Drop Prevention in UR3Cobots

Mohan Teja Guddimettla
MS ESDS,50560465
SUNY Buffalo,
Buffalo, USA,
Email: mohantej@buffalo.edu

Mounika Reddy Katikam Venkata
MS ESDS,50559720
SUNY Buffalo,
Buffalo, USA,
Email: mkatikam@buffalo.edu

Sudheer Kumar Reddy Batthina
MS ESDS,50559684
SUNY Buffalo,
Buffalo, USA,
Email: sbatthin@buffalo.edu

***Abstract*—The UR3e is a very small and light robotic arm used in tight spaces, like desks or inside machines. Sometimes, the UR3e loses grip, and if it holds something fragile, loss of grip could be very destructive. This paper aims to analyze and predict grip loss to prevent accidents. By examining the programming of the robot and understanding its limitations, adjustments can be made to ensure a safe and efficient operation.**

I. Problem Statement

The objective of the proposed project is to bring fault detection and predictive maintenance in robotics, particularly in the UR3 cobot through multi-dimensional time series operational data analysis. This involves understanding electrical currents, temperatures, and joint speeds to develop models that predict equipment failure and optimize operational parameters for performance and longevity.

II. Background

As robotics in industry grows, ensuring maximum efficiency with minimum downtimes is crucial. Current scheduled maintenance is overly conservative, with unexpected failures leading to costly disruptions. The UR3 CobotOps Dataset provides operational data that can enhance maintenance using data-driven methods.

III. Significance

This project aims to develop predictive models for intelligent forecasting of failures, thereby reducing operational downtime and prolonging cobot life through timely maintenance. The results will contribute to Industry 4.0 objectives such as resiliency, flexibility, and efficiency in manufacturing.

IV. Data Sources

The UR3 CobotOps Dataset from the UCI Machine Learning Repository consists of 7,409 instances with 20 features, providing multivariate, time-series data on the UR3 cobot's operations, useful for fault detection, predictive maintenance, and optimization studies.

V. Loading Dataset

We initialized PySpark and loaded the dataset by specifying the file location and using an appropriate method to read the data into a DataFrame. Enabling the header option ensures the first row is recognized as column names. This

```
# Load the dataset
file_path = "/Users/sudheer/Downloads/dataset.csv"
data_df = spark.read.csv(file_path, header=True, inferSchema=True)
```

Fig. 1. Load dataset

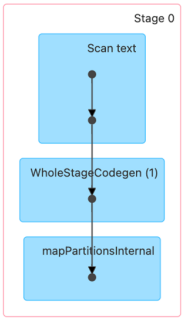


Fig. 2. DAG1

DAG in Stage 0 begins by scanning text data and applying WholeStageCodegen for optimized execution. The process concludes with mapPartitionsInternal, which handles partition-level computations efficiently.

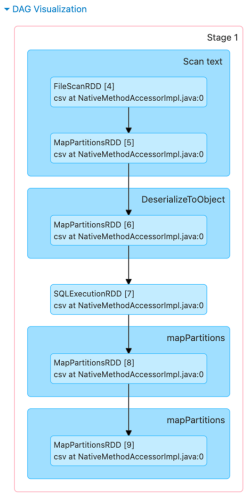


Fig. 3. DAG2

This DAG for Stage 1 starts with scanning text data from a CSV file using FileScanRDD, followed by partition-based

transformations through MapPartitionsRDD. Data is deserialized into objects and processed using SQLExecutionRDD for structured operations. The stage concludes with further partition-based processing for efficient data handling.

VI. Data Cleaning and Preprocessing

1) **a Renaming Columns**: It's vital to rename columns to understand its original units and have a consistent column naming strategy to use in later parts of the data process and this is also required for clarity and consistency in data analysis and visualization. Below we are Expanding on all the column names for better understanding. Renamed columns with their respective units for better understanding of their comparisons while EDA.
In this code snippet, we are renaming the columns of a PySpark DataFrame to improve clarity and readability. A dictionary renamed_columns defines the mapping of old column names to new, more descriptive names. The select method along with alias is used to apply these new names to the DataFrame, and the updated DataFrame is displayed using data_df.show()

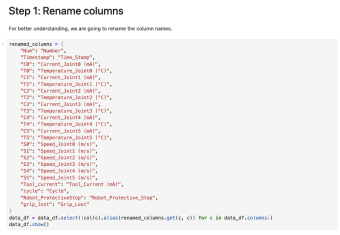


Fig. 4. renaming coulmns

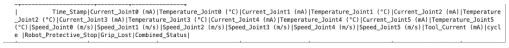
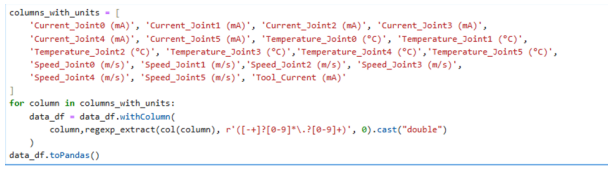


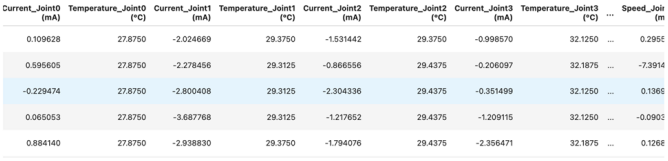
Fig. 5. Renaming output

2) **Data Normalization**: We have already included the units in the column names, so it's no longer necessary to have the units in the data itself. Therefore, we have removed the units from the data to avoid redundancy.
This code extracts numeric values from columns containing units (e.g., "mA", "°C", "m/s") using regular expressions, effectively removing the units. It then converts the extracted values into a numerical format to standardize the data for further analysis.



```
columns_with_units = [
    'Current_Joint0 (mA)', 'Current_Joint1 (mA)', 'Current_Joint2 (mA)', 'Current_Joint3 (mA)',
    'Current_Joint4 (mA)', 'Current_Joint5 (mA)', 'Temperature_Joint0 (°C)', 'Temperature_Joint1 (°C)',
    'Temperature_Joint2 (°C)', 'Temperature_Joint3 (°C)','Temperature_Joint4 (°C)','Temperature_Joint5 (°C)',
    'Speed_Joint0 (m/s)', 'Speed_Joint1 (m/s)','Speed_Joint2 (m/s)', 'Speed_Joint3 (m/s)',
    'Speed_Joint4 (m/s)', 'Speed_Joint5 (m/s)', 'Tool_Current (mA)'
]
for column in columns_with_units:
    data_df = data_df.withColumn(
        column,regexp_extract(col(column), r'([-+]?[0-9]*\.?[0-9]+)', 0).cast("double")
    )
data_df.toPandas()
```

Fig. 6. data normalization



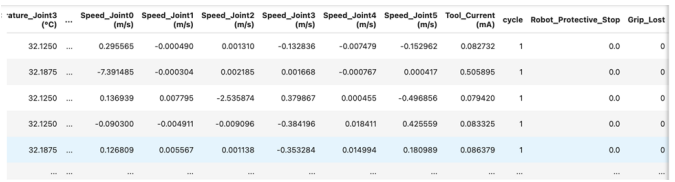
Current_Joint0 (mA)	Temperature_Joint0 (°C)	Current_Joint1 (mA)	Temperature_Joint1 (°C)	Current_Joint2 (mA)	Temperature_Joint2 (°C)	Current_Joint3 (mA)	Temperature_Joint3 (°C)	...	Speed_Joi (m
0.109628	27.8750	-2.024669	29.3750	-1.531442	29.3750	-0.998570	32.1250	...	0.295
0.595605	27.8750	-2.278456	29.3125	-0.866556	29.4375	-0.206097	32.1875	...	-7.391
-0.229474	27.8750	-2.800408	29.3125	-2.304336	29.4375	-0.351499	32.1250	...	0.136
0.065053	27.8750	-3.687768	29.3125	-1.217652	29.4375	-1.209115	32.1250	...	-0.090
0.884140	27.8750	-2.938830	29.3750	-1.794076	29.4375	-2.356471	32.1875	...	0.126

Fig. 7. normalization output

3) **Data Standardization**: Standardise the output columns into one single format because we found out all the different variations of True or False in the data and mapped them to a single format.
This code updates the Grip_Lost column by converting textual values like "TRUE", "T", and their lowercase equivalents to 1, and all other values to 0. This transformation standardizes the column into a binary format, making it suitable for numerical analysis.

```
data_df = data_df.withColumn(
    "Grip_Lost",
    when(col("Grip_Lost").isin("TRUE", "T", "t", "true"), 1).otherwise(0)
)
data_df.toPandas()
```

Fig. 8. standardize data



ature_Joint3 (°C)	...	Speed_Joint0 (m/s)	Speed_Joint1 (m/s)	Speed_Joint2 (m/s)	Speed_Joint3 (m/s)	Speed_Joint4 (m/s)	Speed_Joint5 (m/s)	Tool_Current (mA)	cycle	Robot_Protective_Stop	Grip_Lost
32.1250	...	0.295565	-0.000490	0.001310	-0.132836	-0.007479	-0.152962	0.082732	1	0.0	0
32.1875	...	-7.391485	-0.000304	0.002185	0.001668	-0.000767	0.000417	0.505895	1	0.0	0
32.1250	...	0.136939	0.007795	-2.535874	0.379867	0.000455	-0.496856	0.079420	1	0.0	0
32.1250	...	-0.090300	-0.004911	-0.009096	-0.384196	0.018411	0.425559	0.083325	1	0.0	0
32.1875	...	0.126809	0.005567	0.001138	-0.353284	0.014994	0.180989	0.086379	1	0.0	0
...	...	...	...	...	...	...	...	...	...	...	...

Fig. 9. standardize data output

4) **Removing Unwanted Columns**: A few columns exist in a dataset which are made to help the creators for indexing but are not really useful for analysis, removing such columns will help create clearer and focused dataset.
This code removes the column named "Number" from the DataFrame by creating a list of columns to keep. It then maps the remaining column values into a new RDD (cleaned_rdd) and converts it back to a DataFrame using toDF. The updated DataFrame excludes the unwanted column and retains the specified structure.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import avg, col
from pyspark.sql.types import TimestampType, DoubleType

# Remove Unwanted Columns
columns_to_keep = [col_name for col_name in data_df.columns if col_name != "Number"]
cleaned_rdd = data_df.rdd.map(lambda row: [row[c] for c in columns_to_keep])
data_df = cleaned_rdd.toDF(columns_to_keep)
```

Fig. 10. removing unwanted columns

	Time_Stamp	Temperature_Joint0 (°C)	Temperature_Joint1 (°C)	Temperature_Joint2 (°C)	Temperature_Joint3 (°C)	Temperature_Joint4 (°C)	Temperature_Joint5 (°C)	cycle	Robot_P
0	2022-10-26 08:21:37.159	28.688	30.375	30.563	33.438	33.8750	33.5000	9	
1	2022-10-26 08:21:41.185	28.688	30.375	30.563	33.500	33.8750	33.5000	9	
2	2022-10-26 08:21:45.197	28.688	30.375	30.563	33.500	33.9375	33.5000	9	
3	2022-10-26 08:21:47.205	28.750	30.375	30.563	33.500	33.9375	33.5000	9	
4	2022-10-26 08:21:49.210	28.688	30.438	30.563	33.563	33.9375	33.5625	9	
...	...	...	...	...	...	...	...	...	...
3784	2022-10-26 15:36:02.555	37.188	40.313	40.688	43.375	45.2500	44.5625	264	
3785	2022-10-26 15:36:03.562	37.188	40.313	40.688	43.375	45.2500	44.5000	264	
3786	2022-10-26 15:36:04.571	37.188	40.250	40.688	43.375	45.2500	44.5000	264	
3787	2022-10-26 15:36:05.572	37.188	40.313	40.625	43.375	45.2500	44.5625	264	
3788	2022-10-26 15:36:06.580	37.125	40.313	40.688	43.375	45.2500	44.5625	264	

3789 rows × 37 columns

Fig. 11. removing unwanted columns output

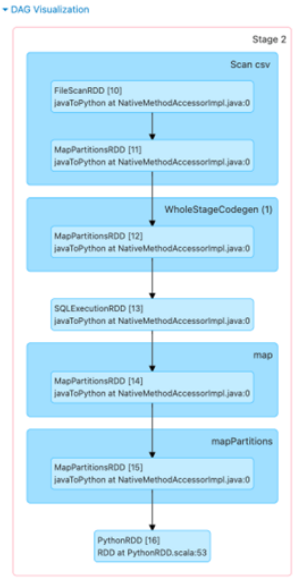


Fig. 12. Removing unwanted columns DAG

This DAG visualization represents the execution flow of a Spark job in Stage 2. It starts by scanning a CSV file using FileScanRDD, followed by optimizations through WholeStageCodegen for efficient processing. The execution involves multiple transformations, such as mapping and partitioning, as shown by MapPartitionsRDD and SQLExecutionRDD, eventually producing the final PythonRDD for the Python-based processing. Each step represents a transformation or action in the Spark workflow.

5) **Handling Missing and Duplicate Values**: Duplicate or very similar rows will skew our data into preference for a single type so we must consider that and limit it.
We checked all the rows, removing the ones with missing values, as their number is negligible compare with total number of rows in dataset. In this way, data completeness will be granted without influencing too much the overall dimension of the dataset.
The code removes rows with missing values using na.drop() and eliminates duplicate rows by converting the DataFrame to an RDD, applying distinct(), and converting it back to a DataFrame. This ensures the dataset is clean and free from redundant or incomplete data for analysis.

```
data_df = data_df.na.drop()
distinct_rdd = data_df.rdd.distinct()
data_df = distinct_rdd.toDF(data_df.columns)
```

Fig. 13. remove missing and duplicate values

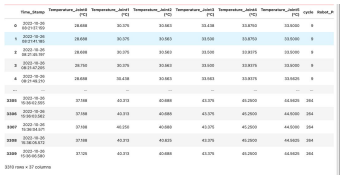


Fig. 14. remove missing and duplicate values output

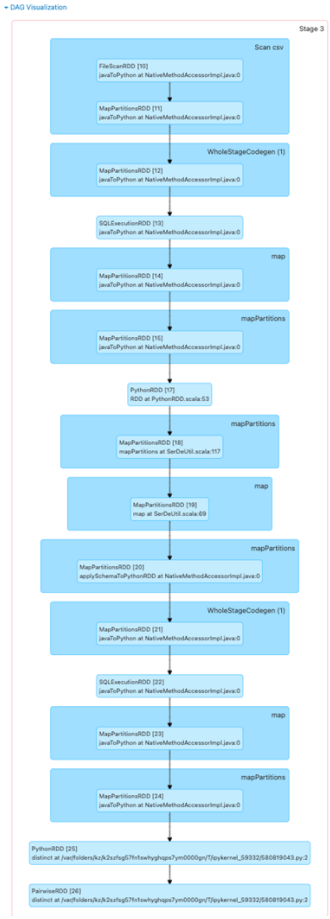


Fig. 15. Missing and duplicate value DAG

This DAG visualization outlines the execution flow for Stage 3 of a Spark job. It starts by reading a CSV file (FileScanRDD) and applies optimizations using WholeStageCodegen to improve processing efficiency.

The data undergoes transformations such as mapping (MapPartitionsRDD) and SQL execution (SQLExecutionRDD) to process and partition the data at various stages. Further, additional operations like schema application, distinct filtering, and pairwise transformations are performed. Finally, the stage completes with Python-based processing (PythonRDD) for customized operations.

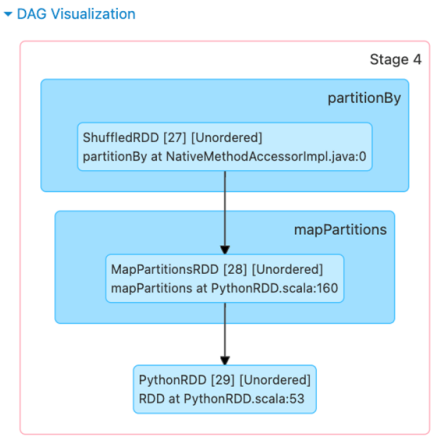


Fig. 16. missing and duplicate value DAG2

This DAG represents Stage 4, where data is repartitioned using partitionBy (ShuffledRDD) to reorganize it across nodes. The partitioned data is then processed using mapPartitions and passed to a Python-based operation (PythonRDD) for further customized processing.

6) **Format Time Stamp and Rounding up the decimal values**: We changed the date column format from object type to datetime format for performing the correct time-based analysis. That way, it will handle all operations regarding dates much better, including filtering and sorting.
We rounded the decimal values to 3 places for all current, speed, and tool current columns to ensure consistency and precision. This adjustment improves the readability and uniformity of the data.
This code formats the Time_Stamp column into a consistent timestamp format and converts specified columns to Double Type for numerical processing. It also rounds the decimal values in these columns to three decimal places for improved precision and uniformity.

```
data_df = data_df.withColumn("Time_Stamp", to_timestamp(col("Time_Stamp"), "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"))
columns = [
    'Current_Joint0 (mA)', 'Current_Joint1 (mA)', 'Current_Joint2 (mA)', 'Current_Joint3 (mA)',
    'Current_Joint4 (mA)', 'Current_Joint5 (mA)', 'Temperature_Joint0 (°C)', 'Temperature_Joint1 (°C)',
    'Temperature_Joint2 (°C)', 'Temperature_Joint3 (°C)', 'Speed_Joint0 (m/s)', 'Speed_Joint1 (m/s)',
    'Speed_Joint2 (m/s)', 'Speed_Joint3 (m/s)', 'Speed_Joint4 (m/s)', 'Speed_Joint5 (m/s)', 'Tool_Current (mA)'
]
# Convert other columns to double
for col_name in columns:
    data_df = data_df.withColumn(col_name, col(col_name).cast(DoubleType()))
    data_df = data_df.withColumn(col_name, round(col(col_name), 3))

data_df.toPandas()
```

Fig. 17. format time stamp and rounding up decimals

	Time_Stamp	Current_Joint0 (mA)	Temperature_Joint0 (°C)	Current_Joint1 (mA)	Temperature_Joint1 (°C)	Current_Joint2 (mA)	Temperature_Joint2 (°C)	Current_Joint3 (mA)	Temperature_
0	2022-10-26 08:17:21.847	0.110	27.875	-2.025	29.375	-1.531	29.375	-0.999	
1	2022-10-26 08:17:22.852	0.596	27.875	-2.278	29.313	-0.867	29.438	-0.206	
2	2022-10-26 08:17:23.857	-0.229	27.875	-2.800	29.313	-2.304	29.438	-0.351	
3	2022-10-26 08:17:24.863	0.065	27.875	-3.688	29.313	-1.218	29.438	-1.209	
4	2022-10-26 08:17:25.877	0.884	27.875	-2.939	29.375	-1.794	29.438	-2.356	

Fig. 18. format time stamp and rounding up decimals output

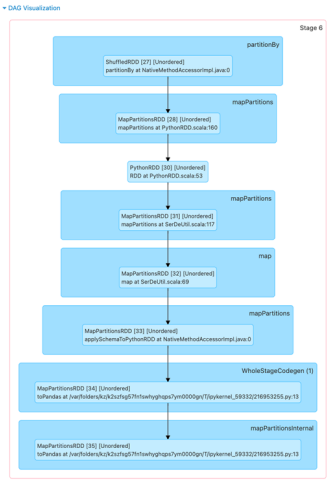


Fig. 19. format time stamp and rounding up decimals DAG

This DAG for Stage 6 begins with repartitioning the data using partitionBy (ShuffledRDD), followed by multiple transformations such as mapping and partition-based processing (MapPartitionsRDD). It applies schema transformations and optimizations with WholeStageCodegen, ultimately converting the processed data into a Pandas DataFrame (toPandas)

7) **Combine columns**: We combined Robot Protective Stop and Grip Lost into a new column called Combined Status and then encoded it into a numerical format for easier processing. It enhances the dataset for better analysis and modeling.
The code concatenates the Robot_Protective_Stop and Grip_Lost columns into a single string column, Combined_Status, using an underscore as a separator. It then assigns numerical codes to specific combinations of Combined_Status.

```
# Combine Robot_Protective_Stop and Grip_Lost columns into a single string column
data_df = data_df.withColumn(
    "Combined_Status",
    concat_ws("_", col("Robot_Protective_Stop").cast(StringType()), col("Grip_Lost").cast(StringType()))
)
data_df = data_df.withColumn(
    "Combined_Status",
    when(col("Combined_Status") == "0_0", 0)
    .when(col("Combined_Status") == "1_0", 1)
    .when(col("Combined_Status") == "0_1", 2)
    .when(col("Combined_Status") == "1_1", 3)
    .otherwise(4)  # For unexpected values
)
# Print unique values in Combined_Status
data_df.select("Combined_Status").distinct().show()
```

Fig. 20. Combine Columns

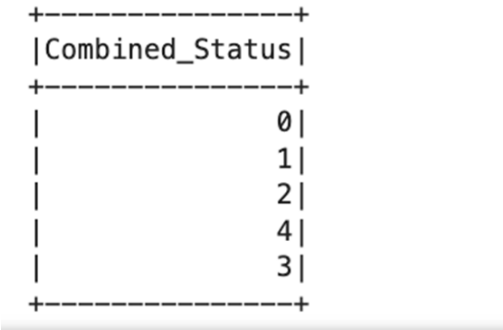


Fig. 21. Combine columns output

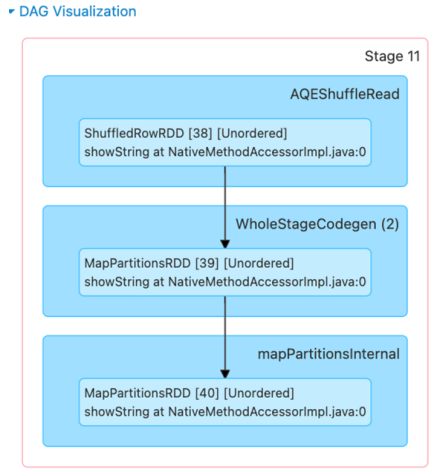


Fig. 22. Combine Columns DAG

This DAG for Stage 11 starts with AQEShuffleRead, which reads shuffled data from previous stages for processing. The WholeStageCodegen step optimizes the execution of transformations for efficiency. Finally, the data is processed in partitions through mapPartitionsInternal to complete the computation. This DAG for

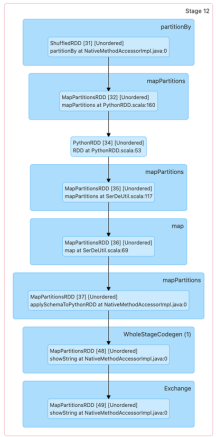


Fig. 23. Combine Columns DAG2

Stage 12 begins with data repartitioning using partitionBy (ShuffledRDD), followed by partition-specific processing through multiple mapPartitions operations. Schema is applied to the data, and WholeStageCodegen optimizes the workflow for efficient execution. The processed data is exchanged across partitions in the final step to complete the computation.

8) **Decompose columns**: We decomposed current and speed into separate positive and negative components to enable a more detailed analysis of directional data.
This code separates each column of current and speed data into two new columns: Positive for positive values and Negative for negative values (as positive magnitudes). The original columns are then dropped, leaving a cleaner dataset for analysis.

```
for col_name in [
    "Current_Joint0 (mA)", "Current_Joint1 (mA)", "Current_Joint2 (mA)", "Current_Joint3 (mA)",
    "Current_Joint4 (mA)", "Current_Joint5 (mA)", "Tool_Current (mA)" ,"Speed_Joint0 (m/s)", "Speed_Joint1 (m/s)",
    "Speed_Joint2 (m/s)", "Speed_Joint3 (m/s)", "Speed_Joint4 (m/s)", "Speed_Joint5 (m/s)"
]:
    data_df = data_df.withColumn(f"{col_name}_Positive", when(col(col_name) > 0, col(col_name)).otherwise(0))
    data_df = data_df.withColumn(f"{col_name}_Negative", when(col(col_name) < 0, -col(col_name)).otherwise(0))
    data_df = data_df.drop(col_name)
```

Fig. 24. Decompose Columns

ure_Joint1 (°C)	Current_Joint2 (mA)	Temperature_Joint2 (°C)	Current_Joint3 (mA)	Temperature_Joint3 (°C)	Current_Joint4 (mA)	...	Speed_Joint1 (m/s)_Negative	Speed_Joint2 (m/s)_Positive	Speed_Joint2 (m/s)_Negative	Speed_ (m/s)_P
32.3125	-0.732	32.6875	-0.878	35.7500	-0.146	...	0.037	0.031	0.000	
32.3125	-1.441	32.7500	-1.442	35.8125	0.336	...	0.060	0.022	0.000	
32.3750	-0.885	32.6875	-0.343	35.8125	0.107	...	0.000	0.000	0.007	
32.3750	-0.955	32.7500	-0.426	35.8125	-0.011	...	0.000	0.000	0.000	
32.3750	-1.040	32.6875	-0.493	35.8125	-0.014	...	0.000	0.000	0.000	
...	...	...	...	...	...	...	...	...	...	
40.3125	-1.083	40.6875	-0.495	43.3750	-0.019	...	0.000	0.000	0.000	
40.3125	-1.094	40.6875	-0.516	43.3750	-0.009	...	0.000	0.000	0.000	
40.2500	-1.121	40.6875	-0.502	43.3750	-0.003	...	0.000	0.000	0.000	
40.3125	-1.092	40.6250	-0.491	43.3750	-0.001	...	0.000	0.000	0.000	
40.3125	-1.099	40.6875	-0.502	43.3750	-0.026	...	0.000	0.000	0.000	

Fig. 25. Decompose Columns Output

9) **Removing Outliers**: This involves the removal of outliers in the current, speed, and temperature columns using a combined quantile-based approach, keeping 95 percent of the data. This ensures that extreme values are filtered out, which will allow for more robust and accurate analysis.
This code identifies and removes outliers from numerical columns by calculating the 2.5th and 97.5th percentiles (lower_percentile and upper_percentile) using approximate quantiles. It defines a filtering function, is_within_bounds, to retain rows where all numeric column values fall within the calculated bounds. The filtered RDD is converted back to a DataFrame with the same schema for further processing.

```
from pyspark.sql.functions import col, expr
lower_percentile = 0.025
upper_percentile = 0.975
numeric_columns = [field.name for field in data_df.schema.fields if field.dataType.typeName() in ("int", "double")]
quantile_bounds = {}
for column in numeric_columns:
    bounds = data_df.approxQuantile(column, [lower_percentile, upper_percentile], 0.001)
    quantile_bounds[column] = bounds

def is_within_bounds(row):
    for col in numeric_columns:
        lower, upper = quantile_bounds[col]
        if not (lower <= row[col] <= upper):
            return False
    return True

filtered_rdd = data_df.rdd.filter(is_within_bounds)
data_df = spark.createDataFrame(filtered_rdd, data_df.schema)
```

Fig. 26. Remove Outliers

Fig. 27. Remove Outliers DAG

This DAG shows the transition from Stage 74, which was skipped (likely due to caching or optimization), to Stage 75. In Stage 75, data is repartitioned using partitionBy, followed by a series of transformations such as mapPartitions, mapping, and optimized execution via WholeStageCodegen.

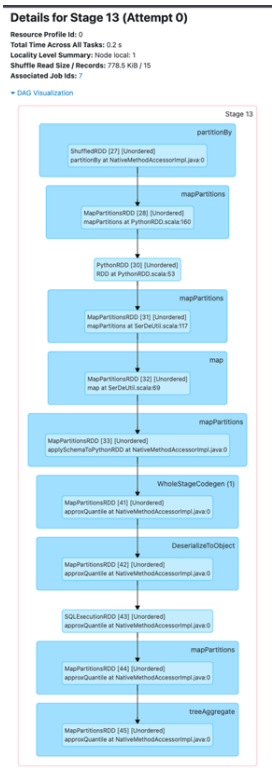


Fig. 28. Remove Outliers DAG2

This DAG for Stage 13 begins with data repartitioning (ShuffleRDD) and processes it through multiple transformations, including mapPartitions and schema application. The workflow is optimized with WholeStageCodegen and involves deserialization and SQL execution for calculating approximate quantiles. The stage concludes with treeAggregate to finalize aggregated computations across partitions.

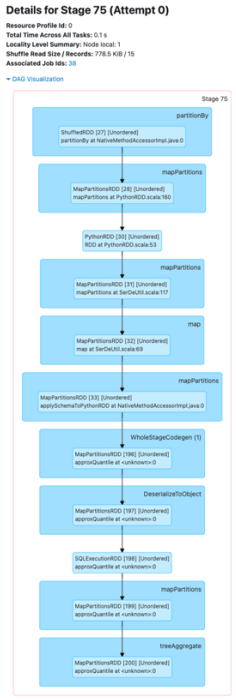


Fig. 29. Remove Outliers DAG3

This DAG for Stage 75 begins with data repartitioning using partitionBy and processes the data through several transformations like mapPartitions and schema application. It leverages WholeStageCodegen for optimization and performs deserialization, followed by SQL execution to compute approximate quantiles. The stage concludes with treeAggregate to combine results across partitions efficiently.

VII. Target Variable

The target variable was created by combining Robot_Protective_Stop and Grip_Lost into a single column, Combined_Status, which became the target variable for our analysis.

VIII. Algorithms Used

This project focuses on classification using the following algorithms:

- Logistic Regression
- Decision Tree
- Random Forest
- Support Vector Machine (SVM)

- Naive Bayes
- Gradient Boosting

A. *Logistic Regression:*

Justification for Choosing Logistic Regression: Logistic Regression was chosen for its simplicity and interpretability, making it a reliable choice for predicting UR3 cobot states such as grip loss and protective stops. Its ability to provide clear insights into feature importance makes it suitable for understanding key factors affecting cobot operations.

Model Training: The Logistic Regression model was trained using PySpark's MLlib framework. The training process used an 80-20 split, with 80% of the data allocated for training and 20% for testing. The model was trained with default convergence parameters, ensuring efficient optimization.

```
[5]: # Logistic Regression Model
lr = LogisticRegression(labelCol='Combined_Status', featuresCol='features')
start_time = time.time()
lr_model = lr.fit(train_data)
end_time = time.time()
lr_predictions = lr_model.transform(test_data)
lr_accuracy, lr_precision, lr_recall, lr_f1 = evaluate_model(lr_predictions)
results.append(['Logistic Regression', lr_accuracy, lr_precision, lr_recall, lr_f1])
print(f"Logistic Regression - Accuracy: {lr_accuracy}, Precision: {lr_precision}, Recall: {lr_recall}, F1 Score: {lr_f1}")
print(f"Time Taken : {end_time-start_time}")
Logistic Regression - Accuracy: 0.9494799405646359, Precision: 0.9207082268003517, Recall: 0.9494799405646359, F1 Score: 0.9320625173151672
Time Taken : 5.30454158782959
```

Fig. 30. Logistic Regression

Effectiveness of the Algorithm: Logistic Regression achieved an accuracy of 94.95%, performing well for frequent operational states but potentially facing challenges with rare events like grip failures due to class imbalance.

Performance Metrics:

- **Overall Accuracy:** 94.95%, reflecting reliable classification performance
- **Precision:** 0.9207, showing the model's ability to minimize false positives.
- **Recall:** 0.9495, reflecting the model's capability to capture most true positives.
- **F1-Score:** 0.93.21, representing a balanced trade-off between precision and recall for frequent events.

Gained Intelligence/Insights: Logistic Regression provided interpretable results, emphasizing the relationship between features and the cobot's operational status. However, its performance on rare failure cases suggests the need for strategies to address class imbalance, such as resampling techniques or advanced models.

Time Taken: 5.30 seconds.

Comparison with Previous Model in Phase2:

The PySpark Logistic Regression model demonstrated comparable accuracy (94.95% vs. 95% in Phase 2) while achieving higher precision (0.9207 vs. 0.91) and F1-Score (0.9321 vs. 0.93), though the Phase 2 model had slightly better recall (0.95 vs. 0.9495). PySpark completed training in 5.30 seconds, while the Phase 2 model was faster for small data, finishing in 0.13 seconds. However, PySpark handled all classes consistently, while the Phase 2 model struggled with minority classes, showing 0.00 precision and recall for some. Overall, PySpark's distributed processing made it better suited for large datasets, offering balanced and scalable performance.

DAG Visualization for Logistic Regression Model:

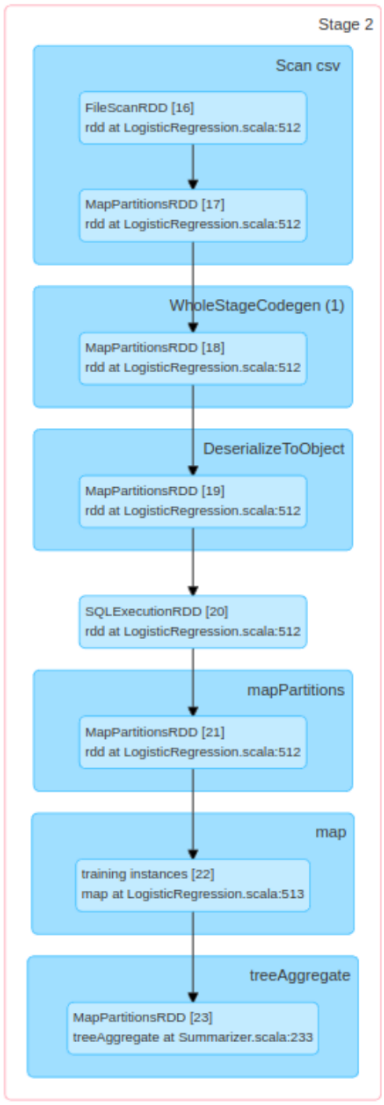


Fig. 31. DAG LR

This DAG represents the execution flow for the PySpark Logistic Regression model during Stage 2. It starts by scanning the CSV file using the FileScanRDD, followed by transformations such as mapPartitions and WholeStageCodegen for efficient data processing. The DeserializeToObject step converts raw data into usable objects for further operations. The data is then partitioned for distributed computation using mapPartitions, and the training instances are processed through the map function. Finally, the treeAggregate operation is used to aggregate results from partitions, ensuring efficient model training across the distributed dataset. This workflow highlights PySpark's ability to parallelize and optimize tasks for large-scale machine learning.

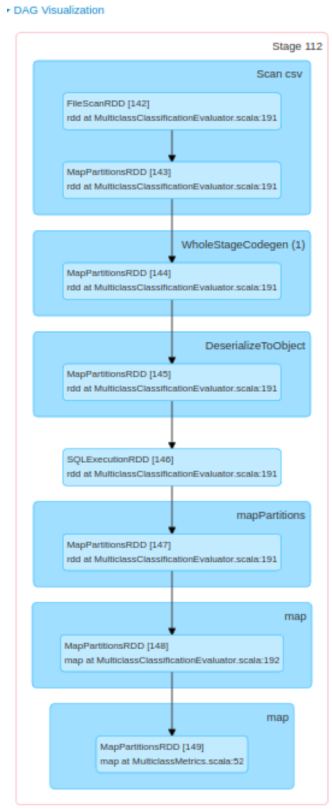


Fig. 32. DAG LR2

This DAG represents the evaluation process of the Logistic Regression model using PySpark. It starts with reading the dataset through the FileScanRDD, followed by distributed data processing using mapPartitions. The WholeStageCodegen optimizes execution by generating efficient code for processing. Data is deserialized into objects using DeserializeToObject, enabling transformations like mapPartitions and map for calculating evaluation metrics. Finally, metrics such as accuracy, precision, recall, and F1-score are computed using the MulticlassMetrics evaluator. This workflow highlights PySpark's efficiency in evaluating machine learning models on distributed datasets.

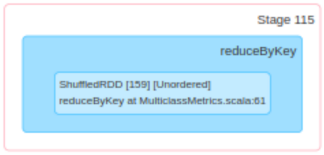


Fig. 33. DAG LR3

This DAG stage represents the reduceByKey operation, where data is shuffled and aggregated by key to compute metrics such as confusion matrix values. The ShuffledRDD ensures distributed aggregation, enabling efficient computation across partitions.

B. Decision Tree

Justification for Choosing Decision Tree: The Decision Tree was selected for its simplicity and interpretability, which allows for easy diagnosis of cobot operation failures. Additionally, its ability to reveal key decision-making patterns makes it a practical choice for this task.

Model Training: The Decision Tree model was trained using an 80-20 split, with 80% of the data used for training and 20% for testing. Default parameters were used for the Decision Tree Classifier, focusing on leveraging PySpark's distributed processing capabilities to handle large datasets.

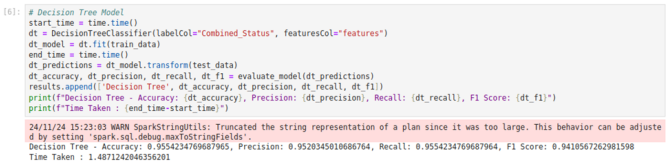


Fig. 34. Decision Tree model

Effectiveness of the Algorithm: The model achieved an overall accuracy of 95.54%, reflecting strong classification performance. Aggregated precision (0.9552), recall (0.9554), and F1-Score (0.9411) metrics demonstrate the model's reliability in making accurate predictions and maintaining balance between precision and recall. The efficiency of PySpark enabled the model to be trained and evaluated in approximately 1.487 seconds, showcasing the advantages of distributed processing for large datasets.

Performance Metrics:

- **Overall Accuracy:** 95.54%, reflecting reliable classification performance
- **Precision:** 0.9552, highlighting the model's ability to predict positive cases accurately
- **Recall:** 0.9554, indicating its strong sensitivity in identifying true positives.
- **F1-Score:** 0.9411, showing an effective balance between precision and recall.

Gained Intelligence/Insights: The Decision Tree provided insights into the influence of joint speeds and currents on grip status, revealing critical relationships for improving cobot operations.

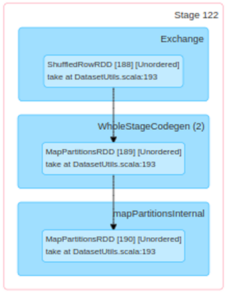
Time Taken: 1.4871242046356201 seconds.

Comparison with Previous Model in Phase2:

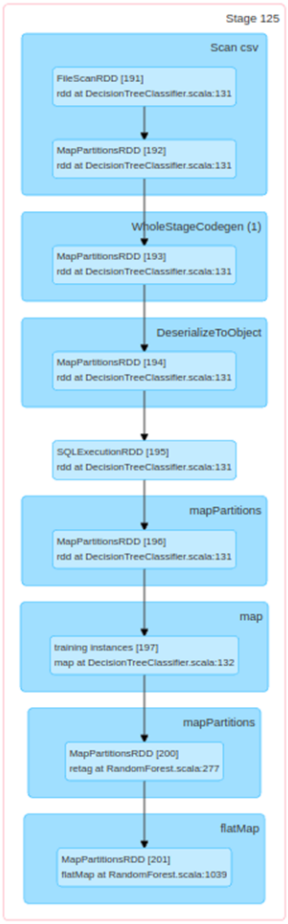
The Phase 2 Decision Tree model (tuned) achieved slightly higher accuracy (96% vs. 95.54% for PySpark) and slightly better weighted recall (0.96 vs. 0.9554) and F1-Score (0.95 vs. 0.9411). However, the PySpark model demonstrated better precision (0.9523 vs. 0.95) and significantly reduced training time (1.487 seconds vs. unreported time in Phase 2 but typically slower for larger datasets). While both models showed strong performance for majority classes, the Phase 2 model struggled with minority classes, with lower precision and recall for those groups. PySpark's distributed processing ensured

scalability and efficiency, making it better suited for large datasets, whereas the Phase 2 model achieved slightly higher overall metrics but was less effective in handling imbalanced data.

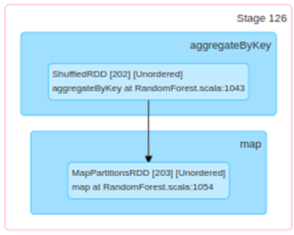
DAG Visualization for Decision Tree Model:



This DAG for Stage 122 starts by shuffling the data to redistribute it across partitions for efficient processing. It then optimizes the computation with code generation (WholeStageCodegen) and finishes by applying transformations to each partition individually to complete the task.



This DAG for Stage 125 begins with loading data from a CSV (FileScanRDD) and applying optimizations like WholeStageCodegen. It then processes data partitions, maps training instances, and performs transformations for Random Forest computations.



This stage aggregates data by key using aggregateByKey to combine results from partitions, followed by a map transformation to process the aggregated data.

This stage uses reduceByKey to merge values with the same key, likely for computing metrics like accuracy or other evaluation statistics for the Random Forest model.

C. Random Forest

Justification for Choosing Random Forest: Random Forest was chosen for its ensemble approach, which reduces overfitting and is well-suited for predicting both common and rare events in UR3 cobot operations. Its ability to aggregate decisions from multiple trees ensures robust and accurate predictions.

Model Training: The Random Forest model was trained using PySpark's MLlib framework with default parameters, focusing on leveraging the distributed computation capabilities of Spark for efficient training. The training process followed an 80-20 split, with 80% of the data used for training and 20% for testing.

Fig. 35. Random Forest model

Effectiveness of the Algorithm: The Random Forest model achieved an accuracy of 94.79%, showcasing strong performance across most classes. Its high recall indicates effectiveness in identifying true positives, while precision was slightly lower, suggesting some challenges with false positives. Despite this, the model delivered balanced predictions suitable for operational tasks.

Performance Metrics:

- **Overall Accuracy:** 94.79%, reflecting reliable classification performance.
- **Precision:** 0.8987, highlighting the model's ability to make accurate positive predictions.
- **Recall:** 0.9479, showing strong sensitivity in identifying true cases.
- **F1-Score:** 0.9227, capturing the overall balance between precision and recall.

Gained Intelligence/Insights: The PySpark Random Forest model highlighted the significance of features such as joint speeds and currents as key factors influencing grip retention and failure in cobot operations. By leveraging its ensemble approach, the model provided consistent and interpretable insights into the factors affecting cobot performance, making it valuable for diagnostics and operational decision-making.

Time Taken: 1.0409176349639893 seconds.

Comparison with Previous Model in Phase2:

The Phase 2 Random Forest model achieved slightly higher accuracy (96% vs. 94.79%) and better metrics such as precision (0.96 vs. 0.8987), recall (0.96 vs. 0.9479), and F1-Score (0.95 vs. 0.9227). However, the PySpark model outperformed in time taken, completing training and evaluation in 1.04 seconds, compared to 0.596 seconds for the Phase 2 model. While the Phase 2 model benefited from hyperparameter tuning (n_estimators=50) and excelled in majority class predictions, it struggled significantly with minority classes (e.g., Class 2 and Class 3, with recall as low as 0.08 or 0.00). In contrast, the PySpark model provided more consistent, scalable performance, making it better suited for large datasets.

DAG Visualization for Random Forest Model:

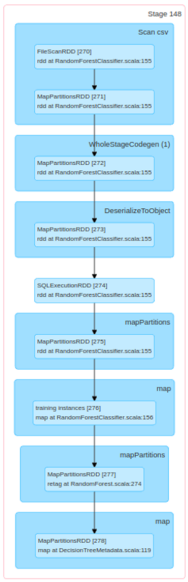


Fig. 36. DAG rf1

This DAG for Stage 148 begins with reading data from a CSV file (FileScanRDD), followed by optimizations like WholeStageCodegen and deserialization. It processes data using partition-based operations (mapPartitions) and trains instances (map), completing the Random Forest training process.

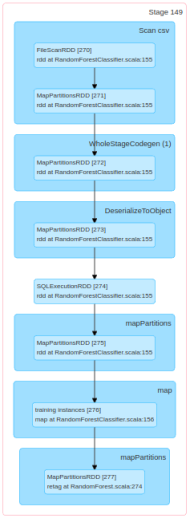


Fig. 37. DAG rf2

This DAG for Stage 149 starts with reading data from a CSV (FileScanRDD), applies optimizations like WholeStageCodegen and deserialization, and processes data using partition-based operations (mapPartitions). It then maps the training instances (map) and continues with further partition-based processing to complete the task.

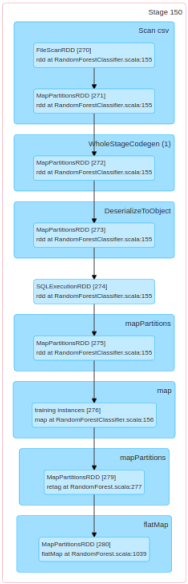


Fig. 38. DAG rf3

This DAG for stage 150 starting with a CSV scan, optimizations, and partition-based operations. Unlike Stages 148 and 149, it adds a flatMap operation at the end, likely to finalize transformations for Random Forest ensemble outputs or further distribute tasks across partitions.

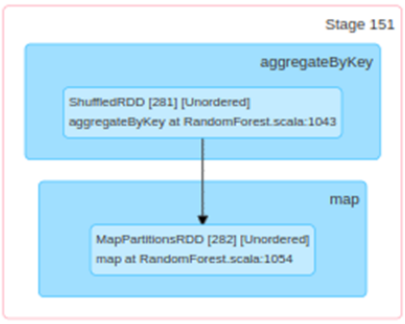


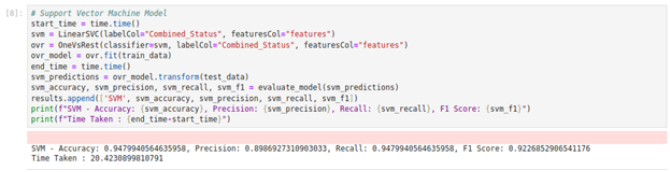
Fig. 39. DAG rf4

This DAG for Stage 151 uses aggregateByKey to combine results across partitions, followed by a map to refine outputs for final evaluation or storage.

D. Support Vector Machine (SVM)

Justification for Choosing SVM: SVM was chosen for its ability to handle complex relationships and high-dimensional data effectively, making it suitable for predicting cobot operational states. The use of the One-vs-Rest approach in PySpark ensures effective handling of multi-class classification tasks. However, as a linear kernel was used (LinearSVC), the model is best suited for capturing linear patterns rather than non-linear ones.

Model Training: The PySpark SVM model was trained using the LinearSVC classifier with default parameters, without explicit hyperparameter tuning. The One-vs-Rest approach was used to handle multi-class classification. The training process followed an 80-20 split, with 80% of the data used for training and 20% for testing. PySpark's distributed computing capabilities enabled efficient training on the dataset, though the training time was higher compared to other models due to the computational intensity of SVM.



```
[8]: # Support Vector Machine Model
start_time = time.time()
svm = LinearSVC(labelCol="Combined_Status", featuresCol="features")
ovr = OneVsRest(classifier=svm, labelCol="Combined_Status", featuresCol="features")
ovr_model = ovr.fit(train_data)
end_time = time.time()
svm_predictions = ovr_model.transform(test_data)
svm_accuracy, svm_precision, svm_recall, svm_f1 = evaluate_model(svm_predictions)
results.append(('SVM', svm_accuracy, svm_precision, svm_recall, svm_f1))
print(f"SVM - Accuracy: {svm_accuracy}, Precision: {svm_precision}, Recall: {svm_recall}, F1 Score: {svm_f1}")
print(f"Time Taken : {end_time-start_time}")

SVM - Accuracy: 0.9479940564635958, Precision: 0.8986927310903033, Recall: 0.9479940564635958, F1 Score: 0.9226852906541176
Time Taken : 20.4230899810791
```

Fig. 40. SVM model

Effectiveness of the Algorithm: The SVM model achieved an accuracy of 94.79%, demonstrating strong overall performance. Its high recall reflects effectiveness in identifying true positives, while slightly lower precision indicates challenges in handling false positives. The model showed a good balance between precision and recall but had a higher computational cost compared to other models.

Performance Metrics:

- **Overall Accuracy:** 94.79%, reflecting reliable classification performance.
- **Precision:** 0.8987, reflecting the model's ability to predict positive cases accurately.
- **Recall:** 0.9479, showing the model's effectiveness in capturing most true positives.
- **F1-Score:** 0.9227, indicating a balanced trade-off between precision and recall.

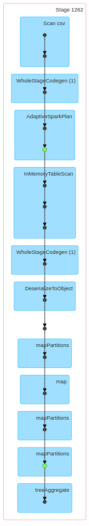
Gained Intelligence/Insights: The SVM model, using a linear kernel, captured key linear relationships between features like joint speeds and grip failures, offering insights into cobot operations. While effective for majority classes, it faced challenges with minority classes and required higher computational time (20.42 seconds). PySpark's distributed processing ensured scalability and efficient handling of the dataset.

Time Taken: 20.4230899810791 seconds.

Comparison with Previous Model in Phase2:

The Phase 2 SVM model achieved slightly better accuracy (95% vs. 94.79% for PySpark SVM), with marginally higher precision (0.91 vs. 0.8987), recall (0.95 vs. 0.9479), and F1-Score (0.93 vs. 0.9227), benefiting from hyperparameter tuning using an RBF kernel. However, it struggled significantly with minority classes, as Classes 1, 2, and 3 had precision and recall of 0.00, indicating poor performance for rare events. The PySpark SVM model, using a linear kernel and default parameters, provided balanced performance across classes and was significantly faster, completing training in 20.42 seconds compared to 71.80 seconds for Phase 2. While the Phase 2 model showed slightly better metrics for the majority class, the PySpark SVM model's computational efficiency and scalability make it more practical for large datasets and real-world applications.

DAG Visualization for SVM Model:



This DAG for Stage 1262 starts with reading a CSV file, optimizing queries using adaptive execution, and scanning in-memory data. It processes partitions with map operations and combines the results using treeAggregate for the final output.

This DAG spans two stages: Stage 1296 processes the CSV file with multiple optimization layers (WholeStageCodegen and BatchEvalPython), partition-based operations (mapPartitions and map), and deserialization. Stage 1297 uses reduceByKey to aggregate and finalize the results from Stage 1296.

E. Naive Bayes:

Justification for Choosing Naive Bayes: Naive Bayes was selected for its simplicity and efficiency in handling high-dimensional data. In this project, it is used to quickly classify grip statuses based on joint speeds and currents, providing a baseline probabilistic approach for cobot operation analysis. Its fast computation makes it ideal for real-time or iterative evaluations during early development stages.

Model Training: The Naive Bayes model was trained using PySpark's MLlib framework with default parameters, as it is inherently less sensitive to hyperparameter tuning compared to other models. The training process followed an 80-20 split, with 80% of the data used for training and 20% for testing. PySpark's distributed processing enabled efficient training and evaluation of the model.

```
[9]: # Naive Bayes Model
start_time = time.time()
nb = NaiveBayes(labelCol="Combined_Status", featuresCol="features")
nb_model = nb.fit(train_data)
end_time = time.time()
nb_predictions = nb_model.transform(test_data)
nb_accuracy, nb_precision, nb_recall, nb_f1 = evaluate_model(nb_predictions)
results.append(['Naive Bayes', nb_accuracy, nb_precision, nb_recall, nb_f1])
print(f"Naive Bayes - Accuracy: {nb_accuracy}, Precision: {nb_precision}, Recall: {nb_recall}, F1 Score: {nb_f1}")
print(f"Time Taken : {end_time-start_time}")

Naive Bayes - Accuracy: 0.9479940564635958, Precision: 0.8986927310903033, Recall: 0.9479940564635958, F1 Score: 0.9226852906541176
Time Taken : 0.4988405704498291
```

Fig. 41. Naive Bayes model

Effectiveness of the Algorithm: The Naive Bayes model achieved an accuracy of 94.79%, demonstrating strong performance for the majority class but facing challenges with minority events. Its computational efficiency, with a time of 0.4988 seconds, makes it ideal for quick evaluations and iterative analysis.

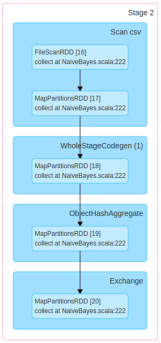
Performance Metrics:

- **Overall Accuracy:** 94.79%, reflecting reliable classification performance.
- **Precision:** 0.8987, highlighting the model's ability to predict positive cases accurately.
- **Recall:** 0.9479, indicating its effectiveness in identifying most true positives.
- **F1-Score:** 0.9227, reflecting a balanced trade-off between precision and recall.

Gained Intelligence/Insights: The Naive Bayes model provided probabilistic insights into the relationships between features like joint speeds and grip status. Its efficient implementation enabled quick evaluations and highlighted the influence of these features on cobot operations.

Time Taken: 0.4988405704498291 seconds.

DAG Visualization for Naive Bayes Model:



This DAG for Stage 2 starts with reading a CSV file, optimizes processing with WholeStageCodegen, aggregates data using ObjectHashAggregate, and shuffles it across partitions with Exchange.

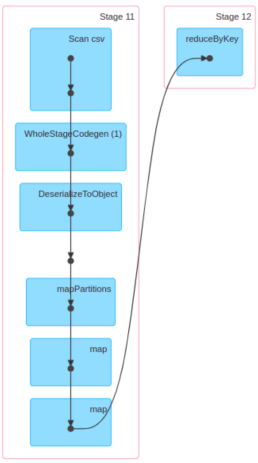


Fig. 42. DAG NB2

This DAG represents the execution stages for the PySpark Naive Bayes model. Stage 11 starts by scanning the CSV file, followed by optimizations in WholeStageCodegen, deserializing the objects, and partitioning the data using mapPartitions

for parallel processing. Subsequent map operations process the data and prepare it for classification. In Stage 12, the reduceByKey operation aggregates results, such as predictions, to finalize the computation efficiently. This distributed workflow showcases Spark's ability to handle large datasets through parallelization and optimized task execution.

F. Gradient Boosting

Justification for Choosing Gradient Boosting: Gradient Boosting was selected for its iterative nature, where each tree corrects the errors of previous ones, making it effective for improving prediction accuracy. In this project, it is well-suited for analyzing cobot operations by achieving high accuracy for both common and rare events. Its ability to handle imbalanced datasets and focus on hard-to-predict samples makes it ideal for tasks requiring precision and reliability.

Model Training: The Gradient-Boosted Tree (GBT) model was trained using PySpark's MLlib framework with default parameters, without explicit hyperparameter tuning. The One-vs-Rest approach was used to handle multi-class classification. The training process followed an 80-20 split, with 80% of the data used for training and 20% for testing. PySpark's distributed processing capabilities enabled efficient training and evaluation on the dataset.

Fig. 43. Gradient Boosting

Effectiveness of the Algorithm: The Gradient Boosting model achieved an accuracy of 95.69%, demonstrating strong performance across both majority and minority classes. Its iterative nature allowed it to effectively minimize errors from previous iterations, resulting in balanced and robust predictions. While it excelled in handling both common and rare events, its slightly higher computational time reflects the complexity of the model.

Performance Metrics:

- **Overall Accuracy:** 95.69%, reflecting reliable classification performance
- **Precision:** 0.9588, highlighting the model's ability to predict positive cases accurately
- **Recall:** 0.9569, indicating its strong sensitivity in identifying true positives.
- **F1-Score:** 0.9421, showing an effective balance between precision and recall.

Gained Intelligence/Insights: The Gradient Boosting model provided valuable insights into the influence of joint currents and speeds on grip status. Its iterative approach effectively minimized errors, uncovering key patterns and relationships that contribute to grip retention and failure. This made it particularly useful for identifying subtle and complex interactions within the dataset.

Time Taken: 14.378748178482056 seconds.

Comparison with Previous Model in Phase2:

The Phase 2 Gradient Boosting model achieved slightly higher accuracy (96% vs. 95.69%), recall (0.96 vs. 0.9569), and F1-Score (0.95 vs. 0.9421) but struggled with minority classes. The PySpark Gradient Boosting model, while slightly lower in some metrics, had better precision (0.9588 vs. 0.95) and completed training in 14.38 seconds, significantly faster than the 2526.60 seconds of the Phase 2 model. Overall, PySpark's efficiency and scalability make it better suited for large datasets, despite Phase 2's slightly better performance in major classes.

DAG Visualization for Gradient Boosting Model:

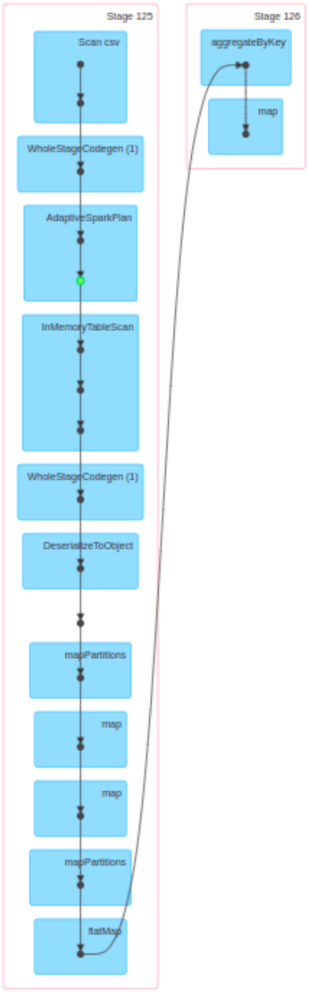


Fig. 44. DAG GB

This DAG illustrates the stages of data processing for the Spark model. Stage 125 begins by scanning the CSV file and optimizing data processing with WholeStageCodegen, AdaptiveSparkPlan, and InMemoryTableScan. In Stage 126, the aggregateByKey operation aggregates data by keys, followed

by a map operation for parallel data transformation. These stages showcase Spark's distributed processing capabilities for handling large datasets efficiently.

Fig. 45. DAG GB2

This DAG represents the execution flow of a Spark model training process. Stage 505 starts with scanning the CSV file and optimizing the execution using WholeStageCodegen, AdaptiveSparkPlan, and InMemoryTableScan for efficient data processing. In Stage 506, the reduceByKey operation aggregates data by keys, followed by a map operation for data transformation. This distributed processing workflow ensures efficient handling of large datasets.

IX. Final Model Comparison

Using PySpark for our models, we observed an increase in overall time taken and a slight decrease in accuracy. This is likely because, compared to the scale at which PySpark excels, our dataset of approximately 7,000 rows is relatively small. PySpark's true usefulness is typically observed with much larger datasets, in the range of 100,000 rows or more.

Despite the increased time for simpler models, the accuracy remained close to previous results, which might be attributed to randomness in training. Interestingly, for complex models like Decision Tree and SVM, we observed a reduction in time taken, demonstrating PySpark's ability to optimize computations for resource-intensive tasks.

In conclusion, while PySpark showed potential for scaling complex models, its benefits are more pronounced with larger datasets. For small datasets like ours, traditional tools might still be more efficient in terms of time and resource utilization.

We could not find one to one implementation of KNN in pyspark-ML so we decided to substituite it with naive-bayes because of the reasons we mentioned above, but we have compared all the other models with their previous results.

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
results_df = pd.DataFrame(results, columns=['Model', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])
sns.set(style="whitegrid")
fig, axes = plt.subplots(2, 2, figsize=(15, 10))
sns.barplot(ax='Accuracy', y='Model', data=results_df, ax=axes[0, 0], hue='Model', palette="viridis", legend=False)
axes[0, 0].set_title('Model Accuracy')
axes[0, 0].set_xlim(0.9, 1)
axes[0, 0].set_ylim(-0.5, len(results_df)-0.5)

# Precision Plot
sns.barplot(ax='Precision', y='Model', data=results_df, ax=axes[0, 1], hue='Model', palette="viridis", legend=False)
axes[0, 1].set_title('Model Precision')
axes[0, 1].set_xlim(0.8, 1)
axes[0, 1].set_ylim(-0.5, len(results_df)-0.5)

# Recall Plot
sns.barplot(ax='Recall', y='Model', data=results_df, ax=axes[1, 0], hue='Model', palette="viridis", legend=False)
axes[1, 0].set_title('Model Recall')
axes[1, 0].set_xlim(0.9, 1)
axes[1, 0].set_ylim(-0.5, len(results_df)-0.5)

# F1 Score Plot
sns.barplot(ax='F1 Score', y='Model', data=results_df, ax=axes[1, 1], hue='Model', palette="viridis", legend=False)
axes[1, 1].set_title('Model F1 Score')
axes[1, 1].set_xlim(0.9, 1)
axes[1, 1].set_ylim(-0.5, len(results_df)-0.5)

plt.tight_layout()
plt.show()
```

Fig. 46. Final model comparision

This helps us compare how well different models perform by creating clear and simple visualizations of their key metrics: Accuracy, Precision, Recall, and F1 Score. Each model's performance is shown as bars in separate plots, making it easy to see which models perform better overall or in specific areas. These visuals allow us to quickly identify the strengths and weaknesses of each model, helping us better understand their effectiveness and make informed decisions.

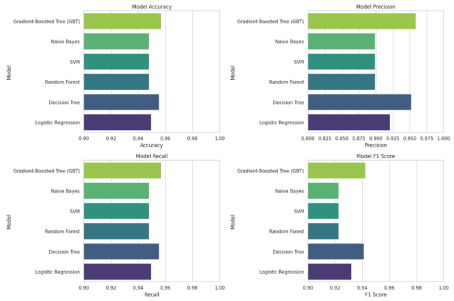


Fig. 47. Plot of models

The bar plots compare the performance of different machine learning models based on Accuracy, Precision, Recall, and F1 Score for predicting the operational states of the UR3 cobot, such as grip retention and failure. Gradient Boosted Tree (GBT) emerges as the top-performing model across all metrics, achieving the highest accuracy of approximately 96%. This indicates its strong capability to handle both common and rare events effectively.

Naive Bayes, SVM, and Random Forest also demonstrate strong performance, achieving accuracies close to GBT and showing balanced Precision, Recall, and F1 Scores. These models highlight their ability to manage class imbalances while maintaining consistent predictions.

Decision Tree and Logistic Regression, while slightly lower in performance, still achieve commendable results, with accuracies above 94%. These models are simpler but effective, making them suitable for tasks where interpretability or computational speed is crucial.

Overall, ensemble-based models like GBT and Random Forest stand out for their robustness, making them the best choices for predictive maintenance and operational diagnostics in the UR3 cobot system. These models excel at delivering precise, balanced predictions for both frequent and rare events.

X. Bonus Task

This tool is designed to simplify the process of exploring datasets, training machine learning models, and generating insights through intuitive visualizations.

As you can see, our user interface is clean, easy to navigate, and designed for non-technical users. At the top, we have the title and a brief description of the app's purpose. The navigation sections guide the user step-by-step through uploading their dataset, preprocessing it, training a model, and making predictions. Each section is interactive and clearly labeled to make the process seamless. Now let's demonstrate how users can input their own data. we can upload a sample CSV file by clicking the "Browse" button. Once the file is uploaded,

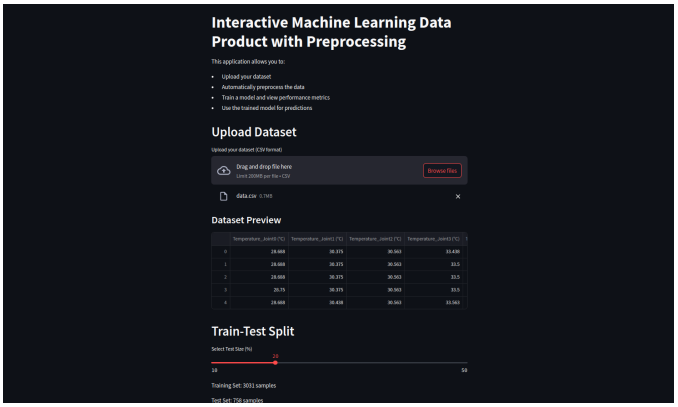


Fig. 48. UI

(a) Working User Interface

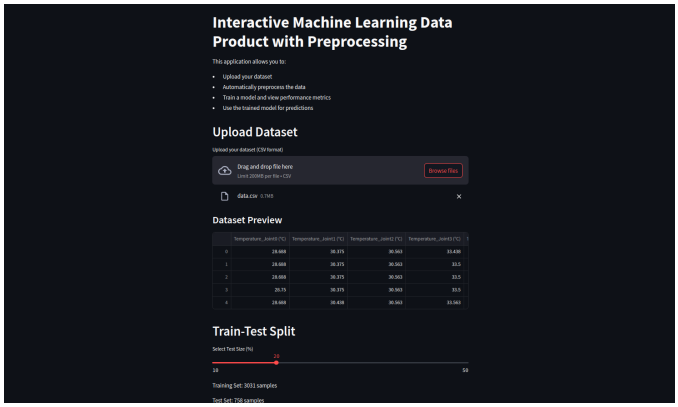


Fig. 49. UI

the dataset is displayed in the preview section, allowing users to quickly verify their data. The app automatically handles missing values and identifies numerical features for analysis, ensuring that users don't need to worry about preprocessing, it's handled behind the scenes. Users can specify the test set size using the slider; for instance, it's defaultly set it to 20%. The app splits the data into training and test sets and displays the sample sizes. Moving on to model training and feedback, the app uses a Random Forest Classifier by default but is flexible to include other models. The training process is automated, and once completed, it displays the model's accuracy score. Users receive a detailed classification report with precision, recall, and F1-score for each class. To make the feedback more visual, we also display the ROC curve and AUC score, which measure the model's performance in distinguishing between classes. Users can interact with the visualizations by zooming in or focusing on specific aspects, making it easier to interpret results. Finally, users can make predictions with this tool. The app displays all numerical features in a grid layout for easy input. Users can enter values for each feature. After entering the data, clicking the "Predict" button runs the trained model to classify the input. The app then displays the predicted class instantly. This feature helps users apply the model to real-world scenarios and answer specific questions using their data. To conclude, this product empowers users by simplifying data preprocessing and model training, providing detailed and intuitive feedback through metrics and visualizations, and offering actionable insights with prediction capabilities. We're proud of the impact this tool can make in democratizing machine learning for everyday users.

(b) Input your own data

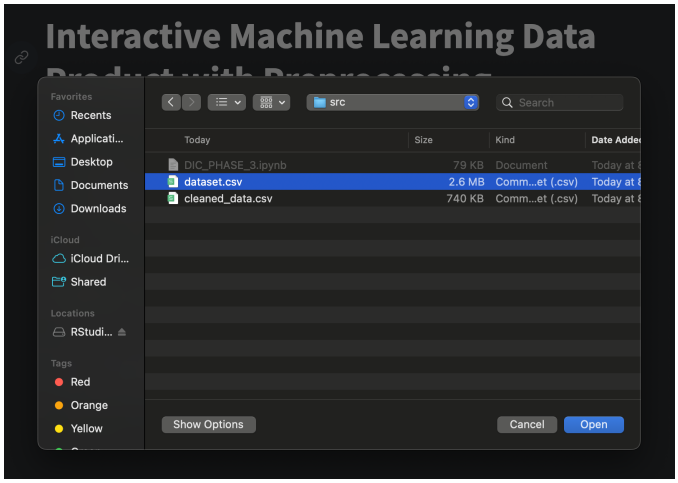


Fig. 50. upload Image

After clicking the browse files button user will get an option to upload their own dataset.

INSTRUCTIONS TO RUN THE APPLICATION :

navigate to the directory in the command prompt and run

```
pip install streamlit pandas numpy scikit-learn seaborn matplotlib

streamlit run bonus.py
```

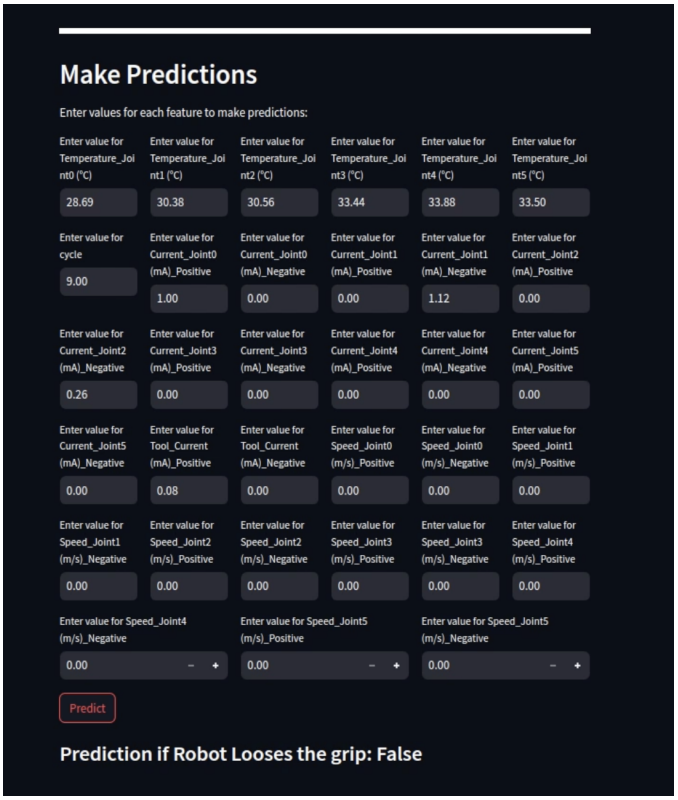


Fig. 51. Enter Caption

(c) Feedback and visualization : The image below shows the feedback model provided for the given dataset; it shows the accuracy, precision and various other metrics, which show how reliable predictions are that the model loses grip

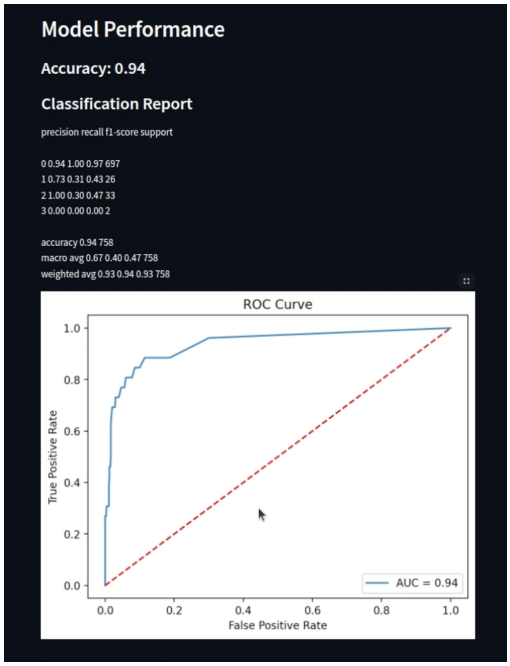


Fig. 52. Feedback and visuvalization

XI. References

- PySpark ML Logistic Regression
- PySpark ML Naive Bayes
- PySpark MLlib Gradient Boosted Trees
- PySpark MLlib Random Forest
- PySpark MLlib Decision Tree
- PySpark MLlib SVM
- UR3 Cobotops - UCI Machine Learning Repository
- https://spark.apache.org/docs/latest/api/python/index.html
- https://spark.apache.org/docs/latest/ml-guide.html
- C. O'Neill and R. Schutt. Doing Data Science., O'Reilly. 2013.
- J. VanderPlas, Python Data Science Handbook., O'Reilly. 2016.

Peer Evaluation Form for Final Group Work

CSE 487/587B

- Please write the names of your group members.

Group member 1 : Mohan Teja Guddimettla

Group member 2 : Mounika Reddy Katikam Vekata

Group member 3 : Sudheer Kumar Reddy Batthina

- Rate each groupmate on a scale of 5 on the following points, with 5 being HIGHEST and 1 being LOWEST.

Evaluation Criteria	Group member 1	Group member 2	Group member 3
How effectively did your group mate work with you?	5	5	5
Contribution in writing the report	5	5	5
Demonstrates a cooperative and supportive attitude.	5	5	5
Contributes significantly to the success of the project .	5	5	5
TOTAL	25	25	25

Also please state the overall contribution of your teammate in percentage below, with total of all the three members accounting for 100% (33.33+33.33+33.33 ~ 100%) :

Group member 1 : 33.33

Group member 2 : 33.33

Group member 3 : 33.33