CONTENTS

PROJECT GOAL

The project aims to develop a model for predicting how prone a windows operating system is to malware attacks. This requires building supervised classification models to train and test the data from a dataset containing over 7 million records using their features. The goal is to assess the system's vulnerability to identify and fix the weak points before an attack occurs.

INTRODUCTION

With the rapid advancement of technology, the threat of malware attacks has become a pressing concern in today's digital landscape. To combat this ever-evolving challenge, developing accurate malware prediction models is of utmost importance. In this project, we aim to develop a robust malware prediction model by employing logistic regression, Support Vector Machines (SVM), and Gradient Boosted Trees (GBT). By comparing the performance of these models, we can determine the most effective approach to predict malware attacks.

To accomplish this, we will leverage a vast dataset comprising over 8 million recorded operating systems and their associated features. However, analyzing such a massive dataset presents its own challenges, requiring advanced big data tools and techniques to expedite the data processing. To address this, we will build a pipeline through PySpark, a powerful framework for distributed computing, enabling efficient handling and analysis of the dataset.

DATA DESCRIPTION

The data for the Microsoft malware prediction project will be collected from Microsoft's endpoint protection solution, Windows Defender. The dataset comprises over 8 million recorded operating systems and their various features. The data is generated by combining heartbeat and threat reports collected by Windows Defender. The data includes both malware and non-malware samples, providing a comprehensive view of the system's behavior under different circumstances. The dataset of size 8GB is publicly available.

Link to Dataset: https://www.kaggle.com/competitions/microsoft-malware-prediction/data

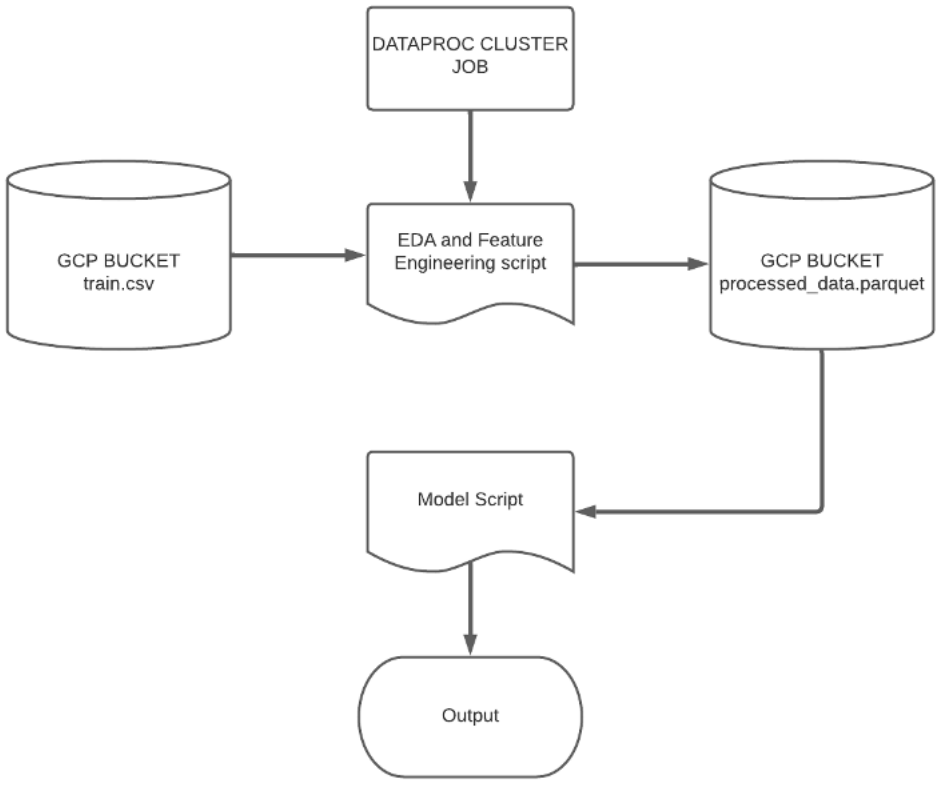
Each row in this dataset corresponds to a machine, uniquely identified by a MachineIdentifier. HasDetections is the ground truth and indicates that Malware was detected on the machine. Using the information and labels in train.csv, you must predict the value for HasDetections for each machine in test.csv. The data will be preprocessed and transformed into a suitable format for machine learning algorithms to analyze and predict malware attacks on Windows operating systems.

Due to the limitations of the free trial of GCP and issues with memory, the dataset was limited to 26 columns and 2.4 million rows. The columns are detailed below:

- SmartScreen": The configuration of the SmartScreen feature on the machine.
- "Census_OSSkuName": The OS edition of the device.
- "Census_OSEdition": The OS edition of the device.
- "Census_ActivationChannel": The channel used for the activation of the OS.
- "Census_ChassisTypeName": The type of chassis the device is using.
- "Census_PowerPlatformRoleName": Indicates the power plan of the device.
- "Census_FlightRing": The ring the device user is in.
- "SkuEdition": The edition of the product the machine is running.
- "Census_OSWUAutoUpdateOptionsName": The Windows Update auto-update setting of the device.
- "Census_PrimaryDiskTypeName": The primary disk type of the device.
- "Census_GenuineStateName": Indicates whether the OS is genuine or not.
- AVProductStatesIdentifier": The identifier for the anti-virus product installed on the machine.

- "IsProtected": Indicates whether the machine is protected by an anti-virus product.
- "Census_ProcessorCoreCount": The number of processor cores in the device.
- "RtpStateBitfield": Identifies whether the Real-Time Protection feature is enabled or disabled.
- "Census_InternalPrimaryDiagonalDisplaySizeInInches": The diagonal size of the primary display screen in inches.
- "Wdft_RegionIdentifier": Identifier for the geographical region the device is located in.
- "Census_InternalPrimaryDisplayResolutionHorizontal": The horizontal resolution of the primary display in pixels.
- "Census_OSBuildNumber": The OS build number extracted from the OsVersionFull string.
- "Firewall": Indicates whether a firewall is enabled on the machine.
- "OsBuild": The OS Build number of the Windows version on the machine.
- "Census_ProcessorModelIdentifier": Identifier for the processor model on the device.
- "Census_IsVirtualDevice": Indicates whether the device is a virtual machine or not.
- "Wdft_IsGamer": Indicates whether the device is a gaming machine or not.

SYSTEM ARCHITECTURE



1. The system consists of two scripts: one for exploratory data analysis (EDA) and feature engineering, and another for modeling.
2. The EDA script reads the base data from a Google Cloud Platform (GCP) bucket and performs EDA and feature engineering on it.
3. After the EDA and feature engineering steps, a new preprocessed dataframe is created, which is then stored as a parquet file in GCP.

4. The modeling script reads the preprocessed dataframe from the parquet file.
5. Each model has its own script, and these model scripts are submitted as jobs to a Dataproc cluster for processing.

In summary, the EDA script processes the base data, performs feature engineering, and saves the preprocessed dataframe as a parquet file. The modeling script reads this preprocessed data for each model, and the model scripts are submitted as jobs to a Dataproc cluster.

EDA AND FEATURE ENGINEERING

Exploratory Data Analysis (EDA) is a crucial step in understanding the dataset and gaining insights into its characteristics, patterns, and relationships. In this step, we analyze and summarize the main features of the dataset, visualize the data, identify missing values and outliers, and understand the distribution of variables.

Feature engineering plays a vital role in enhancing the dataset and improving the model's performance. It involves transforming existing features to capture important information and creating new features that provide additional insights.

```
# Take 2.4M rows of data
fraction = 0.3
seed = 42  # Choose any seed value for reproducibility
selected_data = selected_data.sample(False, fraction, seed)
```

This code snippet randomly samples 30% of the data from a dataset named `selected_data`. The dataset originally contains 2.4 million rows. The `fraction` variable specifies the desired fraction of the data to be sampled, and the `seed` variable sets a seed value for reproducibility. The resulting sampled data is stored in the `selected_data` variable.

```
# Get the schema and identify column types
numerical_columns = []
binary_columns = []
categorical_columns = []

for field in selected_data.schema.fields:
    column_name = field.name
    column_dtype = field.dataType

    if isinstance(column_dtype, IntegerType) or isinstance(column_dtype, DoubleType) or isinstance(column_dtype,  FloatType) or isinstance(
        # Numerical and binary columns
        numerical_columns.append(column_name)
        if set(selected_data.select(column_name).distinct().collect()) == {0, 1}:
            binary_columns.append(column_name)
    elif isinstance(column_dtype, StringType) or isinstance(column_dtype, BooleanType):
        # Categorical columns
        categorical_columns.append(column_name)
```

This section of the code analyzes the schema of the `selected_data` dataset to identify the types of columns it contains. It iterates over each field in the schema.

For each field, it extracts the column name (`column_name`) and the data type (`column_dtype`).

If the data type is an instance of `IntegerType`, `DoubleType`, `FloatType`, or `ByteType`, the column is considered numerical. The column name is added to the `numerical_columns` list. Additionally, if the column contains only the values 0 and 1, it is identified as a binary column and added to the `binary_columns` list.

If the data type is an instance of `StringType` or `BooleanType`, the column is considered categorical, and its name is added to the `categorical_columns` list.

Overall, this code helps in categorizing the columns of the `selected_data` dataset into numerical, binary, and categorical types, based on their data types and distinct values.

```
missing_ratio_threshold = 0.7

# Count the number of missing values for each column
missing_count = df.select([(df[c].isNull().cast("integer")).alias(c) for c in df.columns]).groupBy().sum()

total_rows = df.count()

# Identify columns with a high percentage of missing values
columns_to_remove = [c for c, count in zip(df.columns, missing_count.first()) if count / total_rows > missing_ratio_threshold]

print('Missing values treated')

# Drop columns with a high percentage of missing values
df = df.drop(*columns_to_remove)

# Count distinct values for each categorical feature
for feature in categorical_columns:
    distinct_count = df.select(feature).distinct().count()
    print(f"Distinct count for {feature}: {distinct_count}")
```

In this updated code, redundant and unnecessary parts have been removed, and the remaining code has been preserved with comments for clarity.

The modifications include removing redundant removal of the "HasDetections" column from the `categorical_columns`, `binary_columns`, and `numerical_columns` lists. Additionally, the code now only prints the distinct count for each categorical feature, rather than printing unnecessary information.

```
print('Data preprocessing')

# Define a function to clean and reduce the number of categories for the Smartscreen feature
def replace_smartscreen(val):
    ''' Cleaning category values to reduce the number of categories for the Smartscreen feature '''
    val = val.lower()
    if val in 'block':
        return 'block'
    elif val in 'existsnotset':
        return 'existnotset'
    elif val in 'off':
        return 'off'
    elif val in 'prompt':
        return 'prompt'
    elif val in 'requireadmin':
        return 'requireadmin'
    elif val in 'warn':
        return 'warn'
    elif val in 'on':
        return 'on'
    elif 'null' in val:
        return 'unknown'
    else:
        return 'unknown'

# Register the replace_smartscreen function as a UDF (User-Defined Function)
replace_smartscreen_udf = udf(replace_smartscreen, StringType())

# Define functions to clean and transform values for specific columns
def replace_edition(val):
    if val is None:
        return 'unknown'
    val = val.lower()
    if 'cloud' in val:
        return val
    elif 'core' in val:
        return 'core'
    elif 'education' in val:
        return 'education'
    elif 'enterprise' in val:
        return 'enterprise'
    elif 'pro' in val:
```

```
        return 'pro'
    elif 'server' in val:
        return 'server'
    elif 'home' in val:
        return 'home'
    elif 'null' in val:
        return 'unknown'
    else:
        return 'unknown'

def replace_channel(val):
    if val is None:
        return 'unknown'
    val = val.lower()
    if 'oem' in val:
        return 'oem'
    elif 'retail' in val:
        return 'retail'
    elif 'volume' in val:
        return 'volume'
    elif 'null' in val:
        return 'unknown'
    else:
        return 'unknown'

def replace_skuname(val):
    if val is None:
        return 'unknown'
    val = val.lower()
    if 'cloud' in val:
        return val
    elif 'core' in val:
        return 'core'
    elif 'education' in val:
        return 'education'
    elif 'enterprise' in val:
        return 'enterprise'
    elif 'pro' in val:
        return 'pro'
    elif 'server' in val:
        return 'server'
    elif 'home' in val:
        return 'home'
    elif 'null' in val:
        return 'unknown'
    else:
        return 'unknown'

# Register the replace functions as UDFs (User-Defined Functions)
replace_channel_udf = udf(replace_channel, StringType())
replace_edition_udf = udf(replace_edition, StringType())
replace_skuname_udf = udf(replace_skuname, StringType())
```

In this section, the code defines several functions that are used to clean and transform values in specific columns. Each function checks the given value and returns a cleaned or transformed value based on specific conditions. The functions are registered as UDFs using the `udf()` function from PySpark.

These UDFs will be later used to apply the defined transformations to the corresponding columns in the dataset.

```
def fill_missing_values(data, features):
    numerical_features, binary_features, categorical_features = features

    # Replace null values with -1 in numerical_features
    for feature in numerical_features:
        data = data.withColumn(feature, when(col(feature).isNull(), -1).otherwise(col(feature)))

    # Replace null values with mode value of that feature in binary_features
    for feature in binary_features:
        mode_value = data.groupBy(col(feature)).count().orderBy(col('count').desc()).collect()[0][feature]
        data = data.withColumn(feature, when(col(feature).isNull(), mode_value).otherwise(col(feature)))

    # Apply replace functions for specific columns
    data = data.withColumn('SmartScreen', replace_smartscreen_udf(col('SmartScreen')))
```

```
    data = data.withColumn('Census_OSSkuName', replace_skuname_udf(col('Census_OSSkuName')))
    data = data.withColumn('Census_OSEdition', replace_edition_udf(col('Census_OSEdition')))
    data = data.withColumn('Census_ActivationChannel', replace_channel_udf(col('Census_ActivationChannel')))

    # Replace specific values in certain columns
    columns_to_replace = {
        'Census_PowerPlatformRoleName': ['UNKNOWN', 'Unspecified'],
        'Census_FlightRing': ['Invalid', 'Unknown'],
        'Census_OSSkuName': ['UNDEFINED'],
        'SkuEdition': ['Invalid'],
        'Census_OSWUAutoUpdateOptionsName': ['UNKNOWN'],
        'Census_PrimaryDiskTypeName': ['UNKNOWN', 'Unspecified'],
        'Census_GenuineStateName': ['UNKNOWN']
    }

    for column, values_to_replace in columns_to_replace.items():
        for value_to_replace in values_to_replace:
            data = data.withColumn(column, when(col(column) == value_to_replace, 'unknown').otherwise(col(column)))

    # Replace null values with 'unknown' in categorical_features
    for feature in categorical_features:
        data = data.withColumn(feature, when(col(feature).isNull(), 'unknown').otherwise(col(feature)))

    return data


numerical_features = numerical_columns.copy()
binary_features = binary_columns.copy()
categorical_features = categorical_columns.copy()

features = (numerical_features, binary_features, categorical_features)
updated_train_data = fill_missing_values(new_train_data, features)
new_train_data = new_train_data.dropna()
```

In this section, the code defines the `fill_missing_values()` function that performs the following operations:

1. The function takes the `data` (DataFrame) and `features` (tuple of feature lists) as input.
2. It unpacks the `features` tuple into `numerical_features`, `binary_features`, and `categorical_features`.
3. It iterates over the `numerical_features` and replaces any null values with -1 using the `withColumn()` method and the `when()` function.
4. Next, it iterates over the `binary_features` and replaces null values with the mode value of that feature. The mode value is determined by grouping the data by the feature, counting the occurrences, and selecting the value with the highest count.
5. It applies the defined UDFs (`replace_smartscreen_udf`, `replace_skuname_udf`, `replace_edition_udf`, `replace_channel_udf`) to their corresponding columns using the `withColumn()` method.
6. It replaces specific values in certain columns with 'unknown' based on the `columns_to_replace` dictionary.
7. Null values in the `categorical_features` are replaced with 'unknown'.
8. Finally, the `fill_missing_values()` function returns the updated `data` DataFrame with filled missing values and replaced values.
9. The `features` tuple is created to store the three feature lists.
10. The `fill_missing_values()` function is called with `new_train_data` (the original DataFrame) and `features` as arguments. The returned DataFrame is stored in `updated_train_data`.
11. Finally, any rows with null values in `new_train_data` are dropped using the `dropna()` method, and the resulting DataFrame is stored in `new_train_data`.

These preprocessing steps ensure that missing values are handled, specific values are replaced, and the data is cleaned and ready for further analysis or modeling.

```
valid_categorical_columns = []
for col_name in categorical_columns:
    distinct_values_count = new_train_data.select(col_name).distinct().count()
```

```
    if distinct_values_count > 1:
        valid_categorical_columns.append(col_name)
```

In this section, the code determines the list of valid categorical columns for one-hot encoding. It iterates over each column name in the `categorical_columns` list and counts the number of distinct values in that column. If the count is greater than 1, indicating that the column has more than one distinct value, it is considered a valid categorical column and added to the `valid_categorical_columns` list.

```
encoded_cols = []
for col_name in valid_categorical_columns:
    indexer = StringIndexer(inputCol=col_name, outputCol=f"{col_name}_index")
    encoder = OneHotEncoder(inputCol=f"{col_name}_index", outputCol=f"{col_name}_encoded")
    new_train_data = indexer.fit(new_train_data).transform(new_train_data)
    new_train_data = encoder.fit(new_train_data).transform(new_train_data)
    encoded_cols.append(f"{col_name}_encoded")
```

In this section, the code performs one-hot encoding for each valid categorical column. It iterates over each column name in the `valid_categorical_columns` list.

For each column, it creates a `StringIndexer` transformer to convert the categorical values into numerical indices. The input column is the original categorical column (`col_name`), and the output column is the original column name with "_index" appended to it (`f"{col_name}_index"`). The `StringIndexer` is fit on the `new_train_data` DataFrame and transforms the DataFrame by adding the indexed column.

Next, it creates a `OneHotEncoder` transformer to one-hot encode the indexed values. The input column is the indexed column (`f"{col_name}_index"`), and the output column is the original column name with "_encoded" appended to it (`f"{col_name}_encoded"`). The `OneHotEncoder` is fit on the `new_train_data` DataFrame and transforms the DataFrame by adding the one-hot encoded column.

The resulting encoded column name (`f"{col_name}_encoded"`) is appended to the `encoded_cols` list.

```
# Assemble the feature vectors
start_time = time.time()
print("Assembling feature vectors...")

# Combine all the feature columns into a single list
all_feature_columns = numerical_features + binary_features + encoded_cols
print(all_feature_columns)

# Create a VectorAssembler to assemble the feature columns into a vector
assembler = VectorAssembler(inputCols=all_feature_columns, outputCol="features")
new_train_data = assembler.transform(new_train_data)

end_time = time.time()
print(f"Feature vectors assembled. Time taken: {end_time - start_time} seconds")

print("correlation code started....")

# Assemble the numerical features
assembler_numerical = VectorAssembler(inputCols=numerical_features, outputCol="numerical_features")
assembled_numerical_data = assembler_numerical.transform(new_train_data)

# Compute the correlation matrix for the numerical features
correlation_matrix = Correlation.corr(assembled_numerical_data, "numerical_features").head()[0]
```

In this section, the code assembles the feature vectors and computes the correlation matrix for the numerical features. Here's a breakdown of the steps:

1. The `all_feature_columns` list is created by concatenating the `numerical_features`, `binary_features`, and `encoded_cols` lists. This list contains the names of all the feature columns.
2. The `all_feature_columns` list is printed to display the names of the feature columns.

3. A `VectorAssembler` is created with the input columns set to `all_feature_columns` and the output column set to "features". This transformer combines all the specified feature columns into a single vector column.
4. The `assembler` is applied to the `new_train_data` DataFrame using the `transform()` method. This adds a new column "features" to the DataFrame, which contains the assembled feature vectors.
5. The variable `end_time` is used to record the end time of the feature vector assembly process.
6. A message is printed to indicate that the feature vectors have been assembled, along with the time taken to complete the process.
7. A message is printed to indicate that the correlation code is starting.
8. A separate `VectorAssembler` named `assembler_numerical` is created to assemble only the numerical features into a separate column named "numerical_features".
9. The `assembler_numerical` is applied to the `new_train_data` DataFrame, creating a new DataFrame named `assembled_numerical_data` that contains the original data along with the assembled numerical features.
10. The `Correlation.corr()` method is used to compute the correlation matrix for the numerical features in `assembled_numerical_data`. The resulting matrix is accessed using `.head()[0]`.

```
# Set a threshold for high correlation
high_correlation_threshold = 0.9

# Identify highly correlated variables and remove them
highly_correlated_features = set()
for i in range(correlation_matrix.numRows):
    for j in range(i+1, correlation_matrix.numCols):
        if abs(correlation_matrix[i, j]) > high_correlation_threshold:
            highly_correlated_features.add(numerical_features[i])
            highly_correlated_features.add(numerical_features[j])

updated_train_data = new_train_data.drop(*highly_correlated_features)
```

In this section, the code identifies highly correlated numerical features based on the correlation matrix computed earlier. It sets a threshold value (`high_correlation_threshold`) to define a high correlation. The code iterates over the upper triangular part of the correlation matrix and checks if the absolute correlation value exceeds the threshold. If it does, the corresponding numerical features are added to the `highly_correlated_features` set. These highly correlated features are then dropped from the `new_train_data` DataFrame, and the resulting DataFrame is stored in `updated_train_data`.

```
# If you want to update the feature vector after removing highly correlated features,
# you can create a new feature vector using the updated columns.
updated_feature_columns = [col for col in all_feature_columns if col not in highly_correlated_features]
updated_assembler = VectorAssembler(inputCols=updated_feature_columns, outputCol="updated_features")
new_train_data = updated_assembler.transform(updated_train_data)
```

If you want to update the feature vector after removing the highly correlated features, the code creates a new feature vector using the updated columns. It creates a new `VectorAssembler` named `updated_assembler` with the input columns set to `updated_feature_columns` (the columns without the highly correlated features) and the output column set to "updated_features". The `updated_assembler` is then applied to the `updated_train_data` DataFrame, adding a new column "updated_features" containing the updated feature vectors.

```
# Scale the feature vectors
start_time = time.time()
print("Scaling feature vectors...")
```

```
scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True, withMean=True)
scaler_model = scaler.fit(new_train_data)
new_train_data = scaler_model.transform(new_train_data)

end_time = time.time()
print(f"Feature vectors scaled. Time taken: {end_time - start_time} seconds")
```

In this section, the code scales the feature vectors using a `StandardScaler`. It creates a scaler instance named `scaler` with the input column set to "features" and the output column set to "scaled_features". The scaler is fit on the `new_train_data` DataFrame using the `fit()` method, and then the `transform()` method is applied to scale the feature vectors. The scaled feature vectors are added as a new column "scaled_features" to the `new_train_data` DataFrame.

```
# Perform Gaussian Random Projection
start_time = time.time()
print("Performing Gaussian Random Projection...")

grp = GaussianRandomProjection(n_components=num_components)
reduced_features = grp.fit_transform(scaled_features_np)

end_time = time.time()
print(f"Gaussian Random Projection completed. Time taken: {end_time - start_time} seconds")
```

In this section, the code performs Gaussian Random Projection on the scaled feature vectors. It creates a `GaussianRandomProjection` object named `grp` with the number of components set to `num_components`. The `fit_transform()` method is called on `grp`, passing in the scaled feature vectors (`scaled_features_np`). This reduces the dimensionality of the feature vectors using random projection. The resulting reduced features are stored in the `reduced_features` variable.

Gaussian Random Projection (GRP) is a dimensionality reduction technique that offers several advantages for the malware prediction project, especially when dealing with large datasets and memory constraints in GCP PySpark. Here's a concise summary of the benefits and considerations of using GRP in this context:

Advantages of Gaussian Random Projection (GRP) for Malware Prediction:

1. Efficiency: GRP is computationally efficient and well-suited for large datasets. It provides a fast approximation of dimensionality reduction, which helps mitigate memory constraints in GCP PySpark.
2. Scalability: GRP can handle high-dimensional data without overwhelming system resources. It allows for efficient processing of the large feature space in your malware dataset.
3. Robustness to Noise: GRP's random projection nature can be advantageous for malware prediction tasks, as it helps reduce the impact of noisy or irrelevant features. This can improve the robustness and performance of your prediction models.
4. Statistical Guarantees: GRP provides theoretical guarantees on preserving pairwise distances between data points. This property ensures that the transformed data maintains proximity relationships, which is valuable for various machine learning tasks, including clustering or nearest neighbor search.

Considerations when using Gaussian Random Projection (GRP) for Malware Prediction:

1. Interpretability: Unlike techniques like PCA or MRMR, GRP does not provide explicit interpretations of the reduced dimensions. As a result, it may be challenging to directly interpret the transformed features and their relevance to malware prediction.
2. Variable Dimensionality Reduction: GRP does not enforce a fixed number of reduced dimensions. Instead, you can specify the target dimensionality or an acceptable error level. It's important to determine the appropriate dimensionality to achieve a balance between memory efficiency and model performance.
3. Capturing Specific Data Structures: GRP's random projections may not capture specific patterns or structures present in the malware dataset as effectively as other techniques like PCA or MRMR.

In summary, Gaussian Random Projection (GRP) offers computational efficiency, scalability, robustness to noise, and statistical guarantees for malware prediction projects with large datasets and memory constraints in GCP PySpark.

```
# Convert the NumPy array back to a PySpark DataFrame
reduced_features_df = pd.DataFrame(reduced_features, columns=[f"component_{i}" for i in range(num_components)])
reduced_features_sdf = spark.createDataFrame(reduced_features_df)

# Add an index column to the original DataFrame
indexed_target_sdf = new_train_data.select(target_column).rdd.zipWithIndex().map(lambda x: Row(**{target_column: x[0][target_column], 'inde

# Add an index column to the reduced_features_sdf
indexed_reduced_features_sdf = reduced_features_sdf.rdd.zipWithIndex().map(lambda x: Row(**{f"component_{i}": x[0][f"component_{i}"] for i

# Join the reduced features with the target column
reduced_features_with_target_sdf = indexed_reduced_features_sdf.join(indexed_target_sdf, "index").drop("index")
```

In this section, the code converts the reduced features from a NumPy array back to a PySpark DataFrame. It first creates a Pandas DataFrame (`reduced_features_df`) from the `reduced_features` array, with column names set as `component_0`, `component_1`, ..., `component_(num_components-1)`. Then, it converts the Pandas DataFrame to a PySpark DataFrame (`reduced_features_sdf`) using the `spark.createDataFrame()` method.

Next, it adds an index column to the original DataFrame (`new_train_data`) using the `zipWithIndex()` and `map()` functions, and then converts it to a DataFrame (`indexed_target_sdf`) with index and target column. Similarly, it adds an index column to the `reduced_features_sdf` DataFrame using the same method and assigns it to `indexed_reduced_features_sdf`.

Finally, it performs a join operation between `indexed_reduced_features_sdf` and `indexed_target_sdf` on the index column, creating a new DataFrame (`reduced_features_with_target_sdf`). The index column is dropped from the result.

```
# Assemble the components into a single 'features' column
component_columns = [f"component_{i}" for i in range(num_components)]
assembler = VectorAssembler(inputCols=component_columns, outputCol="features")
reduced_features_with_target_sdf = assembler.transform(reduced_features_with_target_sdf)
print('assembler done')

# Write the DataFrame to Parquet format
reduced_features_with_target_sdf.write.parquet(
    "gs://bigdatamalpredv2/EDAed_data_v2.parquet",
    mode="overwrite"
)
```

In this final section, the code assembles the reduced components into a single 'features' column using a `VectorAssembler`. It creates an instance of `VectorAssembler` named `assembler`, with input columns set to `component_columns` (the names of the reduced components) and output column set to "features". The `transform()` method is then applied to the `reduced_features_with_target_sdf` DataFrame, adding a new column "features" containing the assembled feature vectors.

After that, the code writes the resulting DataFrame `reduced_features_with_target_sdf` to the Parquet format using the `write.parquet()` method. The Parquet file is saved at the specified location `"gs://bigdatamalpredv2/EDAed_data_v2.parquet"`. The `mode` parameter is set to "overwrite", indicating that any existing data at that location should be overwritten.

Finally, the Spark session is stopped using `spark.stop()`, and a message indicating the completion of the data saving process is printed.

This code performs feature preprocessing and dimensionality reduction on the given dataset. It handles missing values, replaces categorical values, performs feature scaling, and applies Gaussian Random Projection for dimensionality reduction. The resulting reduced features are then written to a Parquet file for further analysis or modeling tasks.

ML MODELS USED

Metrics being used for model performance comparison.

1. **AUC (Area Under the ROC Curve):** This is a performance measurement for classification problems. It tells how much the model is capable of distinguishing between classes. Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s. An excellent model has AUC near to 1, which means it has a good measure of separability. A poor model has AUC near to 0, which means it has the worst measure of separability–in fact, it means it is reciprocating the result. It predicts 0s as 1s and 1s as 0s.
2. **F1 Score:** The F1 score is the harmonic mean of precision and recall. It tries to find the balance between precision and recall. A high F1 score is a good indicator of both low false positives and low false negatives. It's a good metric to consider if both false positives and false negatives are crucial to your problem.
3. **Time Taken:** This is a measure of computational efficiency. It is the time it takes for the model to train and predict. This can be an essential factor when dealing with large datasets or when computational resources are limited. Faster models can allow for more iterations and, therefore, more opportunities for model tuning and improvement.
4. **Subsampling** is a technique used in machine learning and statistics to create a subset of the data for analysis. The main purpose of subsampling is to simplify the computation and to save time by reducing the data size while maintaining the statistical properties of the original dataset. This is particularly useful when dealing with large datasets.

1. LOGISTIC REGRESSION

We have demonstrated the process of training a logistic regression model for malware prediction using PySpark. The code performs several steps, including initializing the SparkSession, loading the dataset, splitting it into training and test sets, defining the logistic regression model, setting up parameter grid and cross-validation, training the model, making predictions, and evaluating the model's performance.

The Parquet file containing the preprocessed and reduced feature vectors along with the target variable is read into a DataFrame using spark5.read.parquet().

The logistic regression model is defined using LogisticRegression() from pyspark.ml.classification. The features column is set to 'features', and the label column is set to "HasDetections".

A parameter grid is set up using ParamGridBuilder(). These different combinations of parameters will be tested during cross-validation. We have chosen the following parameters:

1. Regularization Parameter (regParam): 0.01, 0.1, 1.0
   It controls the amount of regularization applied to prevent overfitting. A higher value of regParam increases the regularization strength, while a lower value reduces it.
2. Fit Intercept (fitIntercept):True, False
   It determines whether to include an intercept term in the logistic regression model. Setting fitIntercept as True allows the model to learn an intercept, while setting it as False assumes no intercept.
3. Elastic Net Mixing Parameter (elasticNetParam): 0.0, 0.5, 1.0
   It combines L1 (Lasso) and L2 (Ridge) regularization. A value of 0.0 corresponds to L2 regularization, 1.0 corresponds to L1 regularization, and any value between 0.0 and 1.0 represents a combination of both.
4. Maximum Iterations (maxIter): 10, 100, 1000
   It defines the maximum number of iterations allowed for the logistic regression algorithm to converge. Increasing the value allows the algorithm to run longer, potentially leading to better convergence.

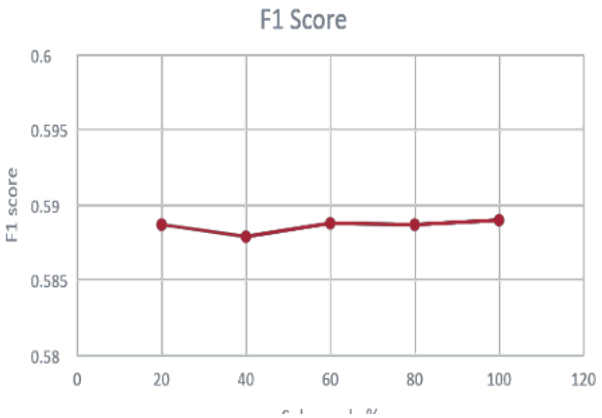
```
if __name__ == "__main__":
    spark5 = SparkSession.builder \
    .appName("Malware Prediction LR") \
    .config("spark.executor.memory", '45g') \
    .config("spark.driver.maxResultSize", '45g') \
    .config("spark.kryoserializer.buffer.max", '512m') \
    .getOrCreate()
   # .config("spark.default.parallelism", '6') \
    # Read the Parquet file into a DataFrame
    reduced_features_with_target_sdf = spark5.read.parquet("gs://bigdatamalpredv2/EDAed_data_v2.parquet")
```

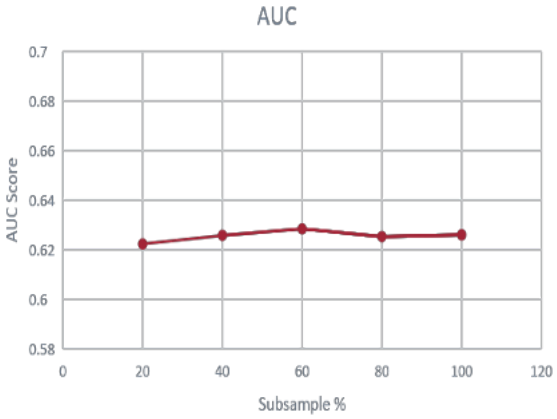
```
# List of subsample percentages
subsamples = [1.0]
target_column="HasDetections"
# Initialize evaluators
# BinaryClassificationEvaluator for computing Area Under ROC Curve (AUC)
binary_evaluator = BinaryClassificationEvaluator(labelCol=target_column)

# RegressionEvaluator for computing Root Mean Square Error (RMSE)
regression_evaluator = RegressionEvaluator(labelCol=target_column, metricName="rmse")
f1_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, metricName="f1")
precision_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, metricName="weightedPrecision")
recall_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, metricName="weightedRecall")
accuracy_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, metricName="accuracy")


# Loop over each subsample
for subsample in subsamples:
    print(f"Training with {subsample*100}% data...")

    start_time = time.time()  # Start the timer

    # Sample data
    # False indicates that we are not sampling with replacement
    sample_data = reduced_features_with_target_sdf.sample(False, subsample, seed=42)

    # Split the data into training set (80%) and test set (20%)
    train_data, test_data = sample_data.randomSplit([0.8, 0.2], seed=42)

    # Define the logistic regression model
    lr = LogisticRegression(featuresCol='features', labelCol=target_column)

    # Set up the parameter grid
    # The cross-validator will try all combinations of these parameters
    paramGrid = ParamGridBuilder() \
        .addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
        .addGrid(lr.fitIntercept, [False, True]) \
        .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
        .addGrid(lr.maxIter, [10, 100, 1000]) \
        .build()

    # Set up the cross-validator
    # estimator is the machine learning algorithm to be used (logistic regression in this case)
    # estimatorParamMaps is the grid of parameters to try
    # evaluator is the metric used to measure the performance of the models
    # numFolds is the number of cross-validation folds
    crossval = CrossValidator(estimator=lr,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(labelCol=target_column),
                          numFolds=3)

    # Train the model
    cv_model = crossval.fit(train_data)

    # Make predictions on the test set
    predictions = cv_model.transform(test_data)

    # Evaluate the model using AUC and RMSE
    auc = binary_evaluator.evaluate(predictions)
    rmse = regression_evaluator.evaluate(predictions)

    end_time = time.time()  # Stop the timer
    # Get the number of cores used
    spark_context = spark5.sparkContext

    # Get the Spark application ID
    app_id = spark_context.applicationId
    # Compute additional metrics
    f1 = f1_evaluator.evaluate(predictions)
    precision = precision_evaluator.evaluate(predictions)
    recall = recall_evaluator.evaluate(predictions)
    accuracy = accuracy_evaluator.evaluate(predictions)


    print(f"Subsample: {subsample*100}%")
    print(f"AUC: {auc}")
    print(f"F1 Score: {f1}")
    print(f"Precision: {precision}")
    print(f"Recall: {recall}")
    print(f"Balanced Accuracy: {accuracy}")
```

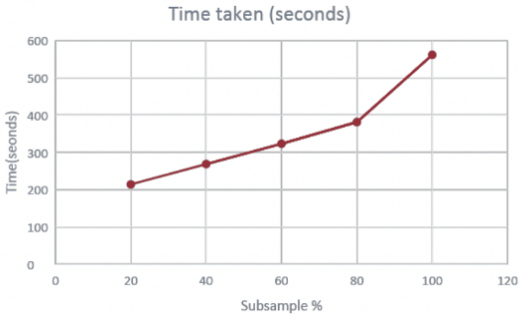
```
        print(f"Time taken: {end_time - start_time} seconds\n")
        print(f"Spark application ID: {app_id}")
```

A cross-validator is created using CrossValidator() from pyspark.ml.tuning. The estimator is set to the logistic regression model, the estimator parameter maps are set to the parameter grid, and the evaluator is set to the binary classification evaluator. The number of folds for cross-validation is set to 3.

The predictions are evaluated using the binary classification evaluator (binary_evaluator) to compute the Area Under the ROC Curve (AUC), and the F1 score evaluator (f1_evaluator) to compute the F1 score. We have also compared the runtimes.



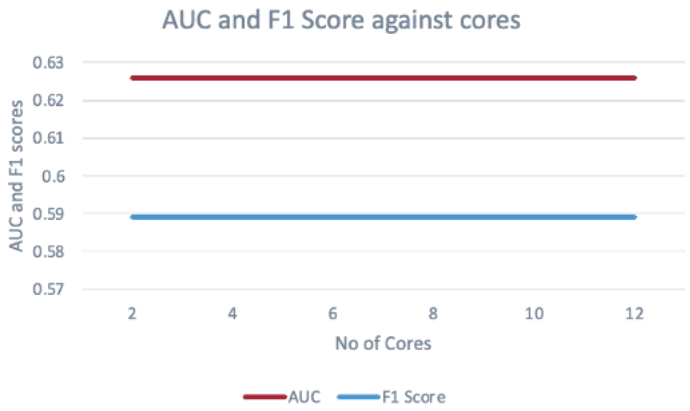


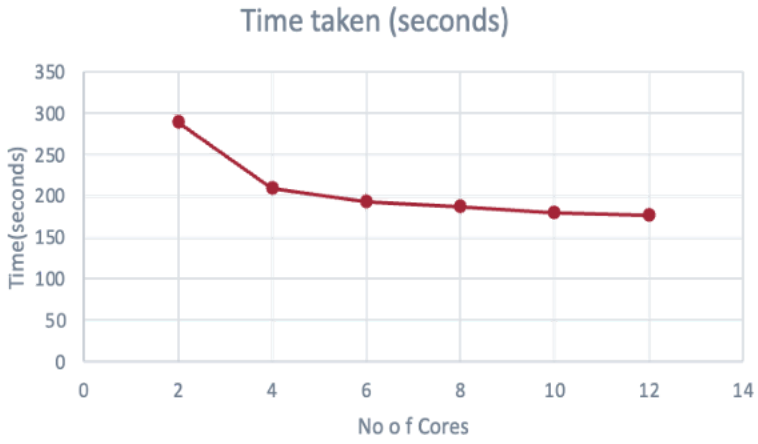


We can observe that the metrics have constantly remained same across all the subsamples and only time has increased with the increase in data.

Performance metrics:

1. **Scale - Subsampling:** We have compared the various metrics against different subsamples. The data was sampled at different percentages - 20%, 40%, 60%, 80% and 100%.
2. **Scale - Cores:** We have compared the various metrics against different cores ,namely, {2, 4, 6, 8, 10, 12} cores. The .config("spark.default.parallelism", str(cores)) configuration setting determines the number of cores allocated for parallel computation, enhancing the performance of the Spark job.





We can observe that the cores only have an effect on time taken and time decreases with increase in number of cores.

2. Gradient-Boosted Tree Classifier

Gradient-Boosted Trees (GBT) is a popular machine learning algorithm used for both classification and regression tasks. In PySpark, GBT is implemented in the GBTClassifier class for classification and the GBTRegressor class for regression.

GBT combines the concept of gradient boosting with decision trees to create an ensemble of trees that iteratively corrects the mistakes made by previous trees. The basic idea behind GBT is to train a sequence of weak learners, typically decision trees, and then combine their predictions to make the final prediction. Each subsequent weak learner is trained to correct the errors made by the previous learners, hence the term "gradient" in GBT.

GBT in PySpark provides various hyperparameters that can be tuned to optimize the model performance, including maxDepth (maximum depth of the decision trees), maxBins (maximum number of bins for continuous features), maxIter (maximum number of boosting iterations), stepSize (learning rate for each iteration), and subsamplingRate (fraction of training data used for each iteration).

Parameters Grid: We're using a grid of parameters to tune our model. This means we're training multiple versions of our GBT model, each with different settings, and then comparing their performance to find the best configuration.

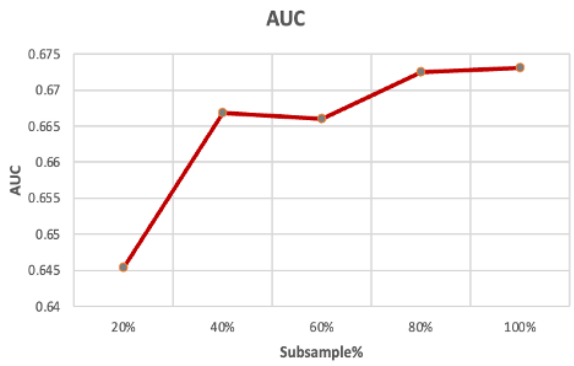
1. Maximum Depth Parameter (maxDepth): 5, 10
2. Maximum number of bins (maxBins): 32, 64

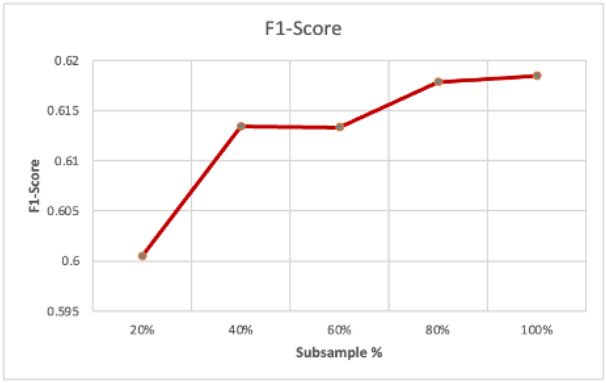
3. Maximum Iterations (maxIter): 20, 30

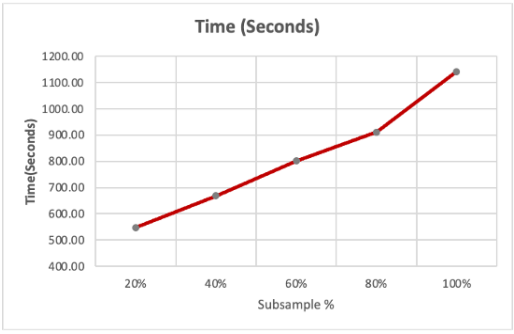
```
# Read the Parquet file into a DataFrame
   reduced_features_with_target_sdf = spark5.read.parquet("gs://dhrithi-yarn-cluster-bucket/EDAed_data.parquet")

   # List of subsample percentages
   subsamples = [1.0, 0.8, 0.6, 0.4, 0.2]
   target_column="HasDetections"

   # Initialize evaluators

   binary_evaluator = BinaryClassificationEvaluator(labelCol=target_column)
   regression_evaluator = RegressionEvaluator(labelCol=target_column)


   # Loop over each subsample
   for subsample in subsamples:
       print(f"Training with {subsample*100}% data...")

       start_time = time.time()  # Start the timer

       # Sample data
       # False indicates that we are not sampling with replacement
       sample_data = reduced_features_with_target_sdf.sample(False, subsample, seed=42)

       # Split the data into training set (80%) and test set (20%)
       train, test = sample_data.randomSplit([0.8, 0.2], seed=42)


       # Train a Gradient-Boosted Tree (GBT) model
       gbt = GBTClassifier(featuresCol="features", labelCol="HasDetections", maxDepth=5, maxIter=20)

        # Define the parameter grid for hyperparameter tuning
       paramGrid = (ParamGridBuilder()
            .addGrid(gbt.maxDepth, [5, 10, 15])
            .addGrid(gbt.maxIter, [20, 30])
            .addGrid(gbt.maxBins, [32, 64, 128])
            .build())


       # Create a CrossValidator
       crossval = CrossValidator(estimator=gbt,
                         estimatorParamMaps=paramGrid,
                         evaluator=BinaryClassificationEvaluator(labelCol=target_column),
                         numFolds=3)  # Number of cross-validation folds

       # Run cross-validation and choose the best model
       cvModel = crossval.fit(train)

       # Make predictions on the testing data using the best model
       predictions = cvModel.transform(test)

       #Evaluate the model's performance

       auc = binary_evaluator.evaluate(predictions)

       # Calculate F1 score
       multiclass_evaluator = MulticlassClassificationEvaluator(labelCol="HasDetections")
       f1_score = multiclass_evaluator.evaluate(predictions, {multiclass_evaluator.metricName: "f1"})


       end_time = time.time()
       # Get the number of cores used
       spark_context = spark5.sparkContext
       num_cores = spark_context.defaultParallelism
       app_id = spark_context.applicationId


       print(f"Subsample: {subsample*100}%")

       print(f"AUC: {auc}")
       print(f"F1 Score:", f1_score)

       print(f"Time taken: {end_time - start_time} seconds\n")
       print(f"Number of cores used: {num_cores}")
       print(f"Spark application ID: {app_id}")
```

Performance metrics:

1. **Scale - Subsampling:** The data was sampled at different percentages - 20%, 40%, 60%, 80% and 100%.

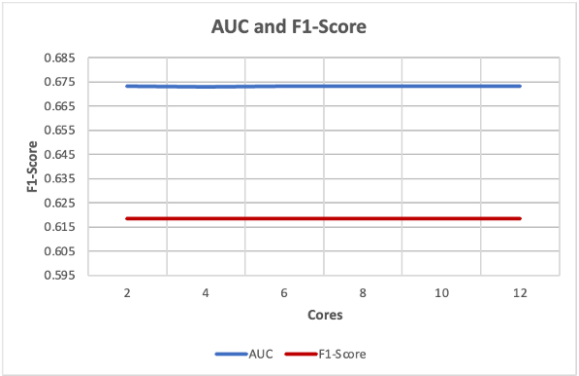


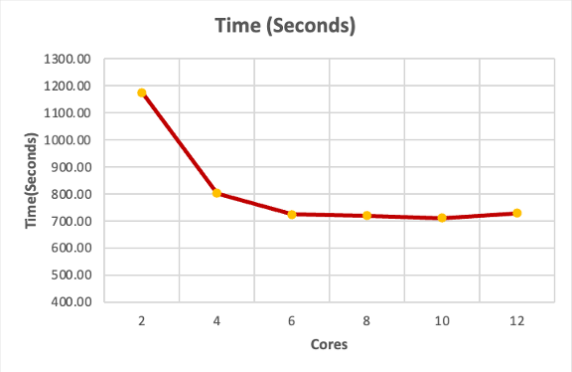




As the above charts illustrate, the AUC values and F1-Scores increase with the sample size except between 40% and 60%. There is also a constant increase in runtime as the sample size increases.

1. **Scale - Cores:** We have compared the various metrics against different cores ,namely, {2, 4, 6, 8, 10, 12} cores.





As the runtime decreases with increase in number of cores until 10 cores after which a slight increase is noted. The AUC and F1-Scores are observed to remain constant.

3. RANDOM FOREST CLASSIFIER

Random Forest is a popular ensemble learning algorithm that combines the predictions of multiple decision trees to make more accurate predictions. In PySpark, Random Forest is implemented in the RandomForestClassifier class for classification tasks and the RandomForestRegressor class for regression tasks.Random Forest is known for its ability to handle high-dimensional data, capture complex interactions, and handle noisy and missing data.

Random Forest in PySpark provides several hyperparameters that can be tuned to optimize model performance. Some of the commonly used hyperparameters include numTrees (number of trees in the forest), maxDepth (maximum depth of each tree), maxBins (maximum number of bins for continuous features), and featureSubsetStrategy (strategy for feature subset selection).

Parameter grid:

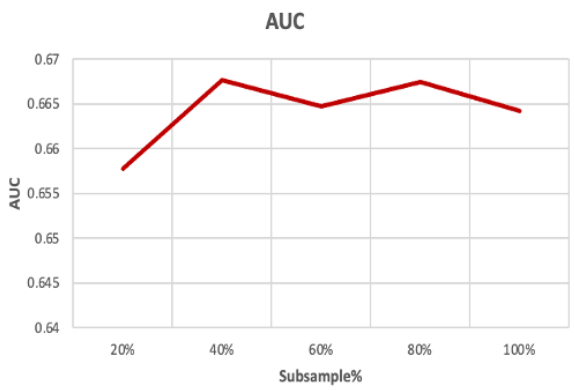
1. Maximum Depth Parameter (maxDepth): 5, 10
2. Number of decision trees(numTrees): 100, 200
3. Strategy for feature subset selection (featureSubsetStrategy): “sqrt”

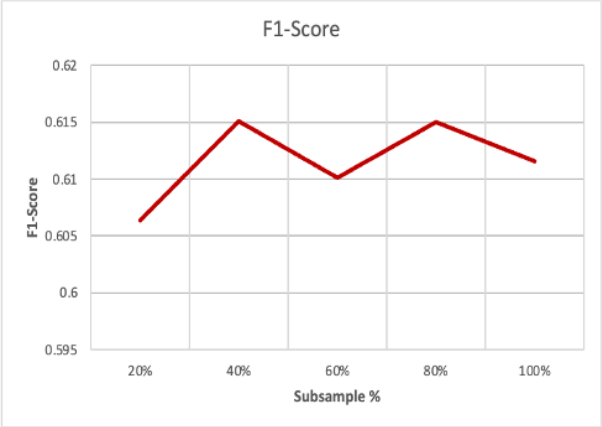
```
if __name__ == "__main__":
    spark5 = SparkSession.builder \
    .appName("Malware Prediction LR") \
    .config("spark.executor.memory", '12g') \
    .config("spark.driver.maxResultSize", '12g') \
    .config("spark.kryoserializer.buffer.max", '512m') \
    .getOrCreate()


    # Read the Parquet file into a DataFrame
    reduced_features_with_target_sdf = spark5.read.parquet("gs://dhrithi-yarn-cluster-bucket/EDAed_data.parquet")

    # List of subsample percentages
    subsamples = [1.0, 0.8, 0.6, 0.4, 0.2]
    target_column="HasDetections"

    # Initialize evaluators

    binary_evaluator = BinaryClassificationEvaluator(labelCol=target_column)
    regression_evaluator = RegressionEvaluator(labelCol=target_column)


    # Loop over each subsample
    for subsample in subsamples:
        print(f"Training with {subsample*100}% data...")

        start_time = time.time()  # Start the timer

        # Sample data
        # False indicates that we are not sampling with replacement
        sample_data = reduced_features_with_target_sdf.sample(False, subsample, seed=42)
```

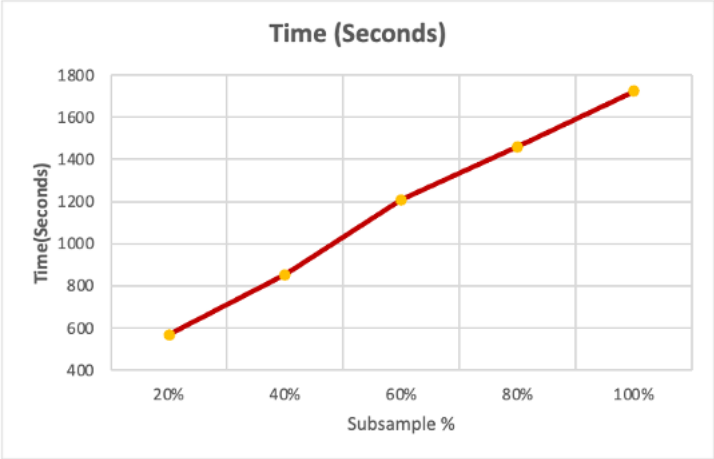
```
# Split the data into training set (80%) and test set (20%)
train, test = sample_data.randomSplit([0.8, 0.2], seed=42)


# Train a Random Forest model
rf = RandomForestClassifier(featuresCol="features", labelCol="HasDetections")

 # Define the parameter grid for hyperparameter tuning
paramGrid = (ParamGridBuilder()
     .addGrid(rf.numTrees, [100, 200])
     .addGrid(rf.maxDepth, [5, 10])
     .addGrid(rf.featureSubsetStrategy, ["sqrt"])
     .build())


# Create a CrossValidator
crossval = CrossValidator(estimator=rf,
                  estimatorParamMaps=paramGrid,
                  evaluator=BinaryClassificationEvaluator(labelCol=target_column),
                  numFolds=3)  # Number of cross-validation folds

# Run cross-validation and choose the best model
cvModel = crossval.fit(train)

# Make predictions on the testing data using the best model
predictions = cvModel.transform(test)

#Evaluate the model's performance
auc = binary_evaluator.evaluate(predictions)

# Calculate F1 score
multiclass_evaluator = MulticlassClassificationEvaluator(labelCol="HasDetections")
f1_score = multiclass_evaluator.evaluate(predictions, {multiclass_evaluator.metricName: "f1"})

# Calculate precision
precision = multiclass_evaluator.evaluate(predictions, {multiclass_evaluator.metricName: "weightedPrecision"})

# Calculate recall
recall = multiclass_evaluator.evaluate(predictions, {multiclass_evaluator.metricName: "weightedRecall"})


end_time = time.time()
# Get the number of cores used
spark_context = spark5.sparkContext
num_cores = spark_context.defaultParallelism
app_id = spark_context.applicationId


print(f"Subsample: {subsample*100}%")

print(f"AUC: {auc}")
print(f"F1 Score:", f1_score)
print("Recall: ", recall)
print("Precision: ", precision)
print(f"Time taken: {end_time - start_time} seconds\n")
print(f"Number of cores used: {num_cores}")
print(f"Spark application ID: {app_id}")
```

Performance metrics:

1. **Scale - Subsampling:** The data was sampled at different percentages - 20%, 40%, 60%, 80% and 100%.

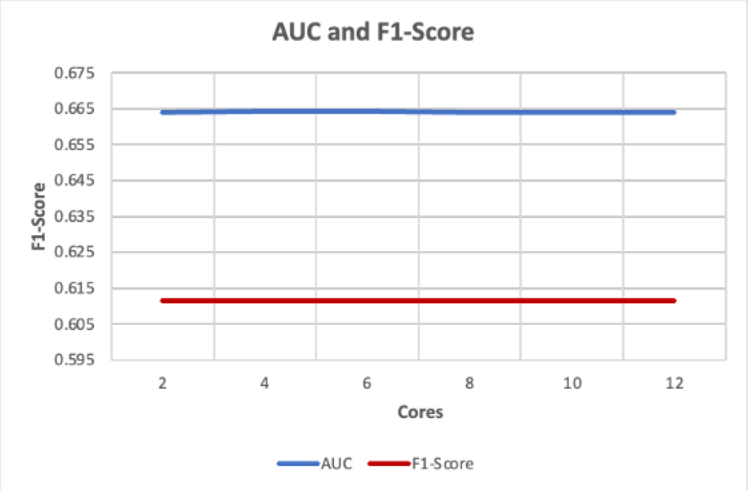


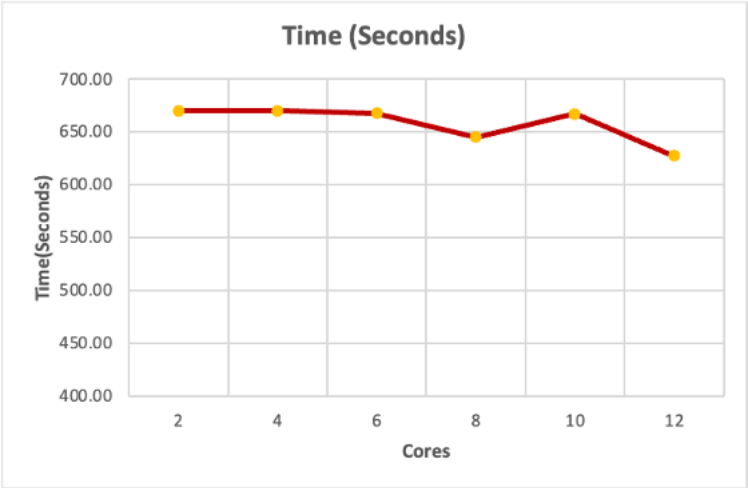




The AUC values and F1-Scores vary slightly between 40% and 100%. There is a uniform increase in time taken as the sample size increases.

1. **Scale - Cores:** We have compared the various metrics against different cores ,namely, {2, 4, 6, 8, 10, 12} cores.





We can observe that the cores only have a slight effect on time taken but not a noticeable difference as we increase the number of cores.

LINEAR SVC

LinearSVC(Linear Support Vector Classification) is a machine learning model that belongs to the support vector machines(SVM) family. It is used for binary classification tasks, where the goal is to predict whether an input belongs to one of two classes. LinearSVC is based on the concept of finding a hyperplane that separates the two classes in the input spaces.

LinearSVC in PySpark provides various hyperparameters that can be tuned to optimize the model performance including regParam(Regularization Parameter), fitIntercept(determines whether to fit an intercept term in the model), tol(sets convergence tolerance for the solver), maxIter (maximum number of boosting iterations), stepSize (learning rate for each iteration), and subsamplingRate (fraction of training data used for each iteration).

Parameters Grid: We're using a grid of parameters to tune our model. This means we're training multiple versions of our GBT model, each with different settings, and then comparing their performance to find the best configuration.

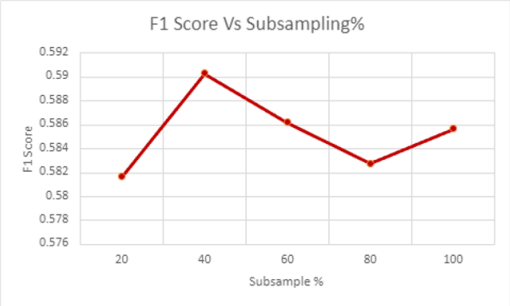
1. Maximum Depth Parameter (maxDepth): 5, 10
2. Maximum number of bins (maxBins): 32, 64
3. Maximum Iterations (maxIter): 20, 30

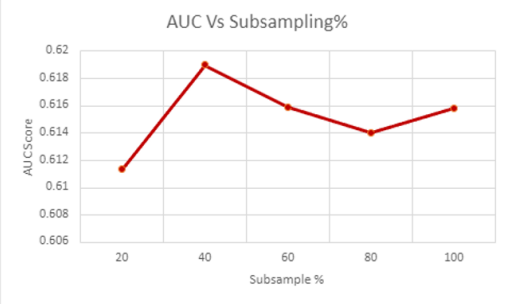
```
if __name__ == "__main__":
    spark = SparkSession.builder \
        .appName("Malware Prediction SVM") \
        .config("spark.executor.memory", '12g') \
        .config("spark.driver.maxResultSize", '12g') \
        .config("spark.kryoserializer.buffer.max", '512m') \
        .getOrCreate()

    # Read the Parquet file into a DataFrame
    reduced_features_with_target_sdf = spark.read.parquet("gs://project-bucketgcp-a/EDAed_data.parquet")


   # List of subsample percentages
subsamples = [1.0, 0.8, 0.6, 0.4, 0.2]
target_column="HasDetections"
# Initialize evaluators
# BinaryClassificationEvaluator for computing Area Under ROC Curve (AUC)
binary_evaluator = BinaryClassificationEvaluator(labelCol=target_column)

# RegressionEvaluator for computing Root Mean Square Error (RMSE)
regression_evaluator = RegressionEvaluator(labelCol=target_column, metricName="rmse")

# MulticlassClassificationEvaluator for computing precision, recall, and F1 score
multi_evaluator = MulticlassClassificationEvaluator(labelCol=target_column)
```

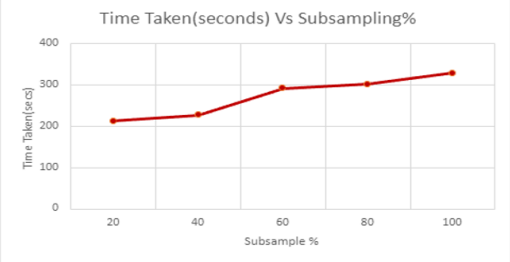
```
timing_values = []  # List to store timing values
subsample_percentages = []  # List to store subsample percentages
rmse_values = []  # List to store RMSE values
# Loop over each subsample
for subsample in subsamples:
    print(f"Training with {subsample*100}% data...")

    start_time = time.time()  # Start the timer

    # Sample data
    # False indicates that we are not sampling with replacement
    sample_data = reduced_features_with_target_sdf.sample(False, float(subsample), seed=42)


    # Split the data into training set (80%) and test set (20%)
    train_data, test_data = sample_data.randomSplit([0.8, 0.2], seed=42)

    # Define the SVM model
    svm = LinearSVC(featuresCol='features', labelCol=target_column)

    # Set up the parameter grid
    # The cross-validator will try all combinations of these parameters
    paramGrid = ParamGridBuilder() \
        .addGrid(svm.regParam, [0.01, 0.1, 1.0]) \
        .addGrid(svm.fitIntercept, [False, True]) \
        .addGrid(svm.maxIter, [10, 100, 1000]) \
        .build()

    # Set up the cross-validator
    # estimator is the machine learning algorithm to be used (SVM in this case)
    # estimatorParamMaps is the grid of parameters to try
    # evaluator is the metric used to measure the performance of the models
    # numFolds is the number of cross-validation folds
    crossval = CrossValidator(estimator=svm,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(labelCol=target_column),
                          numFolds=3)

    # Train the model
    cv_model = crossval.fit(train_data)

    # Make predictions on the test set
    predictions = cv_model.transform(test_data)

    # Evaluate the model using AUC and RMSE
    auc = binary_evaluator.evaluate(predictions)
    rmse = regression_evaluator.evaluate(predictions)
    # Compute accuracy manually
    total_count = predictions.count()
    correct_count = predictions.filter(expr('prediction == HasDetections')).count()
    accuracy = correct_count / total_count
    precision = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: "weightedPrecision"})
    recall = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: "weightedRecall"})
    f1_score = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: "f1"})

    end_time = time.time()  # Stop the timer
    # Get the number of cores used
    spark_context = spark.sparkContext
    num_cores = spark_context.defaultParallelism
    # Get the Spark application ID
    app_id = spark_context.applicationId

    timing_values.append(end_time - start_time)
    subsample_percentages.append(subsample * 100)
    rmse_values.append(rmse)

    print(f"Subsample: {subsample*100}%")
    print(f"Accuracy: {accuracy}")
    print(f"Precision: {precision}")
    print(f"Recall: {recall}")
    print(f"F1 Score: {f1_score}")
    print(f"Time taken: {end_time - start_time} seconds\n")
    print(f"Number of cores used: {num_cores}")
    print(f"Spark application ID: {app_id}")
```

Performance metrics:

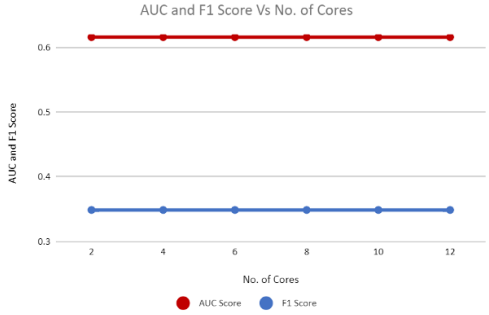
1. Scale - Subsampling : Data was sampled at different percentages (20%, 40%, 60%, 80% and 100%). Metrics have slightly increased across all the samples.
    a. Area Under the ROC Curve (AUC)



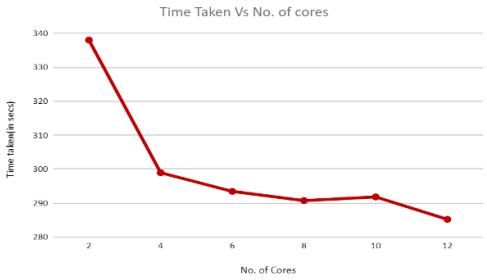




1. Scale - Cores
    a. AUC and F1 Score



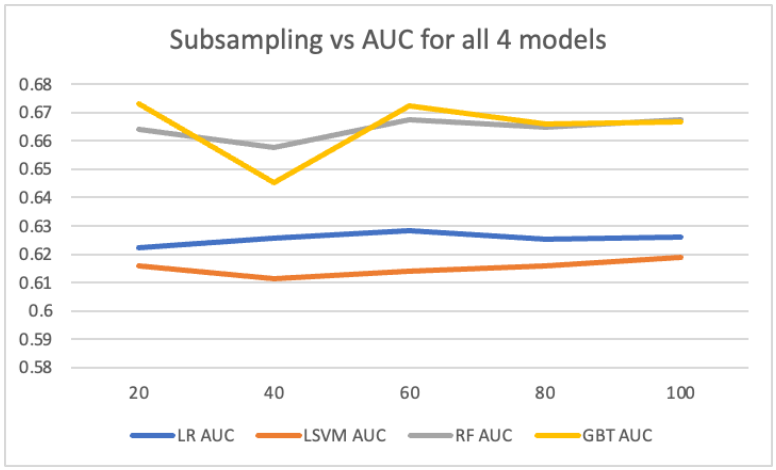
B. Runtime

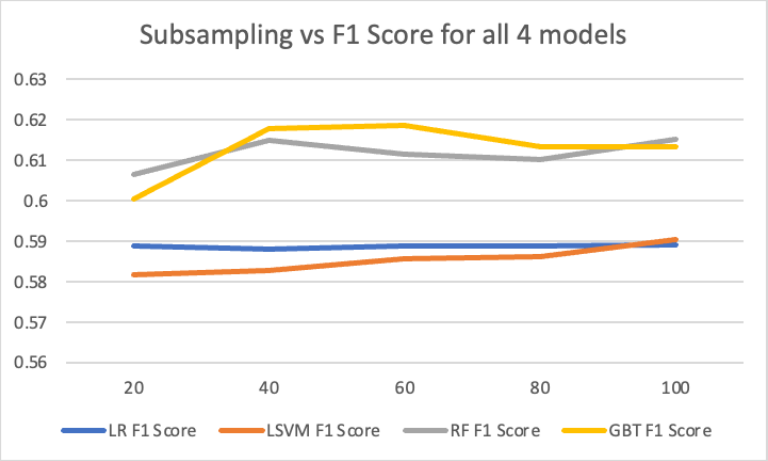


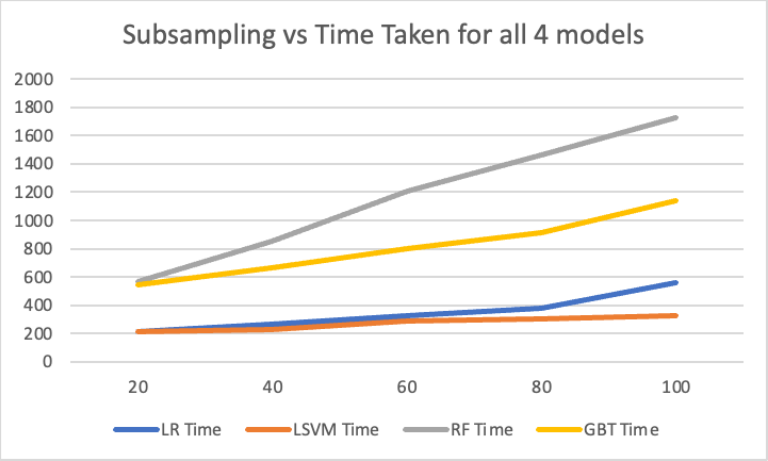
CONCLUSION

- **Subsampling Impact:** Subsampling was implemented to understand its effect on the model performance. However, it was observed that accuracy and ROC-AUC scores remained consistent across all subsample sizes, showcasing the robustness of our model
- **Scalability with Nodes:** Increasing the number of nodes resulted in a decrease in execution time, demonstrating the scalability and parallel processing capabilities of the model. This emphasizes the efficiency gains from leveraging distributed computing resources
- **Data Size and Time:** As expected, the time taken for computation increased with the size of the data. This validates the trade-off between data volume and computation time, highlighting the need for appropriate resource allocation strategies
- **Model Performance:** Among all the evaluated models, GBT exhibited the best performance for binary classification in this particular context. It showed consistent performance across different subsamples and node counts, reinforcing its suitability for the given task
- **Future Considerations:** While the current model performs well, there is always scope for further tuning and optimization, such as adjusting hyperparameters or exploring other modeling techniques. Continual evaluation against new data will also be crucial to maintain the model's performance over time
- GBT has shown us to give the maximum F1-Score. The data being fairly complicated would give better predictions on using advanced algorithms like Neural Networks etc.

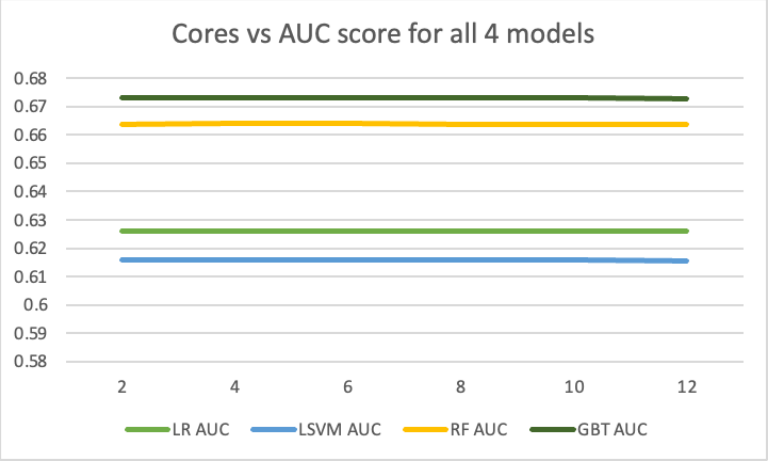
Graphs comparing the Subsampling vs different metrics for all 4 models.

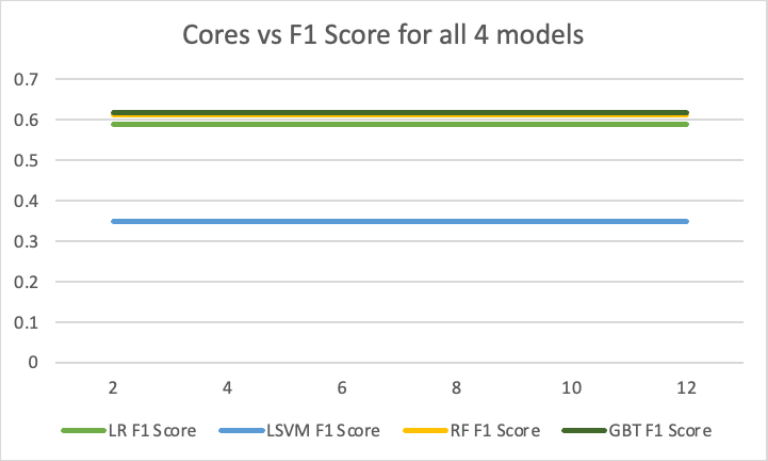


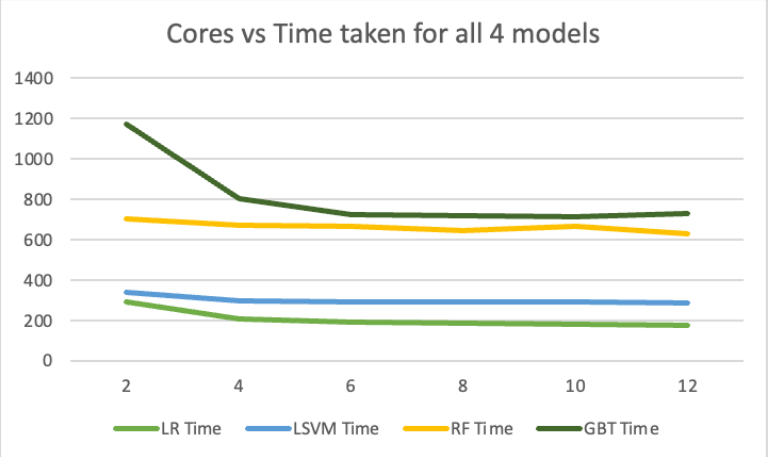




Graphs comparing the effect of different cores on all 4 models.







CHALLENGES FACED

- **Handling Large Data:** The initial dataset was extremely large, which often caused our PySpark environment to run into 'Out of Memory' errors. This challenged our ability to effectively manipulate and process the data. To overcome this, we reduced the dataset to a more manageable size of 2.4 million records
- **High Cardinality:** Many of the categorical columns in the dataset had high cardinality, meaning they contained a large number of unique values. This significantly slowed down the One-Hot Encoding and dimensionality reduction steps, increasing the complexity of our data processing tasks
- **GCP Free Tier Limitations:** The free tier of Google Cloud Platform (GCP) provides a limited number of addresses, restricting us to use no more than three worker nodes. This imposed a constraint on the scale at which we could parallelize our computations and thus impacted the speed of data processing and model training
- **Optimization Needs:** Due to the aforementioned issues, there was a continuous need for optimization and efficient resource usage. We had to strike a balance between computational resources and processing times, which required careful planning and execution of data processing and modeling tasks