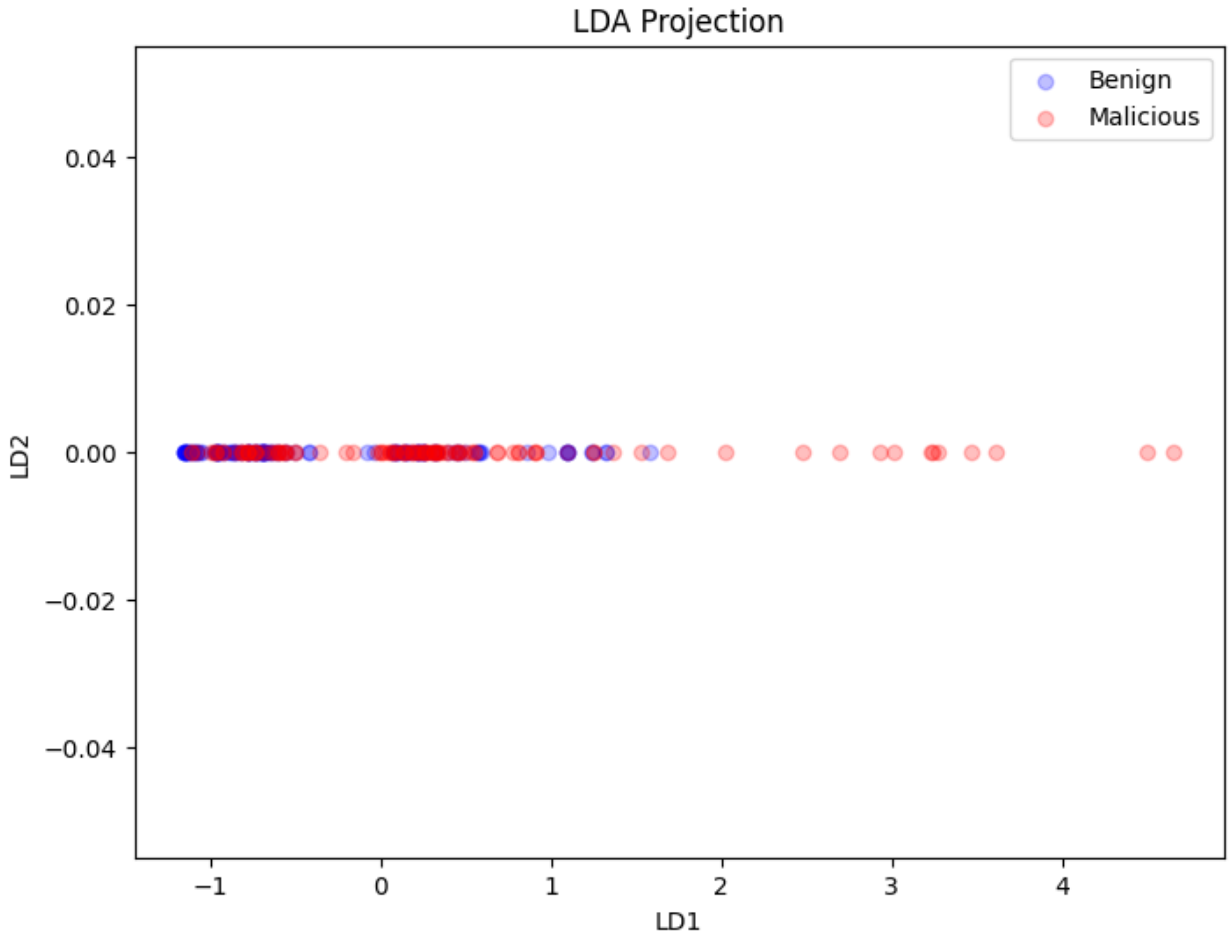


Figure 6.5.2: LDA Projection Graph

As observed in both PCA and LDA projections, a very strong overlap of the clusters of different classes was seen, albeit there is a distinct group of Malicious query data points. However, the size of these distinct groups is not satisfactorily significant enough for this data. Hence, it was concluded that the data is not linearly separable.

6.5.2 Non-Linear

For this, the t-SNE Algorithm was applied. Upon transformation, the k-means algorithm was applied to determine the clusters and plot the cluster polygons to determine how much overlap of data is there.

t-SNE plot

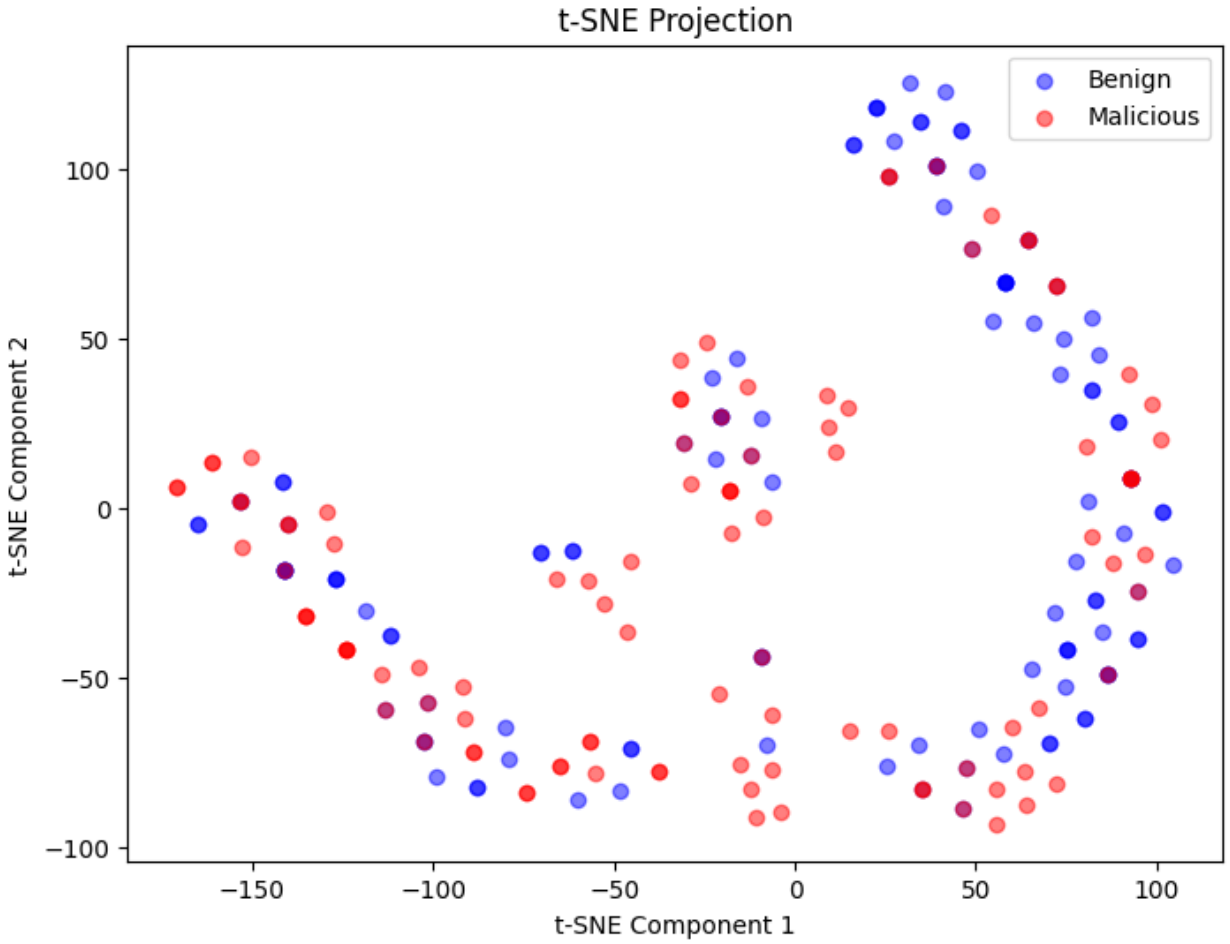


Figure 6.5.3: t-SNE Projection Graph

K-means

We applied k-means and manually iterated the number of clusters until it was not possible to create a 3-vertex polygon. This number was found to be 8 clusters per sample. The plots then overlapped over each other.

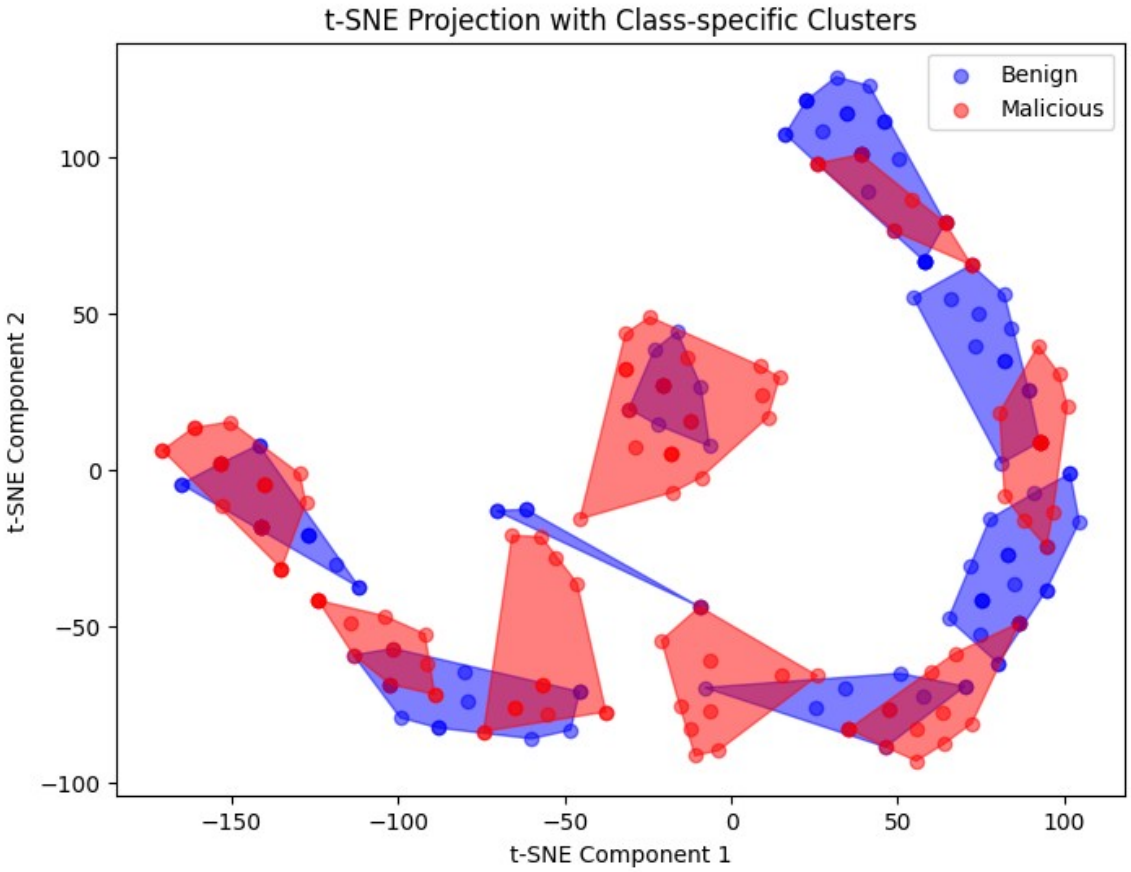


Figure 6.5.4: t-SNE with class-specific cluster polygon graphs

We then analyzed the overlapped areas of the clusters

Table 6.5.1: Cluster Polygon Area Statistics

Variable	**Metric**
Total Area of Clusters	15685.5647
Overlapping Area	2890.6762
Percentage of Overlapping	18.43%
Percentage of No Overlap	81.57%
Total Clusters	16

We concluded that the data is non-linearly separable up to 81.57%, with 18.43% being ambiguous.

6.6 Exploration Findings

With the results of the data exploration, the final training dataset for the models was constructed. It was found that the data is both linearly and non-linearly separable. When it comes to linearly separable data, it can separate more confidently for malicious queries, but it is ambiguous for benign queries.

The following table (Table 6.6.1) is the final training dataset for the model training

Table 6.6.1: Training Dataset Structure

Code	Feature Name	Type	Description
Filter	filter_str	Text	The filter used in the query
DN	denamed	Text	The filter used in the query but with the variable names and values remove
PTM	planningTimeMicro	Float	The amount of time taken by MongoDB to create a plan for the query
WD	whereDetected	Bool	If a $where operator is detected in the filter
RD	regexDetected	Bool	If a $regex operator is detected in the filter
LO	logicOperator	Bool	If a logic operator is detected in the filter
MO	matcherOption	Bool	If a $option operator is detected in the filter
TP	thisPointer	Bool	Presence of the “this” keywor
Label	Label	Bool	The target variables Where 0 = benign 1 = Malicious

7 Model Formulation

7.1 Models

We then started to train and test the models based on the training dataset that had been constructed in the previous stages.

7.1.1 Models Selected

The following classification models have been selected to be used for the experiment

- SVM[38]
    - Linear Kernel
    - Sigmoid Kernel
    - Polynomial Kernel
    - RBF Kernel
- Decision Tree[39]
    - Gradient Boosting
    - Random Forrest
    - Adaboost
    - Bagging
- Naive Bayes Classifier
- Logistic Regression
- K Nearest Neighbors

These models would be used to determine the average performance each dataset gives for a random classifier

7.1.2 Inclusion of FLAML Models

FLAML[40] is a Python library that automatically determines the best model for a given dataset using this library we determine the best model for each dataset

7.2 Dataset Configurations

We also wanted to test the difference in performance between different types of data available to the model. Hence, we categorized sets of variables into different categories depending on when they are available to the model. To further explain the concept, we introduced the concept of the model's "position".

The model's "position" can be in one of two points:

- Front of the database system:
    - The model is only able to be run on query-based data such as the properties of the query's filter and length.
    - A use case for such a model would be the detection of a malicious injection query before it reaches the database.

- Back of the database:
    - The model has access to the query sent and its properties as well as other execution-based properties from the log.
    - A use case for such a model would be to identify malicious injection queries in the log files of a database.

With this insight, we categorized the feature variables into several categories based on the position they are available in:

Table 7.2.1: Category of variables per available position

Position	**Category**	**CODE**	**Feature Name**
Front and Back	Query Filter Raw	QFR	filter_str
	Query Filter Structure	QFS	denamed
(Front and Back) Calculated when Processing	Query Filter Properties	QFP	whereDetected
			regexDetected
			logicOperator
			matcherOption
			thisPointer
Back Only	Query Post Execution Data	QPX	planningTimeMicros

7.2.1 Types

Using different types of combinations of categories, we created the following configurations of the training dataset. The white cell with a checkmark (☑) indicates which categories of variables were allotted to each dataset.

Table 7.2.2: Dataset Configurations with enabled variable categories

Dataset	**QFR**	**QFS**	**QFP**	**QPX**
A1	☑			
A2		☑		
B1	☑		☑	
B2		☑	☑	
C			☑	
D1	☑		☑	☑
D2		☑	☑	☑

7.3 Vectorization

Before inputting the data into the models, some datasets contained text data, requiring conversion into numerical data.

We needed to use a vectorization algorithm to convert the text data. We had a choice between TF-IDF vectorization or count vectorization.

After referencing a comparative study[41] we decided that using the Count Vectorization algorithm for our project was the best approach.

The Count Vectorization algorithm would convert each filter instance into a vector, where each dimension represents a unique word. For the tokenization of the text data into words for the algorithm to process, we used our developed tokenizer.

7.4 Model Results

To analyze the model's performance, we stored the confusion matrix results of each model. Confusion matrix metrics such as accuracy, precision, recall, and F1 score would then be calculated based on these results.

8 Experimental Results

8.1 Model Performance per Dataset

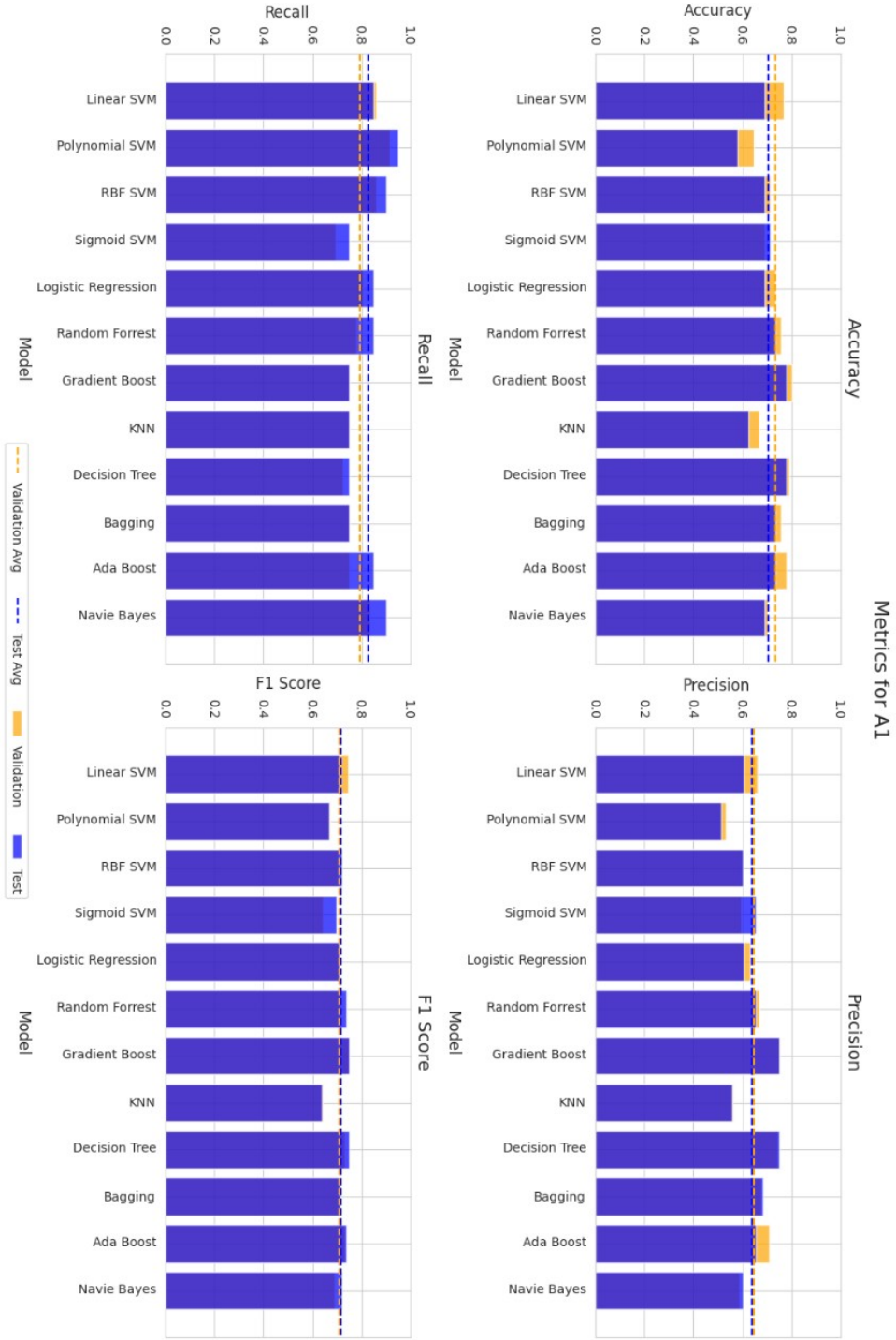


Figure 8.1.1: Model Performance for Dataset A1

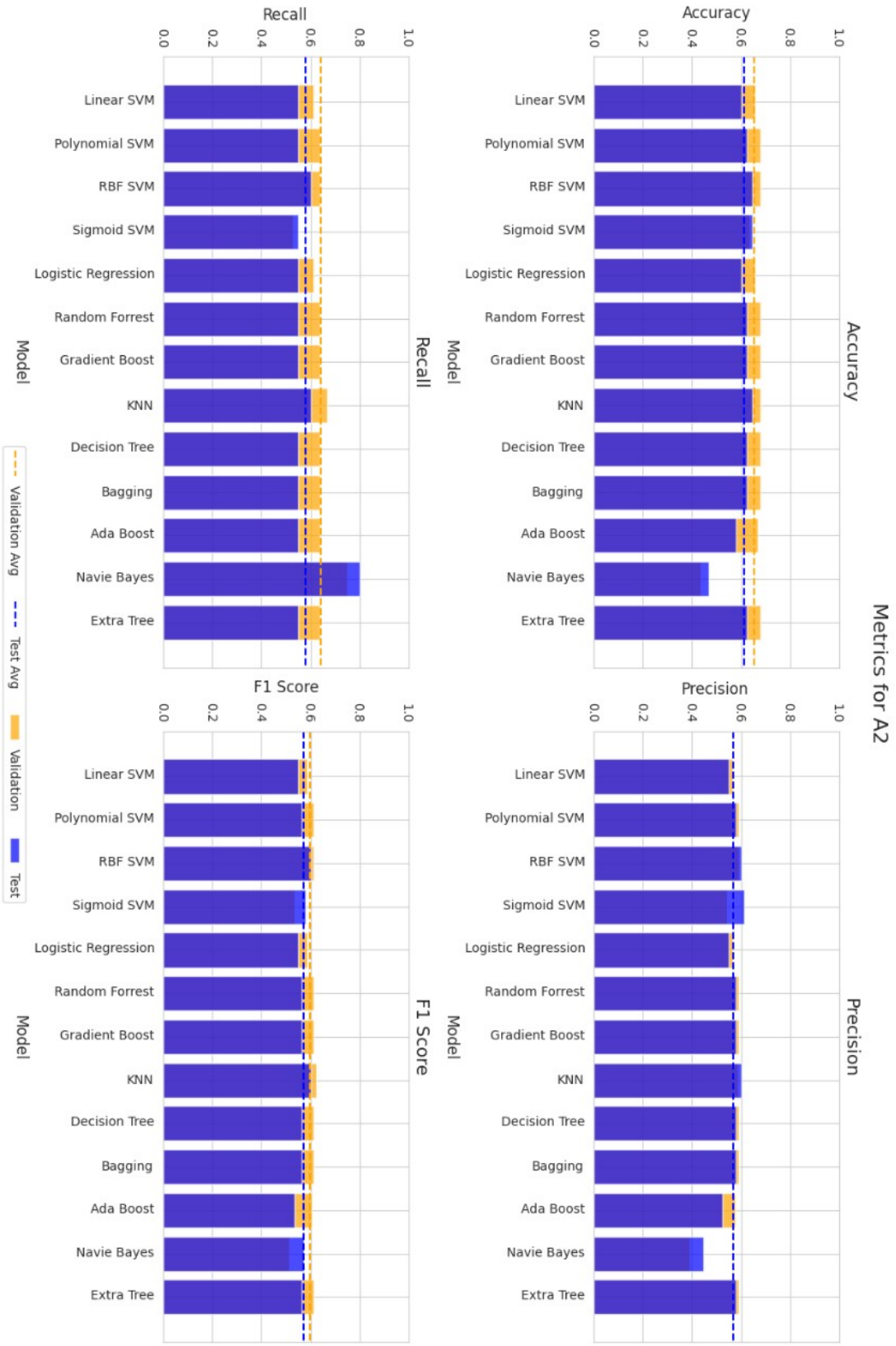


Figure 8.1.2: Model Performance for Dataset A2

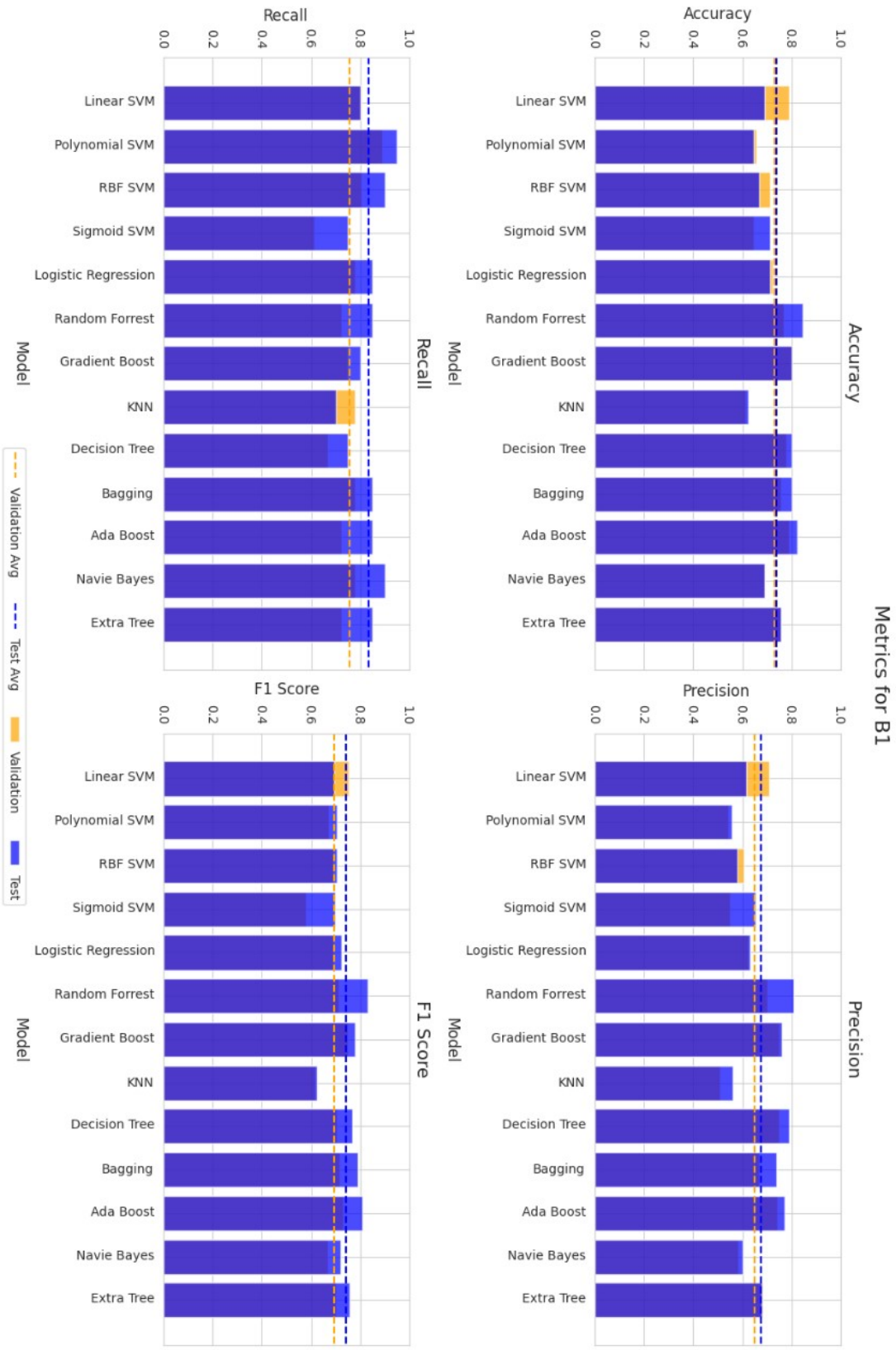


Figure 8.1.3: Model Performance for Dataset B1

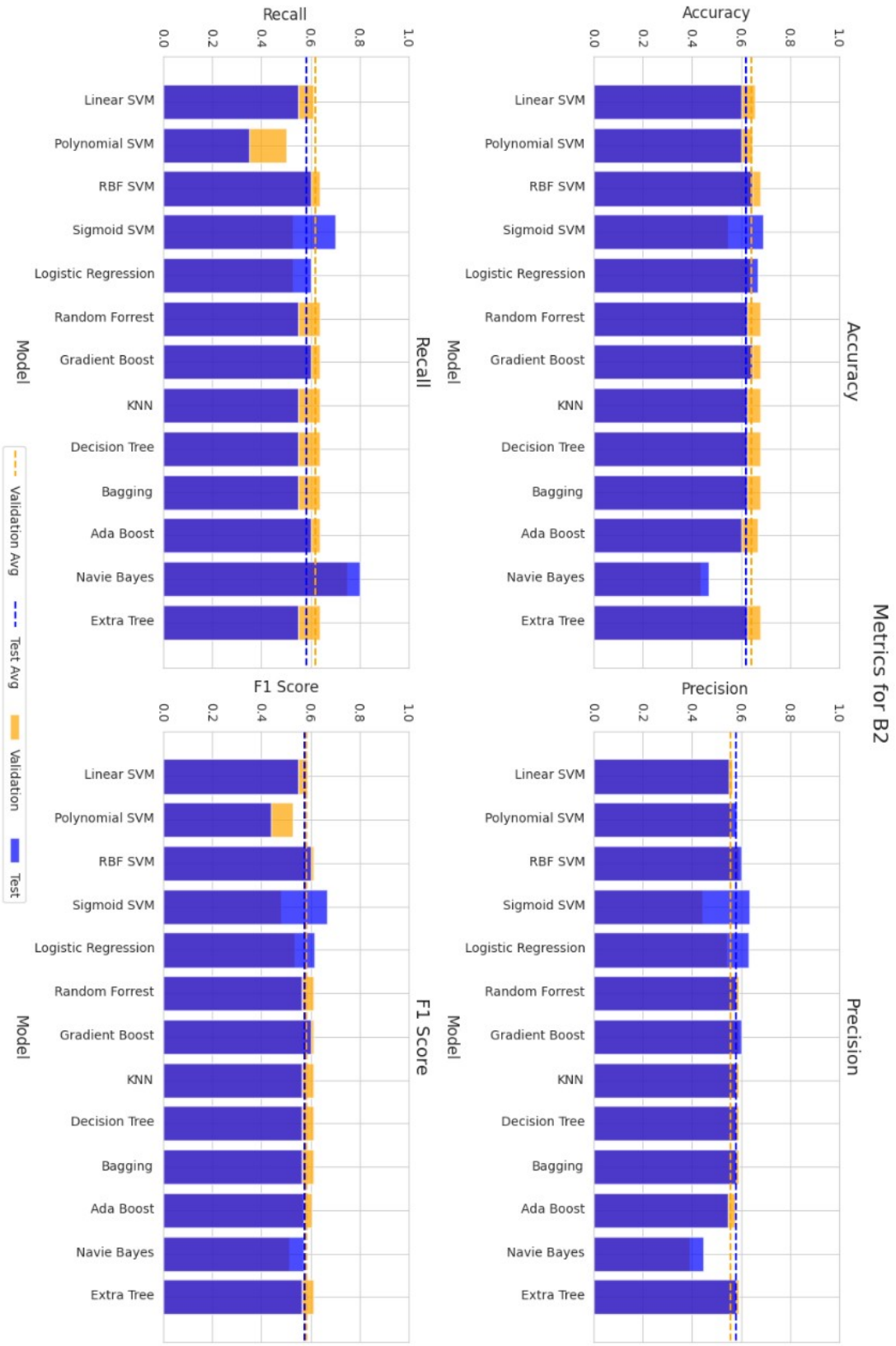


Figure 8.1.4: Model Performance for Dataset B2

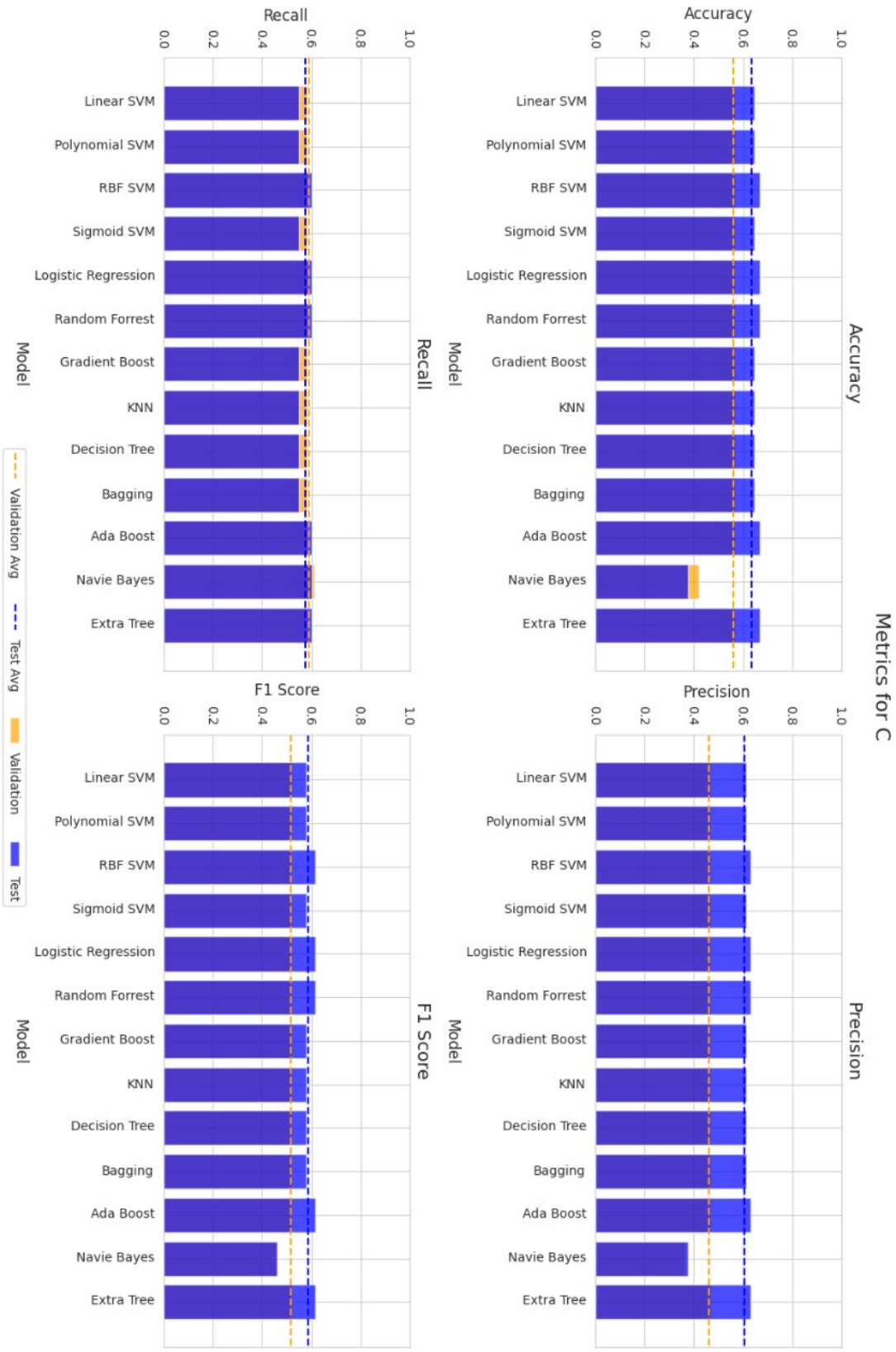


Figure 8.1.5: Model Performance for Dataset C

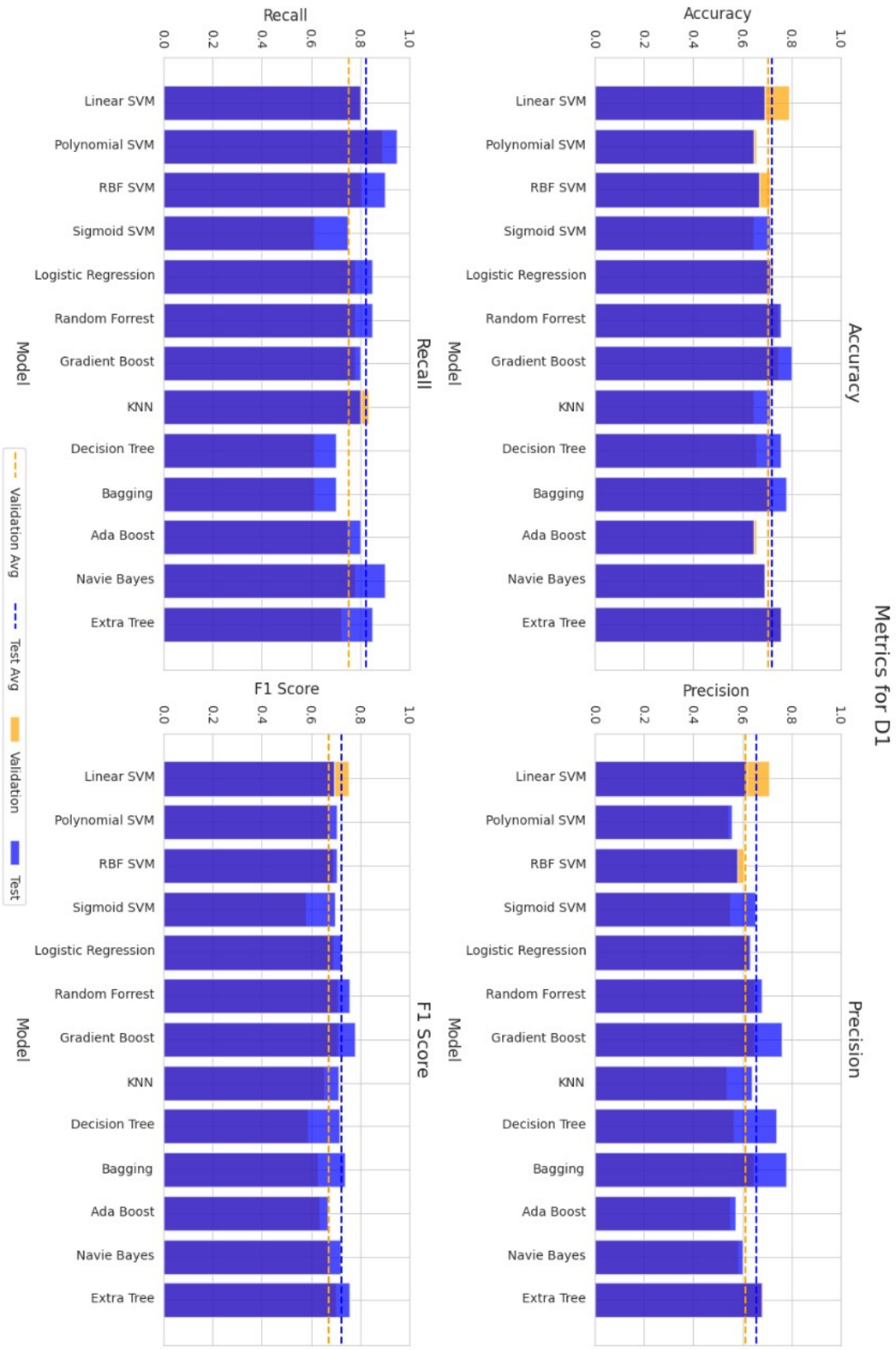


Figure 8.1.6: Model Performance for Dataset D1

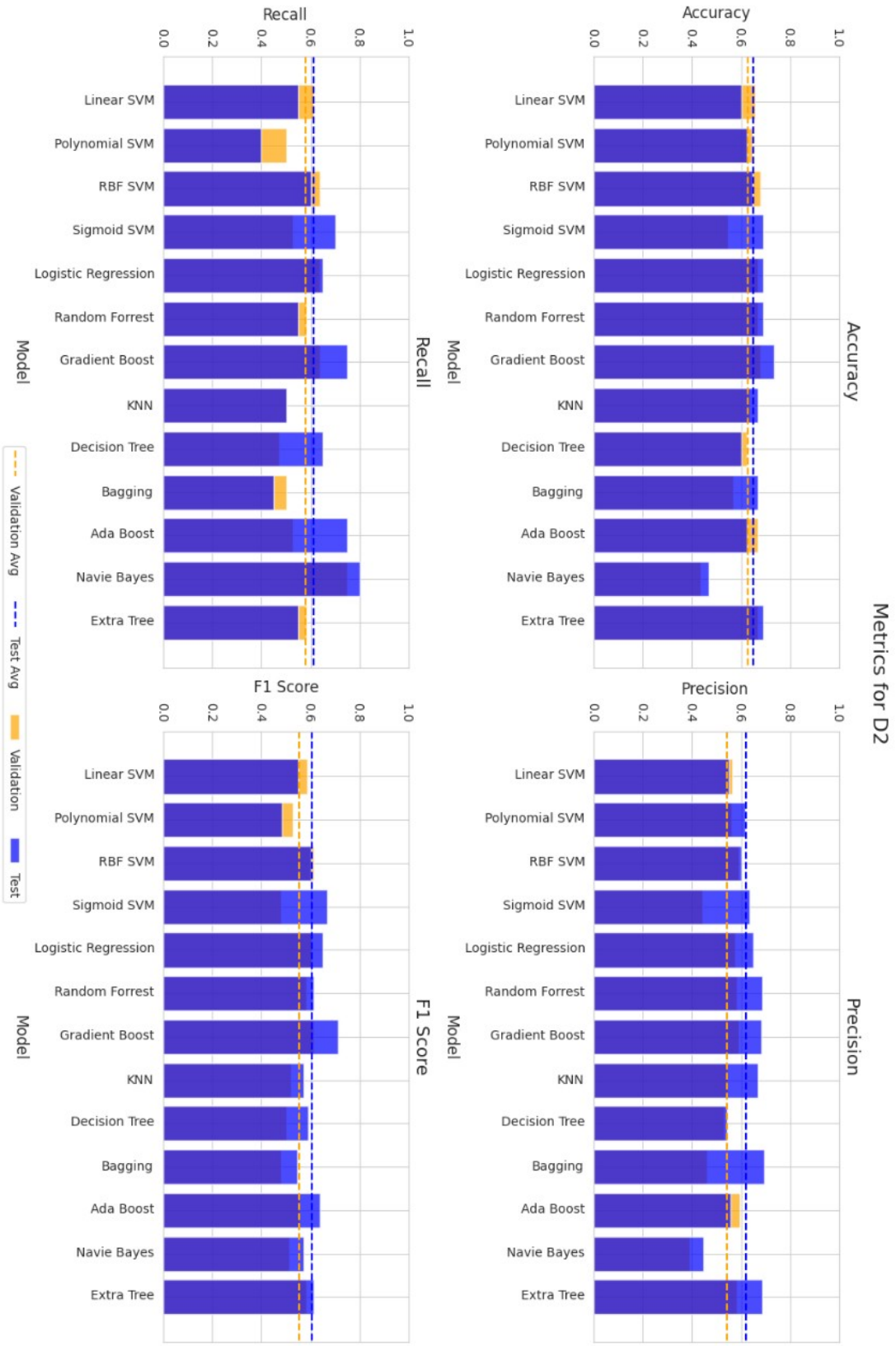


Figure 8.1.7: Model Performance for Dataset D2

Table 8.1.1: Individual Model Performance per Dataset

Dataset	Model	Split	Accuracy	Precision	Recall	F1 Score
A1	Linear SVM	Validation	76.67%	65.96%	86.11%	74.70%
A1	Polynomial SVM	Validation	64.44%	53.23%	91.67%	67.35%
A1	RBF SVM	Validation	71.11%	59.62%	86.11%	70.45%
A1	Sigmoid SVM	Validation	68.89%	59.52%	69.44%	64.10%
A1	Logistic Regression	Validation	73.33%	63.04%	80.56%	70.73%
A1	Random Forrest	Validation	75.56%	66.67%	77.78%	71.79%
A1	Gradient Boost	Validation	80.00%	75.00%	75.00%	75.00%
A1	KNN	Validation	66.67%	56.25%	75.00%	64.29%
A1	Decision Tree	Validation	78.89%	74.29%	72.22%	73.24%
A1	Bagging	Validation	75.56%	67.50%	75.00%	71.05%
A1	Ada Boost	Validation	77.78%	71.05%	75.00%	72.97%
A1	Naive Bayes	Validation	70.00%	58.82%	83.33%	68.97%
A2	Linear SVM	Validation	65.56%	56.41%	61.11%	58.67%
A2	Polynomial SVM	Validation	67.78%	58.97%	63.89%	61.33%
A2	RBF SVM	Validation	67.78%	58.97%	63.89%	61.33%
A2	Sigmoid SVM	Validation	63.33%	54.29%	52.78%	53.52%
A2	Logistic Regression	Validation	65.56%	56.41%	61.11%	58.67%
A2	Random Forrest	Validation	67.78%	58.97%	63.89%	61.33%
A2	Gradient Boost	Validation	67.78%	58.97%	63.89%	61.33%
A2	KNN	Validation	67.78%	58.54%	66.67%	62.34%
A2	Decision Tree	Validation	67.78%	58.97%	63.89%	61.33%
A2	Bagging	Validation	61.11%	51.02%	69.44%	58.82%
A2	Ada Boost	Validation	66.67%	57.50%	63.89%	60.53%
A2	Naive Bayes	Validation	43.33%	39.13%	75.00%	51.43%
B1	Linear SVM	Validation	78.89%	70.73%	80.56%	75.32%
B1	Polynomial SVM	Validation	65.56%	54.24%	88.89%	67.37%
B1	RBF SVM	Validation	71.11%	60.42%	80.56%	69.05%
B1	Sigmoid SVM	Validation	64.44%	55.00%	61.11%	57.89%
B1	Logistic Regression	Validation	72.22%	62.22%	77.78%	69.14%
B1	Random Forrest	Validation	72.22%	63.41%	72.22%	67.53%
B1	Gradient Boost	Validation	80.00%	75.00%	75.00%	75.00%

B1	KNN	Validation	61.11%	50.91%	77.78%	61.54%
B1	Decision Tree	Validation	77.78%	75.00%	66.67%	70.59%
B1	Bagging	Validation	74.44%	67.57%	69.44%	68.49%
B1	Ada Boost	Validation	76.67%	70.27%	72.22%	71.23%
B1	Naive Bayes	Validation	68.89%	58.33%	77.78%	66.67%
B2	Linear SVM	Validation	65.56%	56.41%	61.11%	58.67%
B2	Polynomial SVM	Validation	64.44%	56.25%	50.00%	52.94%
B2	RBF SVM	Validation	67.78%	58.97%	63.89%	61.33%
B2	Sigmoid SVM	Validation	54.44%	44.19%	52.78%	48.10%
B2	Logistic Regression	Validation	63.33%	54.29%	52.78%	53.52%
B2	Random Forrest	Validation	67.78%	58.97%	63.89%	61.33%
B2	Gradient Boost	Validation	67.78%	58.97%	63.89%	61.33%
B2	KNN	Validation	67.78%	58.97%	63.89%	61.33%
B2	Decision Tree	Validation	67.78%	58.97%	63.89%	61.33%
B2	Bagging	Validation	67.78%	61.29%	52.78%	56.72%
B2	Ada Boost	Validation	66.67%	57.50%	63.89%	60.53%
B2	Naive Bayes	Validation	43.33%	39.13%	75.00%	51.43%
C	Linear SVM	Validation	56.67%	46.67%	58.33%	51.85%
C	Polynomial SVM	Validation	56.67%	46.67%	58.33%	51.85%
C	RBF SVM	Validation	56.67%	46.67%	58.33%	51.85%
C	Sigmoid SVM	Validation	56.67%	46.67%	58.33%	51.85%
C	Logistic Regression	Validation	56.67%	46.67%	58.33%	51.85%
C	Random Forrest	Validation	56.67%	46.67%	58.33%	51.85%
C	Gradient Boost	Validation	56.67%	46.67%	58.33%	51.85%
C	KNN	Validation	56.67%	46.67%	58.33%	51.85%
C	Decision Tree	Validation	56.67%	46.67%	58.33%	51.85%
C	Bagging	Validation	56.67%	46.67%	58.33%	51.85%
C	Ada Boost	Validation	56.67%	46.67%	58.33%	51.85%
C	Naive Bayes	Validation	42.22%	36.67%	61.11%	45.83%
D1	Linear SVM	Validation	78.89%	70.73%	80.56%	75.32%
D1	Polynomial SVM	Validation	65.56%	54.24%	88.89%	67.37%
D1	RBF SVM	Validation	71.11%	60.42%	80.56%	69.05%
D1	Sigmoid SVM	Validation	64.44%	55.00%	61.11%	57.89%
D1	Logistic Regression	Validation	72.22%	62.22%	77.78%	69.14%

D1	Random Forrest	Validation	72.22%	61.70%	80.56%	69.88%
D1	Gradient Boost	Validation	73.33%	64.29%	75.00%	69.23%
D1	KNN	Validation	64.44%	53.57%	83.33%	65.22%
D1	Decision Tree	Validation	66.67%	58.33%	58.33%	58.33%
D1	Bagging	Validation	71.11%	60.00%	83.33%	69.77%
D1	Ada Boost	Validation	65.56%	55.10%	75.00%	63.53%
D1	Naive Bayes	Validation	68.89%	58.33%	77.78%	66.67%
D2	Linear SVM	Validation	65.56%	56.41%	61.11%	58.67%
D2	Polynomial SVM	Validation	64.44%	56.25%	50.00%	52.94%
D2	RBF SVM	Validation	67.78%	58.97%	63.89%	61.33%
D2	Sigmoid SVM	Validation	54.44%	44.19%	52.78%	48.10%
D2	Logistic Regression	Validation	66.67%	57.50%	63.89%	60.53%
D2	Random Forrest	Validation	66.67%	58.33%	58.33%	58.33%
D2	Gradient Boost	Validation	72.22%	67.74%	58.33%	62.69%
D2	KNN	Validation	63.33%	54.55%	50.00%	52.17%
D2	Decision Tree	Validation	65.56%	58.06%	50.00%	53.73%
D2	Bagging	Validation	70.00%	63.64%	58.33%	60.87%
D2	Ada Boost	Validation	66.67%	59.38%	52.78%	55.88%
D2	Naive Bayes	Validation	43.33%	39.13%	75.00%	51.43%
A1	Linear SVM	Test	68.89%	60.71%	85.00%	70.83%
A1	Polynomial SVM	Test	57.78%	51.35%	95.00%	66.67%
A1	RBF SVM	Test	68.89%	60.00%	90.00%	72.00%
A1	Sigmoid SVM	Test	71.11%	65.22%	75.00%	69.77%
A1	Logistic Regression	Test	68.89%	60.71%	85.00%	70.83%
A1	Random Forrest	Test	73.33%	65.38%	85.00%	73.91%
A1	Gradient Boost	Test	77.78%	75.00%	75.00%	75.00%
A1	KNN	Test	62.22%	55.56%	75.00%	63.83%
A1	Decision Tree	Test	77.78%	75.00%	75.00%	75.00%
A1	Bagging	Test	73.33%	68.18%	75.00%	71.43%
A1	Ada Boost	Test	73.33%	65.38%	85.00%	73.91%
A1	Naive Bayes	Test	68.89%	60.00%	90.00%	72.00%
A2	Linear SVM	Test	60.00%	55.00%	55.00%	55.00%
A2	Polynomial SVM	Test	62.22%	57.89%	55.00%	56.41%
A2	RBF SVM	Test	64.44%	60.00%	60.00%	60.00%

A2	Sigmoid SVM	Test	64.44%	61.11%	55.00%	57.89%
A2	Logistic Regression	Test	60.00%	55.00%	55.00%	55.00%
A2	Random Forrest	Test	62.22%	57.89%	55.00%	56.41%
A2	Gradient Boost	Test	62.22%	57.89%	55.00%	56.41%
A2	KNN	Test	64.44%	60.00%	60.00%	60.00%
A2	Decision Tree	Test	62.22%	57.89%	55.00%	56.41%
A2	Bagging	Test	64.44%	59.09%	65.00%	61.90%
A2	Ada Boost	Test	57.78%	52.38%	55.00%	53.66%
A2	Naive Bayes	Test	46.67%	44.44%	80.00%	57.14%
B1	Linear SVM	Test	68.89%	61.54%	80.00%	69.57%
B1	Polynomial SVM	Test	64.44%	55.88%	95.00%	70.37%
B1	RBF SVM	Test	66.67%	58.06%	90.00%	70.59%
B1	Sigmoid SVM	Test	71.11%	65.22%	75.00%	69.77%
B1	Logistic Regression	Test	71.11%	62.96%	85.00%	72.34%
B1	Random Forrest	Test	71.11%	64.00%	80.00%	71.11%
B1	Gradient Boost	Test	80.00%	76.19%	80.00%	78.05%
B1	KNN	Test	62.22%	56.00%	70.00%	62.22%
B1	Decision Tree	Test	80.00%	78.95%	75.00%	76.92%
B1	Bagging	Test	77.78%	72.73%	80.00%	76.19%
B1	Ada Boost	Test	77.78%	72.73%	80.00%	76.19%
B1	Naive Bayes	Test	68.89%	60.00%	90.00%	72.00%
B2	Linear SVM	Test	60.00%	55.00%	55.00%	55.00%
B2	Polynomial SVM	Test	60.00%	58.33%	35.00%	43.75%
B2	RBF SVM	Test	64.44%	60.00%	60.00%	60.00%
B2	Sigmoid SVM	Test	68.89%	63.64%	70.00%	66.67%
B2	Logistic Regression	Test	66.67%	63.16%	60.00%	61.54%
B2	Random Forrest	Test	62.22%	57.89%	55.00%	56.41%
B2	Gradient Boost	Test	64.44%	60.00%	60.00%	60.00%
B2	KNN	Test	62.22%	57.89%	55.00%	56.41%
B2	Decision Tree	Test	62.22%	57.89%	55.00%	56.41%
B2	Bagging	Test	57.78%	53.33%	40.00%	45.71%
B2	Ada Boost	Test	60.00%	54.55%	60.00%	57.14%
B2	Naive Bayes	Test	46.67%	44.44%	80.00%	57.14%
C	Linear SVM	Test	64.44%	61.11%	55.00%	57.89%

C	Polynomial SVM	Test	64.44%	61.11%	55.00%	57.89%
C	RBF SVM	Test	66.67%	63.16%	60.00%	61.54%
C	Sigmoid SVM	Test	64.44%	61.11%	55.00%	57.89%
C	Logistic Regression	Test	66.67%	63.16%	60.00%	61.54%
C	Random Forrest	Test	66.67%	63.16%	60.00%	61.54%
C	Gradient Boost	Test	64.44%	61.11%	55.00%	57.89%
C	KNN	Test	64.44%	61.11%	55.00%	57.89%
C	Decision Tree	Test	64.44%	61.11%	55.00%	57.89%
C	Bagging	Test	64.44%	61.11%	55.00%	57.89%
C	Ada Boost	Test	66.67%	63.16%	60.00%	61.54%
C	Naive Bayes	Test	37.78%	37.50%	60.00%	46.15%
D1	Linear SVM	Test	68.89%	61.54%	80.00%	69.57%
D1	Polynomial SVM	Test	64.44%	55.88%	95.00%	70.37%
D1	RBF SVM	Test	66.67%	58.06%	90.00%	70.59%
D1	Sigmoid SVM	Test	71.11%	65.22%	75.00%	69.77%
D1	Logistic Regression	Test	71.11%	62.96%	85.00%	72.34%
D1	Random Forrest	Test	71.11%	62.96%	85.00%	72.34%
D1	Gradient Boost	Test	80.00%	76.19%	80.00%	78.05%
D1	KNN	Test	71.11%	64.00%	80.00%	71.11%
D1	Decision Tree	Test	75.56%	73.68%	70.00%	71.79%
D1	Bagging	Test	73.33%	64.29%	90.00%	75.00%
D1	Ada Boost	Test	64.44%	57.14%	80.00%	66.67%
D1	Naive Bayes	Test	68.89%	60.00%	90.00%	72.00%
D2	Linear SVM	Test	60.00%	55.00%	55.00%	55.00%
D2	Polynomial SVM	Test	62.22%	61.54%	40.00%	48.48%
D2	RBF SVM	Test	64.44%	60.00%	60.00%	60.00%
D2	Sigmoid SVM	Test	68.89%	63.64%	70.00%	66.67%
D2	Logistic Regression	Test	68.89%	65.00%	65.00%	65.00%
D2	Random Forrest	Test	71.11%	73.33%	55.00%	62.86%
D2	Gradient Boost	Test	73.33%	68.18%	75.00%	71.43%
D2	KNN	Test	66.67%	66.67%	50.00%	57.14%
D2	Decision Tree	Test	73.33%	72.22%	65.00%	68.42%
D2	Bagging	Test	64.44%	62.50%	50.00%	55.56%
D2	Ada Boost	Test	62.22%	55.56%	75.00%	63.83%

D2	Naive Bayes	Test	46.67%	44.44%	80.00%	57.14%

8.1.1 Average Model Performance per Dataset

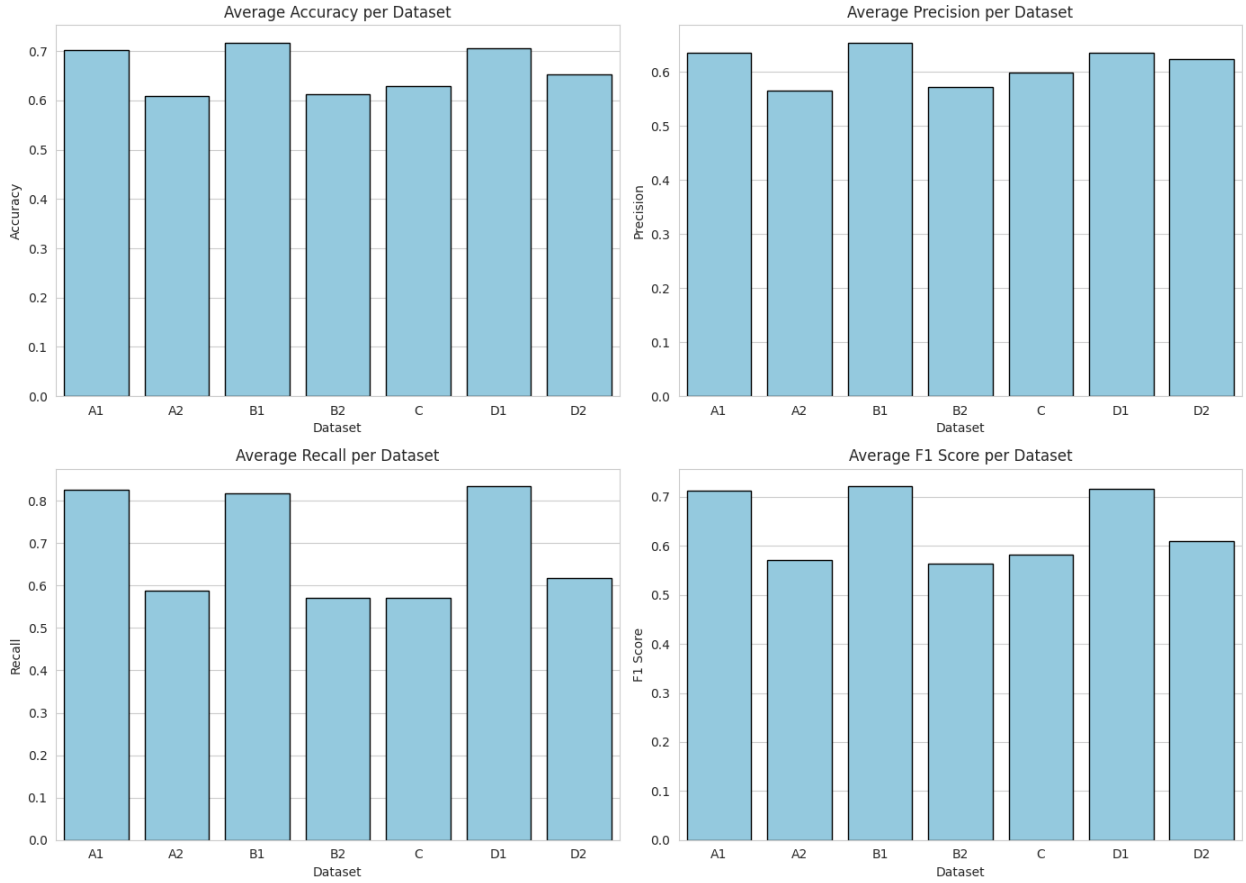


Figure 8.1.8: Bar Plot of Average Model Performance per Dataset

Table 8.1.2: Average Model Performance per Dataset

Dataset	Accuracy	Precision	Recall	F1 Score
A1	0.7019	0.6354	0.8250	0.7127
A2	0.6093	0.5655	0.5875	0.5719
B1	0.7167	0.6535	0.8167	0.7211
B2	0.6130	0.5718	0.5708	0.5635
C	0.6296	0.5983	0.5708	0.5813
D1	0.7056	0.6349	0.8333	0.7163
D2	0.6519	0.6234	0.6167	0.6096

We find that the Top 3 Datasets are A1, B1, and D1 with a Metrics Score Range of 0.7019 - 0.8333

8.1.2 Best FLAML Model per Dataset

Table 8.1.3: Performance of Best FLAML Model per Dataset

Dataset	Model	Accuracy	Precision	Recall	F1
A1	xgboost	0.8400	0.8800	0.8148	0.8462
A2	xgboost	0.5650	0.6800	0.5528	0.6099
B1	xgboost	0.8400	0.8800	0.8148	0.8462
B2	extra_tree	0.6150	0.6800	0.6018	0.6385
C	rf	0.6600	0.7200	0.6429	0.6792
D1	xgboost	0.8450	0.8400	0.8485	0.8442
D2	xgboost	0.6550	0.7600	0.6281	0.6878

9 Evaluation

9.1 Evaluation Criteria

We evaluated the models to determine:

1. Which dataset had significantly the best performance?
2. From the dataset found in the previous evaluation, which model performed the best?
3. Whether our constructed datasets had performed better than the datasets made in previous works.

We are attempting to discover if our constructed datasets have performed better than the datasets made in previous works

Here, A1 represented the dataset similar to the ones used in previous works[28], [8], [9] and A2, B1, B2, C, D1, and D2 represented the datasets we wanted to test.

To simplify the evaluation, we only considered the Top 3 datasets A1, B1, and D1.

Since we wanted to compare with A1, therefore we only considered B1 and D1 and used them in the T-Test/Mann-Whitney Test.

To compare the significance of the difference in performances, we applied either a T-Test or a Mann-Whitney Test with the Null Hypothesis being that the means of the sample were not different from each other. We conducted a one-tailed test, i.e., we wanted to see if the difference in performance was greater than A1.

If a difference was found, then we could say that any random classifier trained on this dataset could significantly perform better than any random classifier trained on A1.

After determining which dataset had performed the best, we then used the FLAML results to determine the best model for this dataset.

9.2 Results

Average Dataset Performance Compared with Dataset A1 Significance Test

Table 9.2.1: Significance Testing of each Confusion Matrix Metric

Dataset	Metric	T Test	MW Test	Significance
B1	Accuracy	0.545	0.704	FALSE
B1	Precision	0.569	0.795	FALSE
B1	Recall	0.884	0.976	FALSE
B1	F1 Score	0.523	0.707	FALSE
D1	Accuracy	0.862	0.953	FALSE
D1	Precision	0.984	0.839	FALSE
D1	Recall	0.891	0.859	FALSE
D1	F1 Score	0.843	0.750	FALSE

We noticed that for all metrics, both the T-Test and Mann-Whitney test failed at the 1% significance level. Hence, we could conclude that a random classifier trained on any of these datasets did not have a statistically significant difference in performance.

9.2.1 Distribution Plots of Dataset B1, D1's Metric against A1

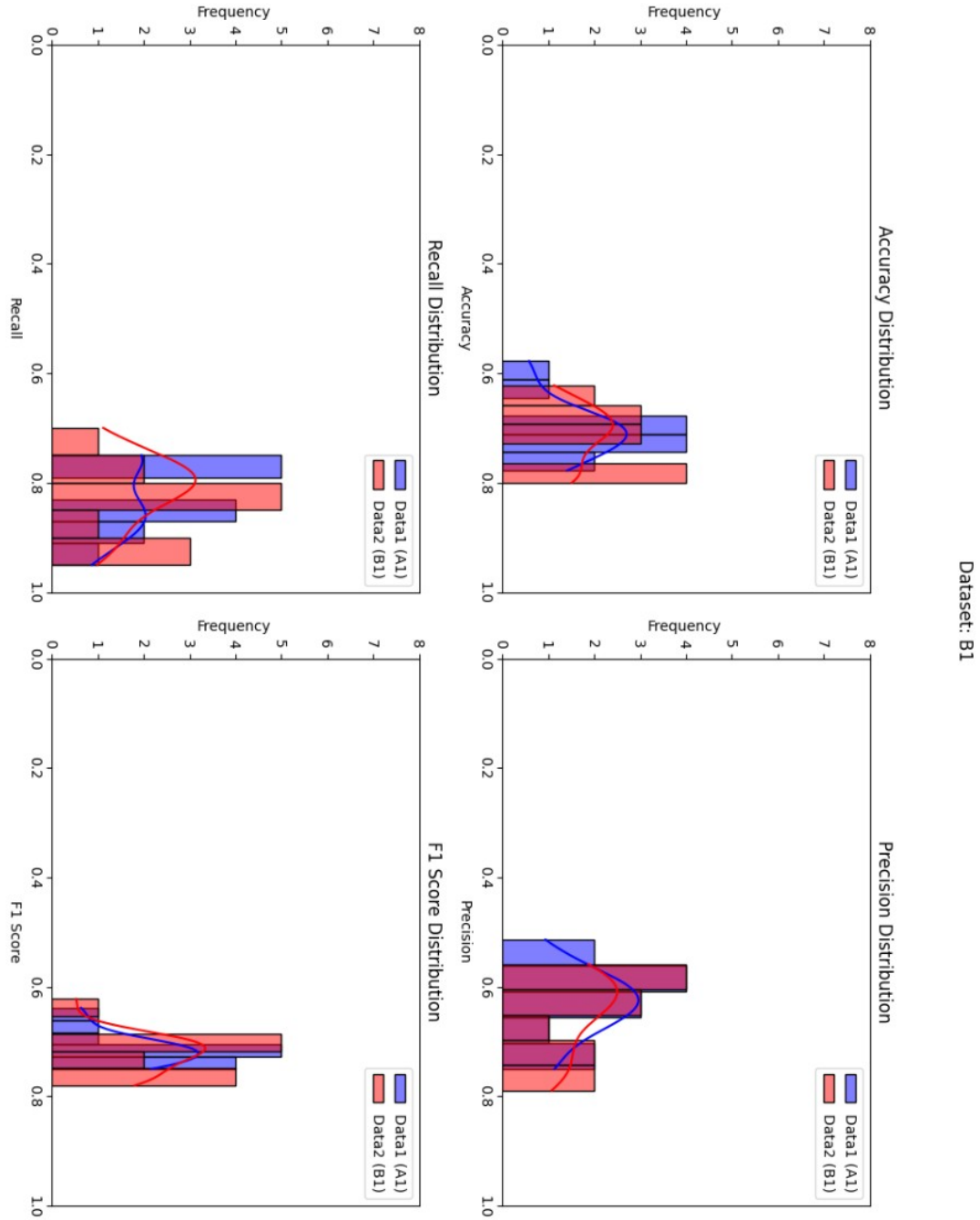


Figure 9.1: Overlapped Model Performance of A1 and B1

Figure 9.2 Model Performance for Dataset D1

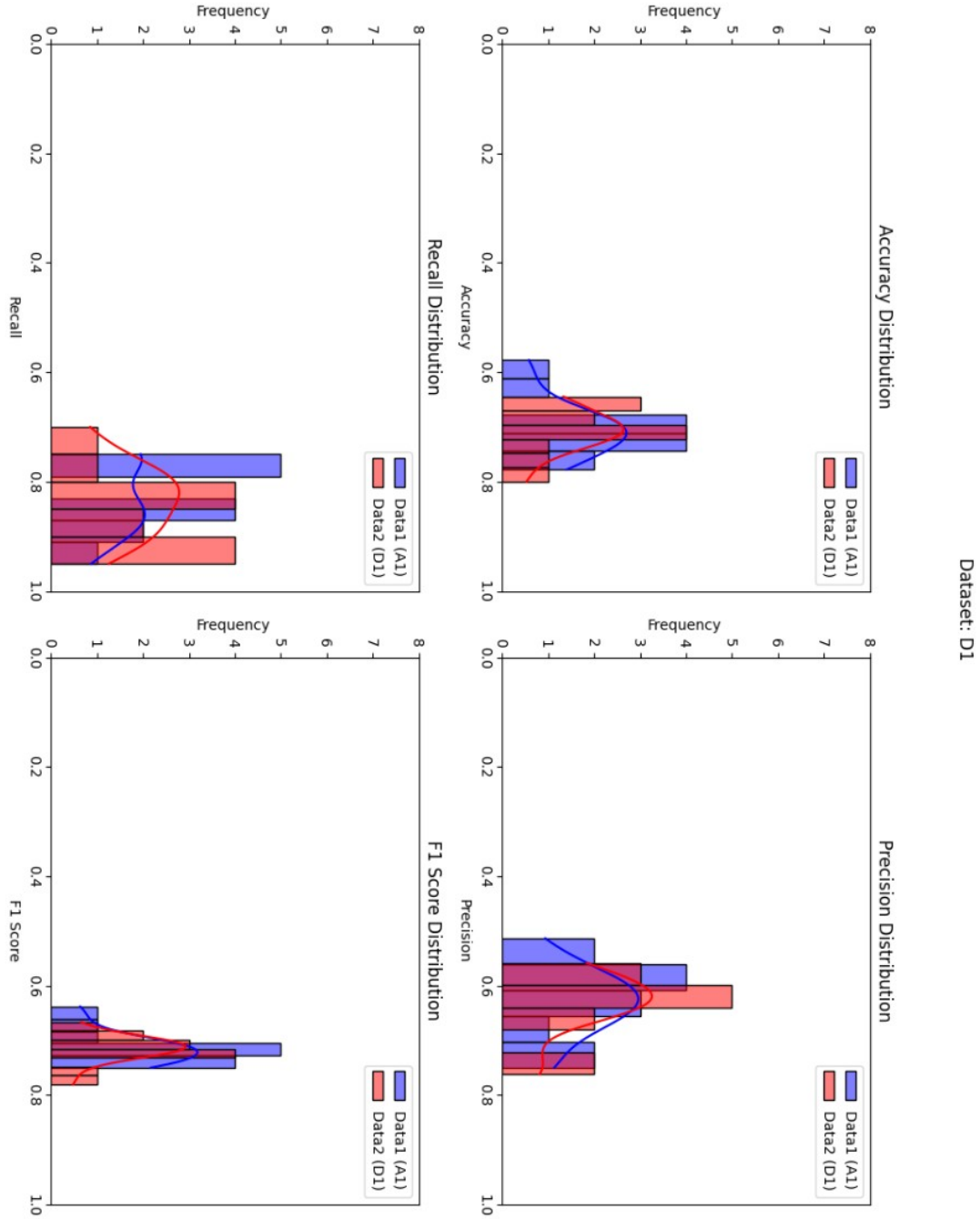


Figure 9.2: Overlapped Model Performance of A1 and D1

9.3 Best Model per Dataset

Since the average performance of each dataset has been confirmed to not be significantly different for the top-performing datasets, we now only looked at the maximum performance achieved per dataset.

We only considered the top 3 performing datasets, namely A1, B1, and D1.

Table 9.2 Top Model Performance for Dataset A1 B1 and D1

Dataset	**Model**	**Accuracy**	**Precision**	**Recall**	**F1**
A1	xgboost	0.8400	0.8800	0.8148	0.8462
B1	xgboost	0.8400	0.8800	0.8148	0.8462
D1	xgboost	0.8450	0.8400	0.8485	0.8442

We found that XGBoost, a Gaussian Ensemble method for decision trees, performed best amongst all models according to FLAML.
Comparing Individual Metrics per Dataset we find that

Table 9.3 Top Model Performance Per Metric over A1 B1 and D1

Dataset	**Best Dataset**	**Value**
Accuracy	D1	84.5%
Precision	A1,B1	88.00%
Recall	D1	84.85%
F1 - Score	A1,B1	84.62%

We found that XGBoost trained on dataset D1 had the best accuracy among all of the models and had the highest recall score. However, XGBoost trained on either dataset A1 or B1 had the best Precision and F1 score.

10 Conclusions

By consolidating all evaluations and results we have now formed our conclusions

We would like to inform about the limitations of the project before presenting our conclusions

10.1 Limitations

Our project encountered several limitations, including:

- Dataset Records are 244
  With only 244 records, our dataset's size was relatively small, potentially affecting the robustness of our models.
- Artificial Dataset
  The dataset we generated through individual queries may not fully reflect real-world scenarios, introducing potential biases.
- Too many Features dropped
  Due to the artificial nature of the dataset, we had to remove many variables during cleaning and exploration, potentially discarding important features.
- Only “Find” queries
  The query dataset used for generating log files consisted exclusively of "find" queries, limiting the model's applicability to this specific type of query.

10.2 Final Conclusion

Our conclusions are as follows, categorized according to our research questions:

1. We confirmed the feasibility of constructing models based on MongoDB log file data.
    a. The best models achieved an accuracy of 84.50%, precision of 88.00%, recall of 84.85%, and F1 score of 84.62%.
2. Our analysis revealed various types of data in MongoDB log files, including raw query text and machine statistics.
    a. The log files of MongoDB provide various types of data such as
        i. The Raw Text of the Query sent and the filter and conditions
        ii. Statistics of the Machine after processing the query
    b. The significant variables found in the dataset were
        i. The Text data of the filter that was used in the "find" query
        ii. The planning time in Microseconds is used to plan the execution of the query
    c. We engineered more features based on the text data of the filters these included
        i. Dummy Variables which indicate the presence of particular operators present in the filter
        ii. The length of the filter sent
        iii. Out of these variables, some of the Dummy Variables were deemed significant enough to be used for the training dataset
    d. A Separability analysis concluded that:
        i. The data is not, visually, significant enough to be linearly separable however it is noted that some malicious injection queries do form distinct groups. However, both benign and malicious queries are mixed and are not separable
        ii. The data is separable non-linearly, however, we have noticed clusters of data being completely overlapped. We have noticed around 81.57% of the data forms distinct groups

3. We formulated models and conducted a comparative study
    a. Out of the top 3 dataset configurations we found no significant difference in the average performance of a random classifier
    b. With the comparison of top models per top 3 datasets, we found that the top model for all 3 was XGBoost
    c. Comparing individual performance metrics we found that
        i. The Model trained and tested on dataset D1 had the highest accuracy of 84.5% and a recall value of 88% compared to Datasets A1, B1 which had an accuracy of only 84% and recall of 81.48%. This suggests that the Model trained on D1 had the highest accuracies and a higher number of malicious queries correctly identified.
        ii. The Model trained on datasets A1, and B1 had the highest precision of 88.00% and an F1 score of 84.62% compared to dataset D1 which had a precision of 84.00% and an F1 score of 84.42%. This suggests that the Model trained on Datasets A1, and B1 had a higher number of true positive malicious queries identified and a balance of true positive malicious queries better than Models trained on Dataset D1

10.3 Future Work

A Future researcher in this topic may explore the following topics to extend work in this field

- Exploratory Data analysis of a Real-life MongoDB log file or any NoSQL/SQL database system log file to identify significant variables to classify injection queries
- Using a similar study as the above construction of subsets of training datasets to compare the performance of the average classification model when some variables are omitted/added
- Application of such models in a real database system operation

In conclusion, our project demonstrates the potential for using machine learning models to detect injection queries in MongoDB log files. While we encountered limitations, such as dataset size and artificiality, our findings provide valuable insights into model performance and feature importance. Further research and refinement could enhance the applicability and effectiveness of such models in real-world scenarios.

References

[1] D. Barbucha, N. T. Nguyen, and J. Batubara, *New Trends in Intelligent Information and Database Systems*, vol. 598. 2015. doi: 10.1007/978-3-319-16211-9.

[2] D. D. Chamberlin, "Early History of SQL," *IEEE Ann. Hist. Comput.*, vol. 34, no. 4, pp. 78–82, Oct. 2012, doi: 10.1109/MAHC.2012.61.

[3] D. R. Deutsch, "The SQL StandarD: How it Happened," *IEEE Ann. Hist. Comput.*, vol. 35, no. 2, pp. 72–75, Apr. 2013, doi: 10.1109/MAHC.2013.30.

[4] J. Clarke-Salt, *SQL Injection Attacks and Defense*. Elsevier, 2009.

[5] A.-G. Babucea, "SQL OR NoSQL DATABASES? CRITICAL DIFFERENCES," no. 1.

[6] A. Ron, A. Shulman-Peleg, and E. Bronshtein, "No SQL, No Injection? Examining NoSQL Security." arXiv, Jun. 12, 2015. doi: 10.48550/arXiv.1506.04082.

[7] C. Strauch, "NoSQL Databases." Accessed: May 03, 2024. [Online]. Available: https://scholar.googleusercontent.com/scholar?q=cache:p6ttZi1mFaAJ:scholar.google.com/+nosql+databases+christof+strauch&hl=en&as_sdt=0,5

[8] H. I. Mejia-Cabrera, D. Paico-Chileno, J. H. Valdera-Contreras, V. A. Tuesta-Monteza, and M. G. Forero, "Automatic Detection of Injection Attacks by Machine Learning in NoSQL Databases," in *Pattern Recognition*, E. Roman-Rangel, Á. F. Kuri-Morales, J. F. Martínez-Trinidad, J. A. Carrasco-Ochoa, and J. A. Olvera-López, Eds., Cham: Springer International Publishing, 2021, pp. 23–32. doi: 10.1007/978-3-030-77004-4_3.

[9] M. R. Ul Islam, Md. S. Islam, Z. Ahmed, A. Iqbal, and R. Shahriyar, "Automatic Detection of NoSQL Injection Using Supervised Learning," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Jul. 2019, pp. 760–769. doi: 10.1109/COMPSAC.2019.00113.

[10] J. Hu, W. Zhao, and Y. Cui, "A Survey on SQL Injection Attacks, Detection and Prevention," in *Proceedings of the 2020 12th International Conference on Machine Learning and Computing*, in ICMLC '20. New York, NY, USA: Association for Computing Machinery, May 2020, pp. 483–488. doi: 10.1145/3383972.3384028.

[11] "LsSQLIDP: Literature survey on SQL injection detection and prevention techniques: Journal of Statistics and Management Systems: Vol 22, No 2." Accessed: May 02, 2024. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/09720510.2019.1580904

[12] M. Shachi, N. Shourav, A. S. Sajid Ahmed, A. Brishty, and N. Sakib, "A Survey on Detection and Prevention of SQL and NoSQL Injection Attack on Server-side Applications," *Int. J. Comput. Appl.*, vol. 183, pp. 1–7, Jun. 2021, doi: 10.5120/ijca2021921396.

[13] Y. Divinsky, "CVE-2023-34362: MOVEIt Transfer zero-day vulnerability exploited in the wild," Vulcan Cyber. Accessed: May 03, 2024. [Online]. Available: https://vulcan.io/blog/cve-2023-34362-moveit-transfer-zero-day-vulnerability-exploited-in-the-wild/

[14] "MOVEit transfer data breaches Deep Dive | ORX News." Accessed: May 03, 2024.

[Online]. Available: https://orx.org/resource/moveit-transfer-data-breaches
[15] "23andMe data theft, MGM's $100 million ransomware loss and the Azure VM breach." Accessed: May 03, 2024. [Online]. Available: https://www.infosecinstitute.com/resources/news/23andme-data-theft-mgms-$100-million-ransomware-loss-and-the-azure-vm-breach/
[16] "ICMR Data Leak Due to 'Cybersecurity Vulnerability'; Logs Under Scanner, Security Audit Suggested | Exclusive," News18. Accessed: May 03, 2024. [Online]. Available: https://www.news18.com/india/icmr-data-leak-cybersecurity-security-audit-8709609.html
[17] M. Rua, Thiyab, D. Musab, A. Ali, F. Abdulqader, and Abdulqader, "THE IMPACT OF SQL INJECTION ATTACKS ON THE SECURITY OF DATABASES," Apr. 2017.
[18] M. Hasan, Z. Balbahaith, and M. Tarique, "Detection of SQL Injection Attacks: A Machine Learning Approach," in *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, Nov. 2019, pp. 1–6. doi: 10.1109/ICECTA48151.2019.8959617.
[19] E. Hosam, H. Hosny, W. Ashraf, and A. S. Kaseb, "SQL Injection Detection Using Machine Learning Techniques," in *2021 8th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, Nov. 2021, pp. 15–20. doi: 10.1109/ISCMI53840.2021.9654820.
[20] K. Ross, "SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources," *Masters Proj.*, Apr. 2018, doi: https://doi.org/10.31979/etd.zknb-4z36.
[21] P. Roy, R. Kumar, and P. Rani, "SQL Injection Attack Detection by Machine Learning Classifier," in *2022 International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*, May 2022, pp. 394–400. doi: 10.1109/ICAAIC53929.2022.9792964.
[22] Sangeeta, S. Nagasundari, and P. B. Honnavali, "SQL Injection Attack Detection using ResNet," in *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, Jul. 2019, pp. 1–7. doi: 10.1109/ICCCNT45670.2019.8944874.
[23] A. S. Sajid Ahmed, M. Shachi, and N. Shourav, "A Hybrid Approach to Detect Injection Attacks on Server-side Applications using Data Mining Techniques," 2022. doi: 10.13140/RG.2.2.27421.18407.
[24] D. Van Landuyt, V. Wijshoff, and W. Joosen, "A study of NoSQL query injection in Neo4j," *Comput. Secur.*, vol. 137, p. 103590, Feb. 2024, doi: 10.1016/j.cose.2023.103590.
[25] A. M. Weeratunga, "Identification of NoSQL Injection Vulnerabilities in MongoDB based Web Applications," Thesis, 2021. Accessed: May 02, 2024. [Online]. Available: https://dl.ucsc.cmb.ac.lk/jspui/handle/123456789/4474
[26] V. Sachdeva and S. Gupta, "Basic NOSQL Injection Analysis And Detection On MongoDB," in *2018 International Conference on Advanced Computation and Telecommunication (ICACAT)*, Dec. 2018, pp. 1–5. doi: 10.1109/ICACAT.2018.8933707.
[27] S. Joseph and K. P. Jevitha, "An Automata Based Approach for the Prevention of NoSQL Injections," in *Security in Computing and Communications*, J. H. Abawajy, S.

Mukherjea, S. M. Thampi, and A. Ruiz-Martínez, Eds., Cham: Springer International Publishing, 2015, pp. 538–546. doi: 10.1007/978-3-319-22915-7_49.
[28] S. Praveen, A. Dcouth, and A. S. Mahesh, "NoSQL Injection Detection Using Supervised Text Classification," in *2022 2nd International Conference on Intelligent Technologies (CONIT)*, Jun. 2022, pp. 1–5. doi: 10.1109/CONIT55038.2022.9848017.
[29] A. M. Eassa, M. Elhoseny, H. M. El-Bakry, and A. S. Salama, "NoSQL Injection Attack Detection in Web Applications Using RESTful Service," *Program. Comput. Softw.*, vol. 44, no. 6, pp. 435–444, Nov. 2018, doi: 10.1134/S036176881901002X.
[30] A. M., O. H., H. M., and A. S., "NoSQL Racket: A Testing Tool for Detecting NoSQL Injection Attacks in Web Applications," *Int. J. Adv. Comput. Sci. Appl.*, vol. 8, no. 11, 2017, doi: 10.14569/IJACSA.2017.081178.
[31] R. D· l·, A. H· v·, P. B. Honnavalli, and N. S·, "The MongoDB injection dataset: A comprehensive collection of MongoDB - NoSQL injection attempts and vulnerabilities," *Data Brief*, vol. 54, p. 110289, Jun. 2024, doi: 10.1016/j.dib.2024.110289.
[32] "What is MongoDB? - MongoDB Manual v6.0." Accessed: May 02, 2024. [Online]. Available: https://www.mongodb.com/docs/v6.0/
[33] "pandas - Python Data Analysis Library." Accessed: May 02, 2024. [Online]. Available: https://pandas.pydata.org/
[34] S. Arora, W. Hu, and P. K. Kothari, "An Analysis of the t-SNE Algorithm for Data Visualization," in *Proceedings of the 31st Conference On Learning Theory*, PMLR, Jul. 2018, pp. 1455–1462. Accessed: May 02, 2024. [Online]. Available: https://proceedings.mlr.press/v75/arora18a.html
[35] T. ArchanaH. and D. Sachin, "Dimensionality Reduction and Classification through PCA and LDA," *Int. J. Comput. Appl.*, vol. 122, no. 17, pp. 4–8, Jul. 2015, doi: 10.5120/21790-5104.
[36] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (PCA)," *Comput. Geosci.*, vol. 19, no. 3, pp. 303–342, Mar. 1993, doi: 10.1016/0098-3004(93)90090-R.
[37] "Linear Discriminant Analysis | SpringerLink." Accessed: May 02, 2024. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4419-9878-1_4
[38] S. Suthaharan, "Support Vector Machine," in *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*, S. Suthaharan, Ed., Boston, MA: Springer US, 2016, pp. 207–235. doi: 10.1007/978-1-4899-7641-3_9.
[39] S. Suthaharan, "Decision Tree Learning," in *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*, S. Suthaharan, Ed., Boston, MA: Springer US, 2016, pp. 237–269. doi: 10.1007/978-1-4899-7641-3_10.
[40] "AutoML & Tuning | FLAML." Accessed: May 02, 2024. [Online]. Available: https://microsoft.github.io//FLAML/
[41] "Study and Comparision of Vectorization Techniques Used in Text Classification | IEEE Conference Publication | IEEE Xplore." Accessed: May 02, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9984608

Appendix

Appendix - 1 Sample Log Entry


```
{
  "t": {
    "$date": "2024-04-04T20:37:10.296+05:30"
  },
  "s": "I",
  "c": "COMMAND",
  "id": 51803,
  "ctx": "conn10",
  "msg": "Slow query",
  "attr": {
    "type": "command",
    "ns": "test_database.test_collection",
    "command": {
      "find": "test_collection",
      "filter": {
        "user": "{}",
        "password": "{}"
      },
      "lsid": {
        "id": {
          "$uuid": "e5e23d7e-5367-4829-a104-ca369c49d4dc"
        }
      },
      "$db": "test_database"
    },
    "planSummary": "EOF",
    "planningTimeMicros": 83,
    "keysExamined": 0,
    "docsExamined": 0,
    "nBatches": 1,
    "cursorExhausted": true,
    "numYields": 0,
    "nreturned": 0,
    "queryFramework": "classic",
```

```
    "reslen": 118,
    "locks": {
      "FeatureCompatibilityVersion": {
        "acquireCount": {
          "r": 1
        }
      },
      "Global": {
        "acquireCount": {
          "r": 1
        }
      }
    },
    "storage": {},
    "cpuNanos": 176371,
    "remote": "127.0.0.1:47038",
    "protocol": "op_msg",
    "durationMillis": 0
  }
}
```


Figure A10.3.1: Log Entry Sample

Appendix - 2 Training Data Sample

Table A2: Sample Rows of Training Dataset

Filter	DN	WD	RD	LO	TP	MO	PTM	label
{'username': {'$ne': '{}'}}	{'': {'$ne': '{}'}}	0	0	1	0	0	42	1
{'username': {'$ne': '{}'}}	{'': {'$ne': '{}'}}	0	0	1	0	0	42	1
{'username': {'$gt': '{}'}}	{'': {'$gt': '{}'}}	0	0	1	0	0	46	1
{'username': {'$gt': '{}'}}	{'': {'$gt': '{}'}}	0	0	1	0	0	46	1

Appendix - 3 Links

Repository - https://github.com/ShaunakPerniUniGoa/NoSQLInjectionDetection