BIG DATA PROJECT REPORT

Ashwini Choudhary

Master 2 Big Data and Business Analytics at CYTech, Cergy, France

Abstract. This project was based on fitbit data published on Kaggle,where I tried to make classification based on running and fitness activities if the person is Fit, Unfit or Moderately Fit.The analysis was conducted using Databricks, with PySpark and Python as the primary programming tools. A significant challenge involved integrating and preprocessing multiple datasets to create a coherent and meaningful dataset for analysis. This required meticulous handling of data inconsistencies and the application of feature engineering techniques to extract actionable insights. The resulting model provides a robust framework for assessing fitness levels, showcasing the potential of big data and machine learning in personal health analytics.

Keywords: DataBricks · Pyspark · Machine Learning · Python · Big Data · Kaggle · Pandas.

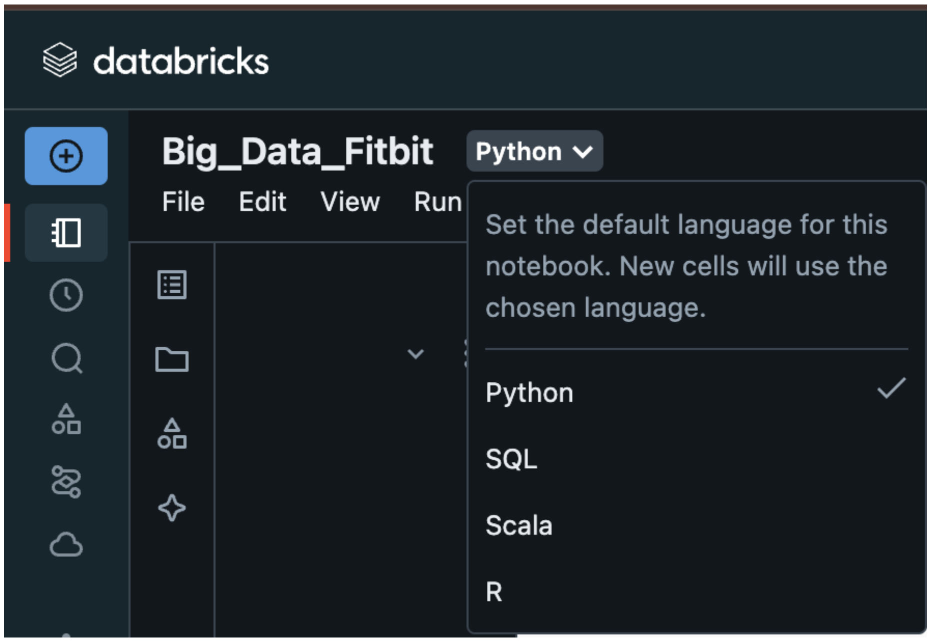
1 INTRODUCTION

I worked with the Fitbit dataset, which comprised multiple CSV files, including *dailyActivitymerged.csv*, *dailyCaloriesmerged.csv*, *heartratesecondsmerged.csv*, and *sleepDaymerged.csv*. The goal was to consolidate these files into a single cohesive dataframe that captured comprehensive information about individuals' activities. To achieve this, I used Python and the Pandas library to handle data manipulation and merging. The primary columns used for merging were ActivityDay and Time, as these were common across most of the datasets. This integration enabled a unified analysis of activity patterns and fitness metrics, forming the foundation for further exploration and modeling.

2 Working with DataBricks and Jupyter Notebook

2.1 DataBricks

I utilized Databricks as the cloud platform to execute this project, leveraging its powerful distributed computing capabilities. By running Spark and PySpark on Databricks, I significantly reduced the processing time compared to running the same operations on a local machine. Databricks also offered exceptional flexibility in choosing programming languages for data analysis, allowing seamless integration of SQL, PySpark, and Python. This versatility, combined with the scalability of the cloud platform, enabled efficient data processing and analysis, making it an ideal choice for handling the large and complex Fitbit dataset.



Set the default language for this notebook. New cells will use the chosen language.

Fig. 1. Data Bricks Depiction

2.2 Jupyter Notebook

I utilized Jupyter Notebook, launched via Anaconda Navigator, as the primary platform for conducting comprehensive data analysis in this project. Jupyter Notebook provided an interactive and user-friendly environment, enabling me to write, execute, and document Python code seamlessly in a single interface. Through the powerful data manipulation and transformation capabilities of Pandas, I effectively handled tasks such as merging datasets, cleaning data, and performing exploratory data analysis.Anaconda Navigator played a crucial role in streamlining the setup process by managing dependencies and ensuring a stable environment for the project. This eliminated the need for manual library installations and compatibility checks, allowing me to focus entirely on the analysis. The combination of these tools provided a highly efficient workflow for working with the Fitbit dataset, supporting iterative development, visualization, and debugging.

3 Problem Statement

For this project, I utilized the Fitbit Fitness Tracker dataset from Kaggle and transformed it into a classification problem. The goal was to predict an individual's fitness level based on activity metrics. I created a target column named Fitness, which categorizes individuals into one of three groups: Fit, Unfit, or

Moderately Fit. The challenge involved analyzing the provided features, such as daily activity, calorie expenditure, heart rate, and sleep data, to build a predictive model capable of accurately classifying a person's fitness level. This problem highlights the potential of data-driven approaches to provide actionable insights into personal health and activity tracking.

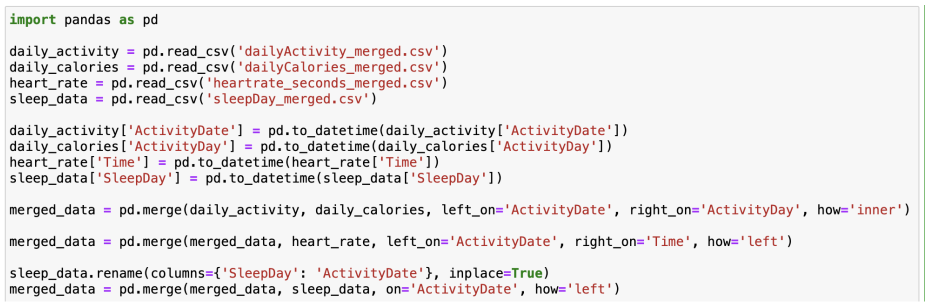
4 Preprocessing in Jupyter Notebook

To prepare the data for analysis, I used Python in Jupyter Notebook to merge the various datasets and perform exploratory data analysis (EDA). During this phase, I handled tasks such as cleaning, transforming, and integrating the data into a cohesive final dataset. This consolidated dataset was then imported into Databricks, where I leveraged its powerful distributed computing capabilities for further analysis and model development.

4.1 Merging CSV files

The data merging process involved combining four datasets: daily activity, calories, heart rate, and sleep data. Each dataset was loaded and relevant date/time columns were standardized using $pd.todatetime()$.

The daily activity and calories datasets were merged using an inner join on their date columns. The resulting dataset was then merged with heart rate data using a left join, aligning by date and time. Finally, the sleep data was merged using a left join on the date column, after renaming it for consistency. This created a single consolidated dataset containing activity, calorie, heart rate, and sleep information for further analysis.

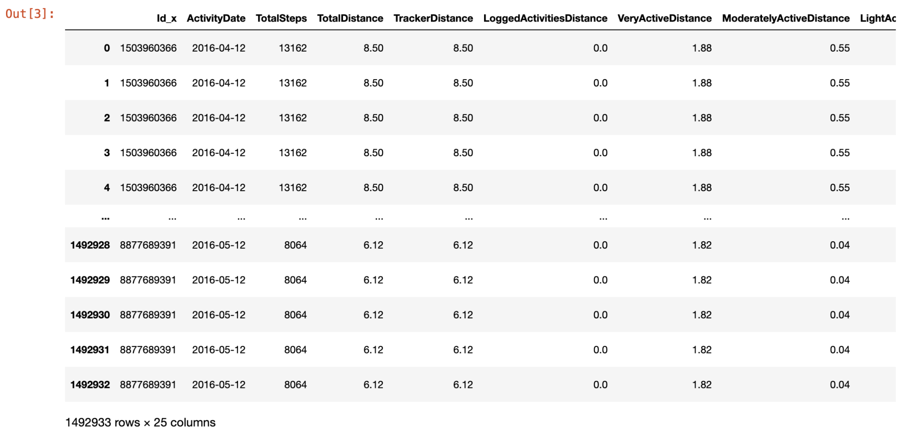


```
import pandas as pd

daily_activity = pd.read_csv('dailyActivity_merged.csv')
daily_calories = pd.read_csv('dailyCalories_merged.csv')
heart_rate = pd.read_csv('heartrate_seconds_merged.csv')
sleep_data = pd.read_csv('sleepDay_merged.csv')

daily_activity['ActivityDate'] = pd.to_datetime(daily_activity['ActivityDate'])
daily_calories['ActivityDay'] = pd.to_datetime(daily_calories['ActivityDay'])
heart_rate['Time'] = pd.to_datetime(heart_rate['Time'])
sleep_data['SleepDay'] = pd.to_datetime(sleep_data['SleepDay'])

merged_data = pd.merge(daily_activity, daily_calories, left_on='ActivityDate', right_on='ActivityDay', how='inner')

merged_data = pd.merge(merged_data, heart_rate, left_on='ActivityDate', right_on='Time', how='left')

sleep_data.rename(columns={'SleepDay': 'ActivityDate'}, inplace=True)
merged_data = pd.merge(merged_data, sleep_data, on='ActivityDate', how='left')
```

Fig. 2. Merging CSV files

4.2 DataFrame after Merging

My resultant dataframe had 1492933 rows × 25 columns and my dataframe was inconsistent.In the coming pages you'll see how I managed to make it consistent. It had duplicate values in huge number.



Out[3]:

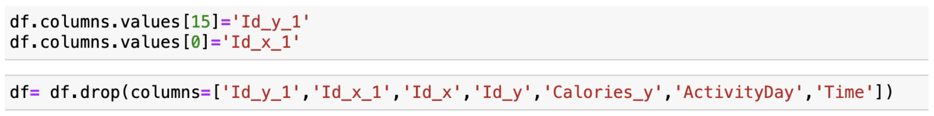
	Id_x	ActivityDate	TotalSteps	TotalDistance	TrackerDistance	LoggedActivitiesDistance	VeryActiveDistance	ModeratelyActiveDistance	LightAc
0	1503960366	2016-04-12	13162	8.50	8.50	0.0	1.88	0.55	
1	1503960366	2016-04-12	13162	8.50	8.50	0.0	1.88	0.55	
2	1503960366	2016-04-12	13162	8.50	8.50	0.0	1.88	0.55	
3	1503960366	2016-04-12	13162	8.50	8.50	0.0	1.88	0.55	
4	1503960366	2016-04-12	13162	8.50	8.50	0.0	1.88	0.55	
...	...	...	...	...	...	...	...	...	
1492928	8877689391	2016-05-12	8064	6.12	6.12	0.0	1.82	0.04	
1492929	8877689391	2016-05-12	8064	6.12	6.12	0.0	1.82	0.04	
1492930	8877689391	2016-05-12	8064	6.12	6.12	0.0	1.82	0.04	
1492931	8877689391	2016-05-12	8064	6.12	6.12	0.0	1.82	0.04	
1492932	8877689391	2016-05-12	8064	6.12	6.12	0.0	1.82	0.04	

1492933 rows × 25 columns

Fig. 3. DataFrame after Merging

4.3 Dropping the columns

I dropped the unnecessary columns which made no sense for my model it included columns like Id,Datetime and Time.



```
df.columns.values[15]='Id_y_1'
df.columns.values[0]='Id_x_1'
```

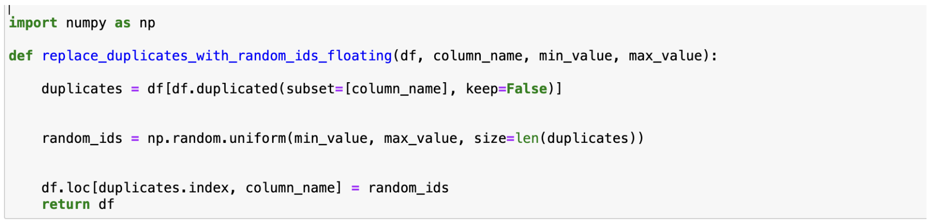
```
df= df.drop(columns=['Id_y_1','Id_x_1','Id_x','Id_y','Calories_y','ActivityDay','Time'])
```

Fig. 4. Dropping the columns

4.4 Imputing Random floating values

In order to address the issue of duplicate values in columns containing floating-point numbers, I implemented a solution where duplicate entries were replaced with randomly generated float values. The random values were imputed within the range defined by the minimum and maximum values of the respective columns. This ensured that the data remained within realistic bounds, preserving the consistency and integrity of the dataset.

To achieve this, I identified the duplicate entries in the dataset and used the random.uniform() function to generate new values within the specified range. These new values replaced the duplicate entries, thus eliminating redundancy while maintaining the validity of the data. The approach was applied to several columns, such as TotalDistance, TrackerDistance, and others, which contain continuous floating-point values, ensuring that the dataset was cleaned and ready for further analysis.

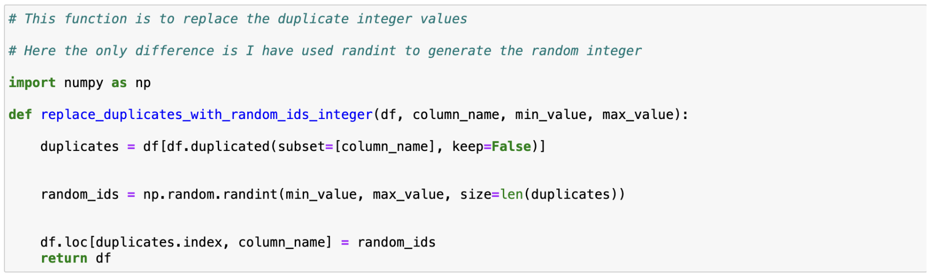


```python
import numpy as np

def replace_duplicates_with_random_ids_floating(df, column_name, min_value, max_value):

    duplicates = df[df.duplicated(subset=[column_name], keep=False)]


    random_ids = np.random.uniform(min_value, max_value, size=len(duplicates))


    df.loc[duplicates.index, column_name] = random_ids
    return df
```

Fig. 5. Function to impute floating values

4.5 Imputing Random Integer values

To handle duplicate values in columns containing integer data, I applied a similar approach to the one used for floating-point columns. Duplicate entries were identified, and they were replaced with randomly generated integer values. These new values were imputed within the range defined by the minimum and maximum values of the respective columns, ensuring the data remained consistent and meaningful.

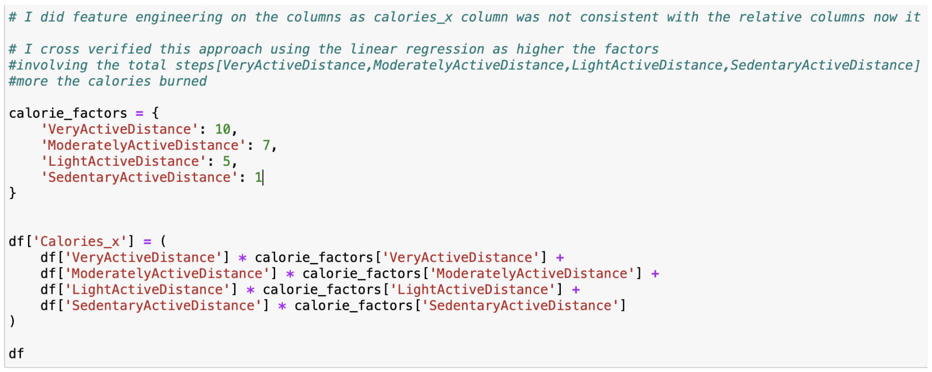


```python
# This function is to replace the duplicate integer values

# Here the only difference is I have used randint to generate the random integer

import numpy as np

def replace_duplicates_with_random_ids_integer(df, column_name, min_value, max_value):

    duplicates = df[df.duplicated(subset=[column_name], keep=False)]


    random_ids = np.random.randint(min_value, max_value, size=len(duplicates))


    df.loc[duplicates.index, column_name] = random_ids
    return df
```

Fig. 6. Function to impute integer values

4.6 Inconsistencies in Calories column

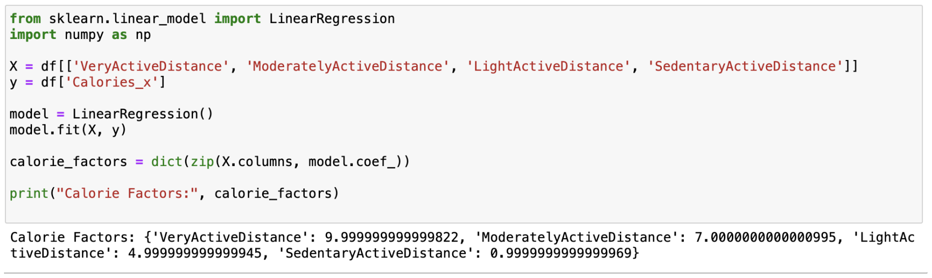
To address inconsistencies in the $Caloriesx$ column and align it with related activity metrics, I performed feature engineering to recalculate the values based on a weighted contribution from various activity distances. The original calorie data did not reflect the expected patterns of energy expenditure corresponding to different levels of physical activity. To correct this, I assigned specific weight factors to each activity type: $high-intensityactivities$, such as $VeryActiveDistance$, were given the highest weight, while lower-intensity activities like $SedentaryActiveDistance$ were assigned the lowest. This weighting system accounted for the varying impact of each activity on calorie burning. Using these weight factors, I recalculated the calorie values as a weighted sum of the distances covered for each activity type.



```
# I did feature engineering on the columns as calories_x column was not consistent with the relative columns now it

# I cross verified this approach using the linear regression as higher the factors
#involving the total steps[VeryActiveDistance,ModeratelyActiveDistance,LightActiveDistance,SedentaryActiveDistance]
#more the calories burned

calorie_factors = {
    'VeryActiveDistance': 10,
    'ModeratelyActiveDistance': 7,
    'LightActiveDistance': 5,
    'SedentaryActiveDistance': 1
}


df['Calories_x'] = (
    df['VeryActiveDistance'] * calorie_factors['VeryActiveDistance'] +
    df['ModeratelyActiveDistance'] * calorie_factors['ModeratelyActiveDistance'] +
    df['LightActiveDistance'] * calorie_factors['LightActiveDistance'] +
    df['SedentaryActiveDistance'] * calorie_factors['SedentaryActiveDistance']
)

df
```

Fig. 7. function to make it consistent Source: Google

4.7 Cross Verification by Linear Regression



```
from sklearn.linear_model import LinearRegression
import numpy as np

X = df[['VeryActiveDistance', 'ModeratelyActiveDistance', 'LightActiveDistance', 'SedentaryActiveDistance']]
y = df['Calories_x']

model = LinearRegression()
model.fit(X, y)

calorie_factors = dict(zip(X.columns, model.coef_))

print("Calorie Factors:", calorie_factors)
```

```
Calorie Factors: {'VeryActiveDistance': 9.999999999999822, 'ModeratelyActiveDistance': 7.0000000000000995, 'LightAc
tiveDistance': 4.999999999999945, 'SedentaryActiveDistance': 0.9999999999999969}
```

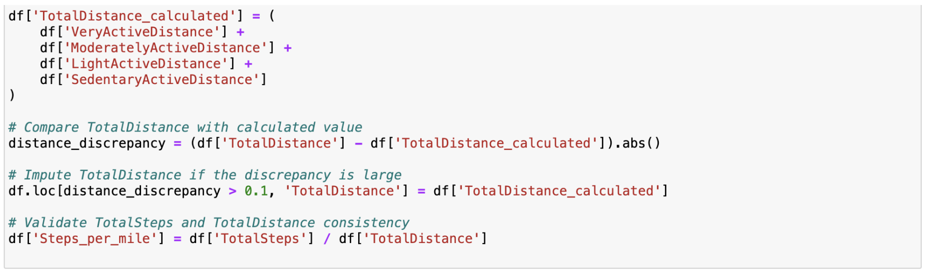
Fig. 8. Linear Regression

4.8 Cross Verifying the Total Distance

A new column, $TotalDistancecalculated$, is created by summing up the values from the $VeryActiveDistance$, $ModeratelyActiveDistance$, $LightActiveDistance$, and $SedentaryActiveDistance$ columns. This represents the total distance based on individual activity types.

Compare with Existing TotalDistance The code calculates the absolute difference between the existing $TotalDistance$ column and the newly computed $TotalDistancecalculated$. This difference is stored in the variable $distancediscrepancy$.

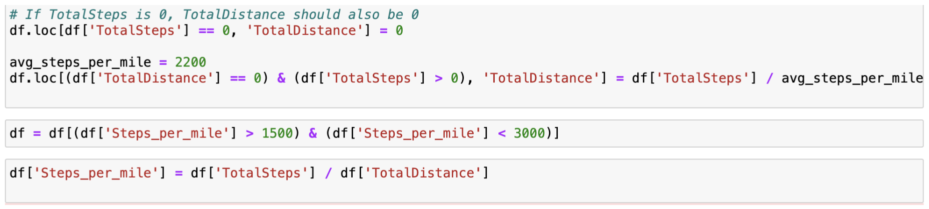
Validate Steps Per Mile The code then calculates a new column, $Stepspermile$, by dividing $TotalSteps$ by $TotalDistance$. This creates a metric that indicates the average number of steps taken per mile, which can be used to validate the data further or for additional analysis.



```
df['TotalDistance_calculated'] = (
    df['VeryActiveDistance'] +
    df['ModeratelyActiveDistance'] +
    df['LightActiveDistance'] +
    df['SedentaryActiveDistance']
)

# Compare TotalDistance with calculated value
distance_discrepancy = (df['TotalDistance'] - df['TotalDistance_calculated']).abs()

# Impute TotalDistance if the discrepancy is large
df.loc[distance_discrepancy > 0.1, 'TotalDistance'] = df['TotalDistance_calculated']

# Validate TotalSteps and TotalDistance consistency
df['Steps_per_mile'] = df['TotalSteps'] / df['TotalDistance']
```

Fig. 9. Validating Total Distance Source:ChatGPT

4.9 Steps per mile



```
# If TotalSteps is 0, TotalDistance should also be 0
df.loc[df['TotalSteps'] == 0, 'TotalDistance'] = 0

avg_steps_per_mile = 2200
df.loc[(df['TotalDistance'] == 0) & (df['TotalSteps'] > 0), 'TotalDistance'] = df['TotalSteps'] / avg_steps_per_mile
```

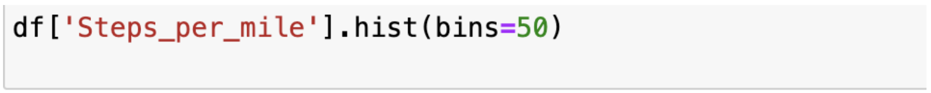
```
df = df[(df['Steps_per_mile'] > 1500) & (df['Steps_per_mile'] < 3000)]
```

```
df['Steps_per_mile'] = df['TotalSteps'] / df['TotalDistance']
```

Fig. 10. Steps per mile depiction

In this section of the code, I address specific inconsistencies in the TotalDistance and TotalSteps columns. First, I set TotalDistance to 0 whenever TotalSteps is 0, ensuring that the data reflects a logical relationship between the two columns—if there are no steps, there should be no distance traveled. Next, I handle cases where TotalDistance is 0 but TotalSteps is greater than 0. In such cases, I impute the TotalDistance by dividing TotalSteps by an average number of steps per mile (2200 steps per mile). This provides a reasonable estimate of the total distance when it's missing but there are steps recorded.

To further clean the data, I filter out any rows where Stepspermile falls outside a realistic range (less than 1500 or greater than 3000 steps per mile), which helps eliminate outliers or illogical data points. Finally, I recalculate Stepspermile by dividing TotalSteps by TotalDistance for all rows, and then visualize the distribution of Stepspermile using a histogram with 50 bins. This provides insights into the general pattern of steps per mile across the dataset and helps to validate the consistency and quality of the data.To understand the distribution of Stepspermile, a histogram is generated with 50 bins. The histogram provides a visual representation of the spread of Stepspermile values, helping to validate that the data is within the expected range and uncovering any patterns or anomalies that may still exist.



```
df['Steps_per_mile'].hist(bins=50)
```

```
<AxesSubplot:>
```

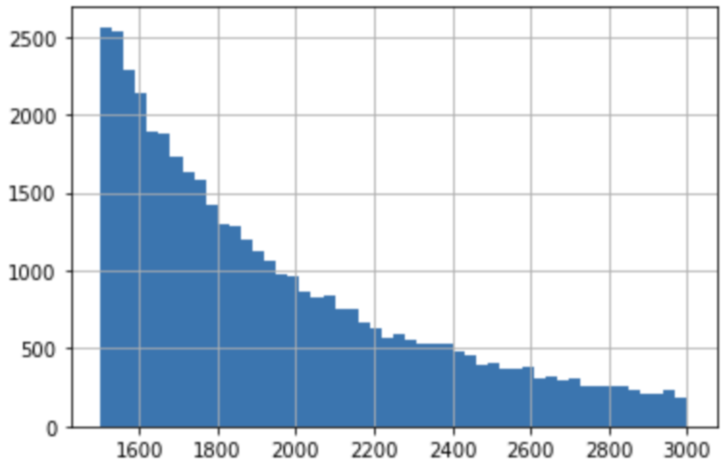
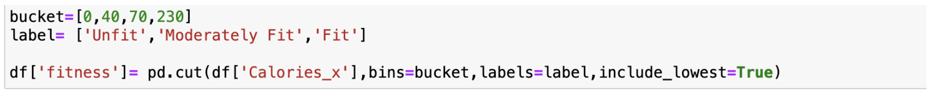


Fig. 11. Histogram of distribution

4.10 Bucketing the values for target column



```
bucket=[0,40,70,230]
label= ['Unfit','Moderately Fit','Fit']

df['fitness']= pd.cut(df['Calories_x'],bins=bucket,labels=label,include_lowest=True)
```

Fig. 12. Bucketing into UnFit, Moderately Fit and Fit

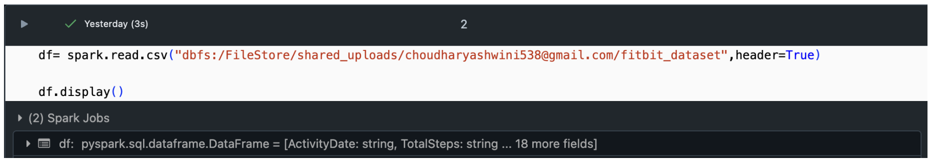
5 Feature Engineering in Databricks

5.1 Reading the Dataset

In this part of the code, I load the Fitbit dataset into a Spark DataFrame using the spark.read.csv function. The dataset is read from a specified path in Databricks

$$(dbfs:/FileStore/shared_uploads/choudharyashwini538@gmail.com/fitbitdataset)$$

, and the header=True argument ensures that the first row of the CSV file is treated as the column names.

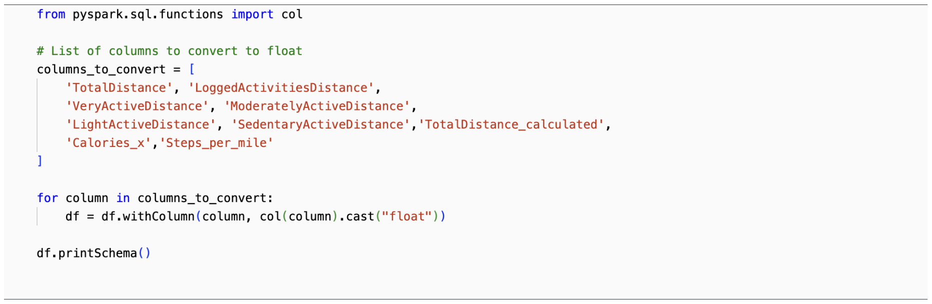


```
df= spark.read.csv("dbfs:/FileStore/shared_uploads/choudharyashwini538@gmail.com/fitbit_dataset",header=True)

df.display()
```

```
(2) Spark Jobs
df:  pyspark.sql.dataframe.DataFrame = [ActivityDate: string, TotalSteps: string ... 18 more fields]
```

Fig. 13. Reading the dataset

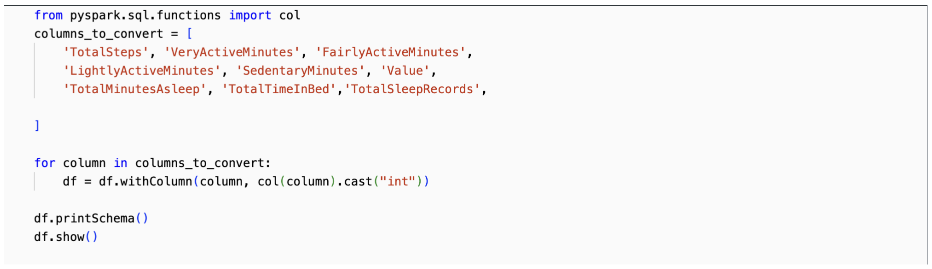
5.2 Converting columns to float type



```
from pyspark.sql.functions import col

# List of columns to convert to float
columns_to_convert = [
    'TotalDistance', 'LoggedActivitiesDistance',
    'VeryActiveDistance', 'ModeratelyActiveDistance',
    'LightActiveDistance', 'SedentaryActiveDistance','TotalDistance_calculated',
    'Calories_x','Steps_per_mile'
]

for column in columns_to_convert:
    df = df.withColumn(column, col(column).cast("float"))

df.printSchema()
```

Fig. 14. Columns converted to float for ANOVA

5.3 Converting columns to Integer type

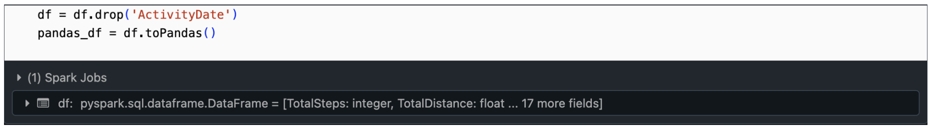


```
from pyspark.sql.functions import col
columns_to_convert = [
    'TotalSteps', 'VeryActiveMinutes', 'FairlyActiveMinutes',
    'LightlyActiveMinutes', 'SedentaryMinutes', 'Value',
    'TotalMinutesAsleep', 'TotalTimeInBed','TotalSleepRecords',

]

for column in columns_to_convert:
    df = df.withColumn(column, col(column).cast("int"))

df.printSchema()
df.show()
```

Fig. 15. Columns converted to Integer for ANOVA

5.4 Converting Dataframe to Pandas Dataframe

The purpose of loading the dataset into a Spark DataFrame was to facilitate the application of the ANOVA method to analyze the relationship between features and the target variable. Since Spark does not natively support ANOVA, this initial step allowed for detailed analysis and identification of correlations in the data. This exploration was particularly useful for deciding which columns to retain and which to drop, especially when their relevance to the target variable was uncertain. While Spark was ultimately used for preprocessing the data in later stages, this intermediate analysis provided critical insights to refine the dataset effectively before proceeding with Spark's distributed processing capabilities.



```
df = df.drop('ActivityDate')
pandas_df = df.toPandas()
```

```
(1) Spark Jobs
df:  pyspark.sql.dataframe.DataFrame = [TotalSteps: integer, TotalDistance: float ... 17 more fields]
```

Fig. 16. Dataframe to Pandas Dataframe

5.5 Applying ANOVA

ANOVA (Analysis of Variance) is a statistical technique used to determine if there are significant differences between the means of three or more groups. In the context of your analysis, you are using ANOVA to evaluate whether various numerical features (like Caloriesx, Stepspermile, TotalDistance, etc.) have a significant impact on the target variable fitness, which is categorical (e.g., "fit", "moderately fit", "sedentary").

Key Points for Results: Null Hypothesis (H0): There is no significant difference between the means of the feature across the fitness groups.

Alternative Hypothesis (H1): At least one group mean for the feature is different from the others.

F-Statistic: The ratio of between-group variance (variability among the different fitness groups) to within-group variance (variability within each fitness group).

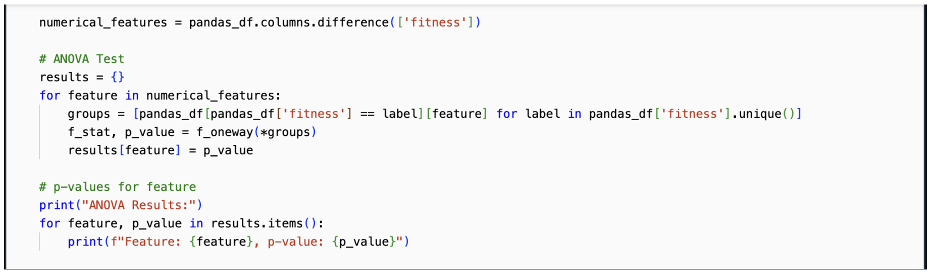
p-Value: The p-value tests the null hypothesis. A p-value lower than 0.05 suggests that the feature significantly influences the target variable (fitness), meaning we can reject the null hypothesis.

Interpretation of Results: Based on the p-values you've obtained from ANOVA for various features:

Highly Significant Features (p-value less than 0.05): Caloriesx, FairlyActiveMinutes, LightActiveDistance, ModeratelyActiveDistance, Stepspermile, TotalDistance, TotalSteps, VeryActiveDistance, VeryActiveMinutes: These features have p-values close to or equal to 0.0, indicating that they have a significant relationship with the fitness categories. We can confidently reject the null hypothesis for these features, meaning their mean values differ across fitness groups.

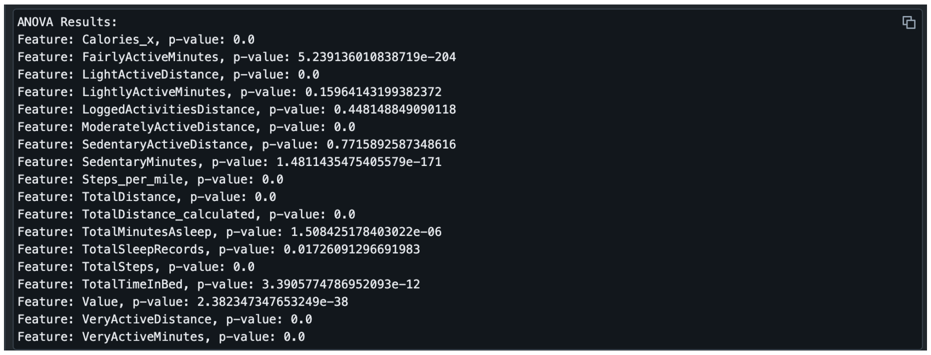
Moderately Significant Features: TotalSleepRecords: p-value = 0.01726 (significant but not as strongly related as other features).

Less Significant Features (p-value greater than 0.05): LightlyActiveMinutes, LoggedActivitiesDistance, SedentaryActiveDistance: These features have p-values greater than 0.05, suggesting no significant difference in their means across the fitness groups. Hence, these features may not provide useful information for predicting the fitness variable.



```
numerical_features = pandas_df.columns.difference(['fitness'])

# ANOVA Test
results = {}
for feature in numerical_features:
    groups = [pandas_df[pandas_df['fitness'] == label][feature] for label in pandas_df['fitness'].unique()]
    f_stat, p_value = f_oneway(*groups)
    results[feature] = p_value

# p-values for feature
print("ANOVA Results:")
for feature, p_value in results.items():
    print(f"Feature: {feature}, p-value: {p_value}")
```

Fig. 17. Applying ANOVA

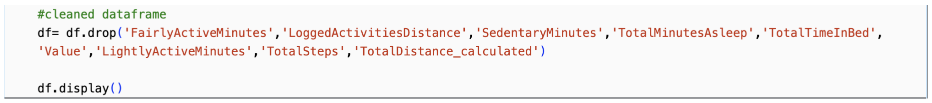


```
ANOVA Results:
Feature: Calories_x, p-value: 0.0
Feature: FairlyActiveMinutes, p-value: 5.239136010838719e-204
Feature: LightActiveDistance, p-value: 0.0
Feature: LightlyActiveMinutes, p-value: 0.15964143199382372
Feature: LoggedActivitiesDistance, p-value: 0.448148849090118
Feature: ModeratelyActiveDistance, p-value: 0.0
Feature: SedentaryActiveDistance, p-value: 0.7715892587348616
Feature: SedentaryMinutes, p-value: 1.4811435475405579e-171
Feature: Steps_per_mile, p-value: 0.0
Feature: TotalDistance, p-value: 0.0
Feature: TotalDistance_calculated, p-value: 0.0
Feature: TotalMinutesAsleep, p-value: 1.508425178403022e-06
Feature: TotalSleepRecords, p-value: 0.01726091296691983
Feature: TotalSteps, p-value: 0.0
Feature: TotalTimeInBed, p-value: 3.3905774786952093e-12
Feature: Value, p-value: 2.382347347653249e-38
Feature: VeryActiveDistance, p-value: 0.0
Feature: VeryActiveMinutes, p-value: 0.0
```

Fig. 18. Results for ANOVA

5.6 Dropping Columns

After analyzing the ANOVA results, several features were identified as either not significantly correlated with the target variable fitness or redundant in their contribution to the dataset. Features such as FairlyActiveMinutes, LoggedActivitiesDistance, SedentaryMinutes, TotalMinutesAsleep, TotalTimeInBed, Value, and LightlyActiveMinutes had high p-values or were less impactful compared to other strongly correlated features. Additionally, TotalSteps and TotalDistance-calculated were considered redundant, as their information was already captured through other variables like Stepspermile and TotalDistance.

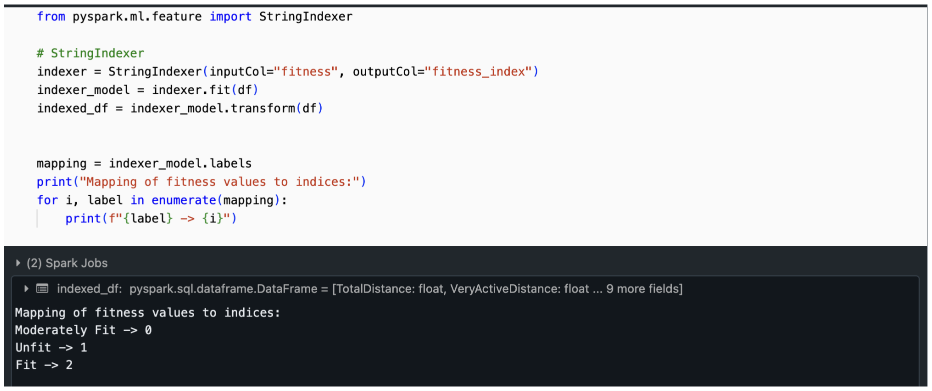


```
#cleaned dataframe
df= df.drop('FairlyActiveMinutes','LoggedActivitiesDistance','SedentaryMinutes','TotalMinutesAsleep','TotalTimeInBed',
'Value','LightlyActiveMinutes','TotalSteps','TotalDistance_calculated')

df.display()
```

Fig. 19. Dropping less significant columns

5.7 Applying String Indexer

Here I mapped the values using String Indexer for the target column.



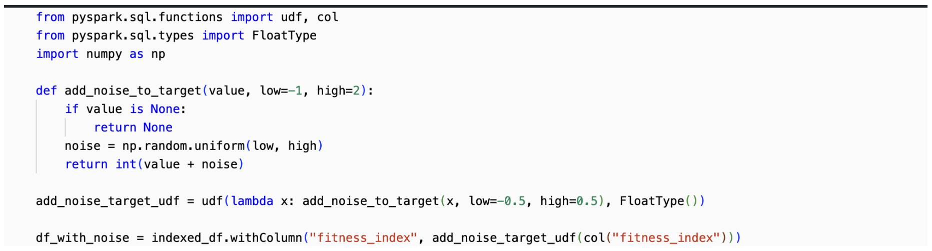
```
from pyspark.ml.feature import StringIndexer

# StringIndexer
indexer = StringIndexer(inputCol="fitness", outputCol="fitness_index")
indexer_model = indexer.fit(df)
indexed_df = indexer_model.transform(df)


mapping = indexer_model.labels
print("Mapping of fitness values to indices:")
for i, label in enumerate(mapping):
    print(f"{label} -> {i}")
```

```
(2) Spark Jobs
indexed_df: pyspark.sql.dataframe.DataFrame = [TotalDistance: float, VeryActiveDistance: float ... 9 more fields]
Mapping of fitness values to indices:
Moderately Fit -> 0
Unfit -> 1
Fit -> 2
```

Fig. 20. Mapped values

5.8 Adding Noise to the Target Column

I added noise to my dataset as it was highly linear in nature and due to which the model performance to too high and the results were not satisfactory.

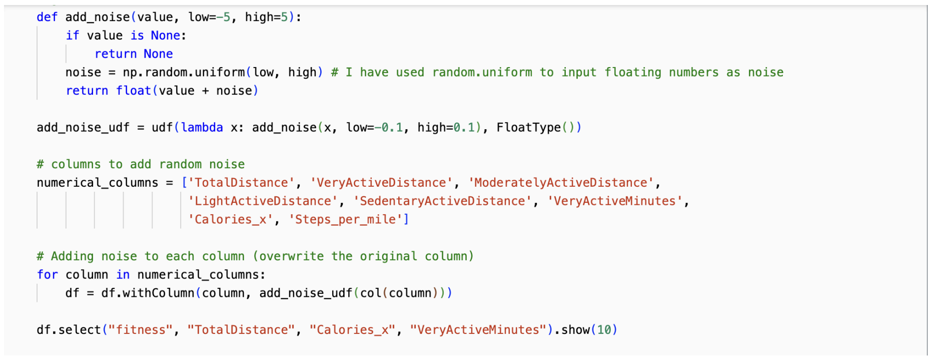


```
from pyspark.sql.functions import udf, col
from pyspark.sql.types import FloatType
import numpy as np

def add_noise_to_target(value, low=-1, high=2):
    if value is None:
        return None
    noise = np.random.uniform(low, high)
    return int(value + noise)

add_noise_target_udf = udf(lambda x: add_noise_to_target(x, low=-0.5, high=0.5), FloatType())

df_with_noise = indexed_df.withColumn("fitness_index", add_noise_target_udf(col("fitness_index")))
```

Fig. 21. Noisy Dataframe

5.9 Adding Noise to the Features

To address the issue of high linearity among certain numerical features in my dataset. By adding small random variations to these columns, using a custom function that generated uniform noise within a specified range, I was able to reduce perfect linear correlations while maintaining the overall data structure. This preprocessing step ensured that the model could better handle variability and avoid overly relying on exact feature alignments, ultimately improving its robustness and predictive capabilities.

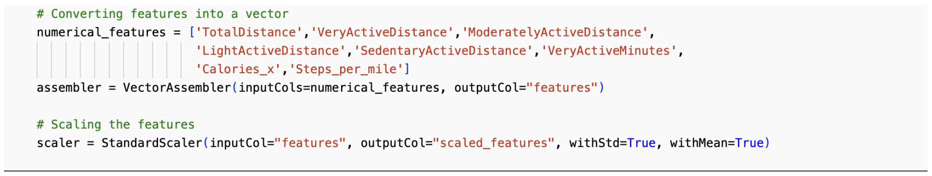


```python
def add_noise(value, low=-5, high=5):
    if value is None:
        return None
    noise = np.random.uniform(low, high) # I have used random.uniform to input floating numbers as noise
    return float(value + noise)

add_noise_udf = udf(lambda x: add_noise(x, low=-0.1, high=0.1), FloatType())

# columns to add random noise
numerical_columns = ['TotalDistance', 'VeryActiveDistance', 'ModeratelyActiveDistance',
                     'LightActiveDistance', 'SedentaryActiveDistance', 'VeryActiveMinutes',
                     'Calories_x', 'Steps_per_mile']

# Adding noise to each column (overwrite the original column)
for column in numerical_columns:
    df = df.withColumn(column, add_noise_udf(col(column)))

df.select("fitness", "TotalDistance", "Calories_x", "VeryActiveMinutes").show(10)
```

Fig. 22. Noisy Dataframe

5.10 Scaling the features

I used Standard Scaler to scale my features so that it is easy for algorithm to process the data.

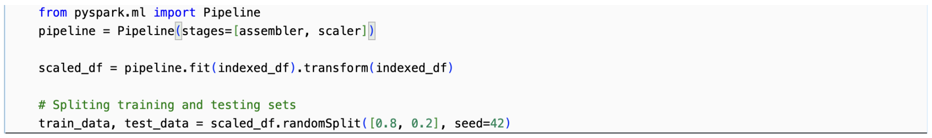


```python
# Converting features into a vector
numerical_features = ['TotalDistance','VeryActiveDistance','ModeratelyActiveDistance',
                      'LightActiveDistance','SedentaryActiveDistance','VeryActiveMinutes',
                      'Calories_x','Steps_per_mile']
assembler = VectorAssembler(inputCols=numerical_features, outputCol="features")

# Scaling the features
scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True, withMean=True)
```

Fig. 23. Applying Standard Scaler

5.11 Splitting into Train and Test

I then split this scaled dataset into training and testing sets, allocating 80 percent of the data for training and 20 percent for testing.

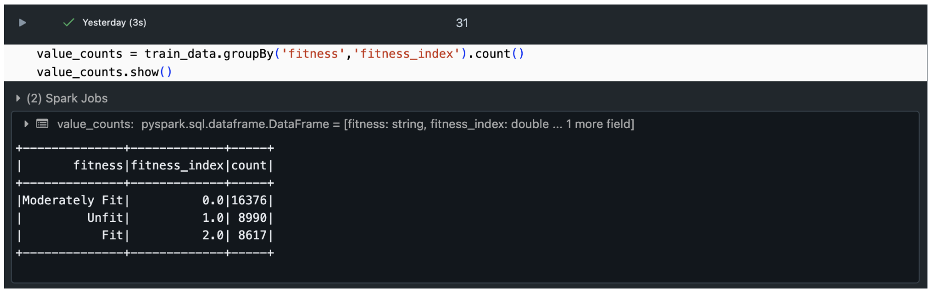


```
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[assembler, scaler])

scaled_df = pipeline.fit(indexed_df).transform(indexed_df)

# Spliting training and testing sets
train_data, test_data = scaled_df.randomSplit([0.8, 0.2], seed=42)
```

Fig. 24. Train and Test Data

5.12 Check for Balance in the Dataset

My training dataset it not balanced its skewed so I have to balance it otherwise my model will not learn the other two classes.



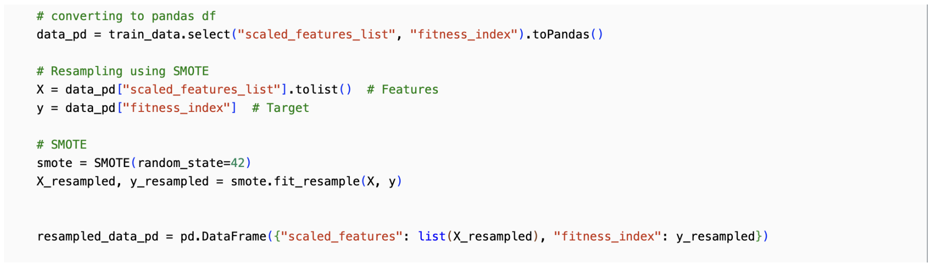
```
value_counts = train_data.groupBy('fitness','fitness_index').count()
value_counts.show()
```

```
value_counts: pyspark.sql.dataframe.DataFrame = [fitness: string, fitness_index: double ... 1 more field]
+-------------+-------------+-----+
|      fitness|fitness_index|count|
+-------------+-------------+-----+
|Moderately Fit|          0.0|16376|
|        Unfit|          1.0| 8990|
|          Fit|          2.0| 8617|
+-------------+-------------+-----+
```

Fig. 25. Dataset Not Balanced

5.13 Apply SMOTE to balance the Dataset

Here I used SMOTE (Synthetic Minority Oversampling Technique) to balance the dataset, it fill's up the minority classes with synthetically generated sample so that our model learn about all classes equally.

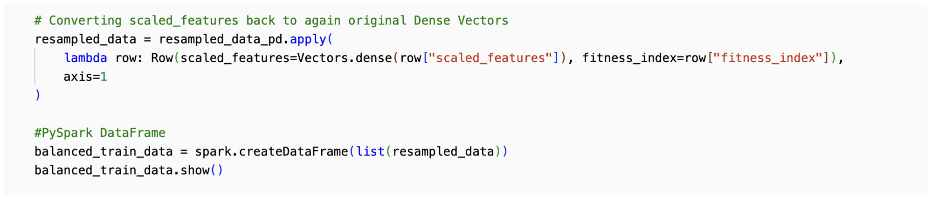


```
# converting to pandas df
data_pd = train_data.select("scaled_features_list", "fitness_index").toPandas()

# Resampling using SMOTE
X = data_pd["scaled_features_list"].tolist()  # Features
y = data_pd["fitness_index"]  # Target

# SMOTE
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)


resampled_data_pd = pd.DataFrame({"scaled_features": list(X_resampled), "fitness_index": y_resampled})
```

Fig. 26. SMOTE

5.14 Converting Dataframe back Pyspark Dataframe

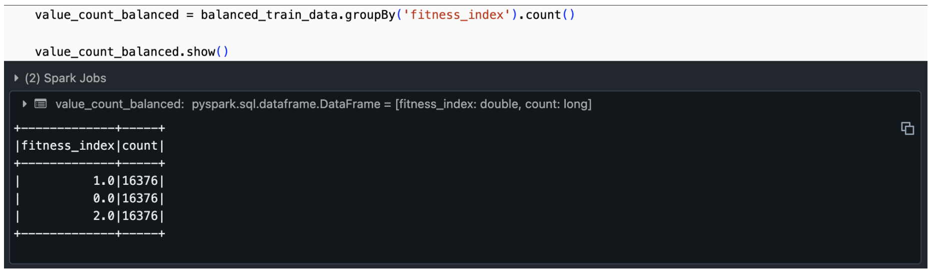
Since there was no library in the pyspark to directly apply SMOTE so, I converted my dataframe to pandas dataframe and after applying SMOTE I converted my dataset back to the dense vectored dataset.



```
# Converting scaled_features back to again original Dense Vectors
resampled_data = resampled_data_pd.apply(
    lambda row: Row(scaled_features=Vectors.dense(row["scaled_features"]), fitness_index=row["fitness_index"]),
    axis=1
)

#PySpark DataFrame
balanced_train_data = spark.createDataFrame(list(resampled_data))
balanced_train_data.show()
```

Fig. 27. Converted Pyspark Dataset

5.15 Dataset is balanced



```
value_count_balanced = balanced_train_data.groupBy('fitness_index').count()

value_count_balanced.show()
```

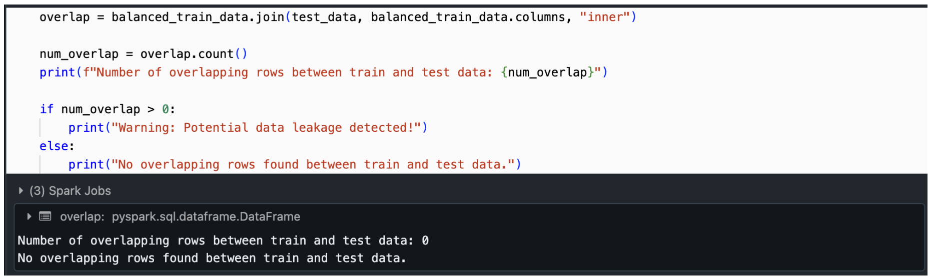
```
(2) Spark Jobs
value_count_balanced:  pyspark.sql.dataframe.DataFrame = [fitness_index: double, count: long]
+-------------+-----+
|fitness_index|count|
+-------------+-----+
|          1.0|16376|
|          0.0|16376|
|          2.0|16376|
+-------------+-----+
```

Fig. 28. Balanced Dataframe

5.16 Check Data Leakage

To ensure a robust and unbiased evaluation of the machine learning model, it was essential to verify that the training and testing datasets were completely distinct. Overlapping data points, also known as data leakage, can compromise the model's performance evaluation by allowing it to "cheat," learning from examples in the training set that also exist in the testing set. This results in overly optimistic performance metrics, which do not reflect the model's true generalization capability.

To address this, I performed a rigorous check for overlap by conducting an inner join between the training and testing datasets using all columns as the matching criteria. This ensured that any duplicate or identical records appearing in both sets would be identified. The total count of overlapping rows was computed and displayed. If any overlap was detected, it triggered a warning to signal potential data leakage and the need for corrective measures, such as revisiting the data-splitting process. If no overlapping rows were found, it confirmed that the data splitting process was successful, and the model's evaluation would be reliable and valid.



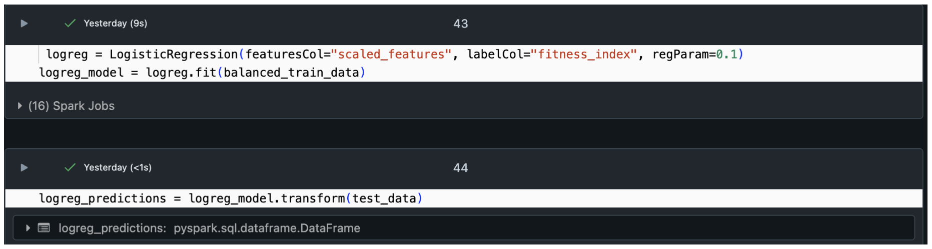
```
overlap = balanced_train_data.join(test_data, balanced_train_data.columns, "inner")

num_overlap = overlap.count()
print(f"Number of overlapping rows between train and test data: {num_overlap}")

if num_overlap > 0:
    print("Warning: Potential data leakage detected!")
else:
    print("No overlapping rows found between train and test data.")
```

```
(3) Spark Jobs
overlap: pyspark.sql.dataframe.DataFrame
Number of overlapping rows between train and test data: 0
No overlapping rows found between train and test data.
```

Fig. 29. Balanced Dataframe

6 Applying Model - Logistic Regression

The model was trained on a balanced training dataset to ensure that all fitness categories were equally represented, which prevents bias in predictions. The featuresCol parameter specified the scaled features to be used for training, while the labelCol identified the target variable, fitnessindex. Additionally, a regularization parameter (regParam=0.1) was applied to prevent overfitting by penalizing overly complex models.



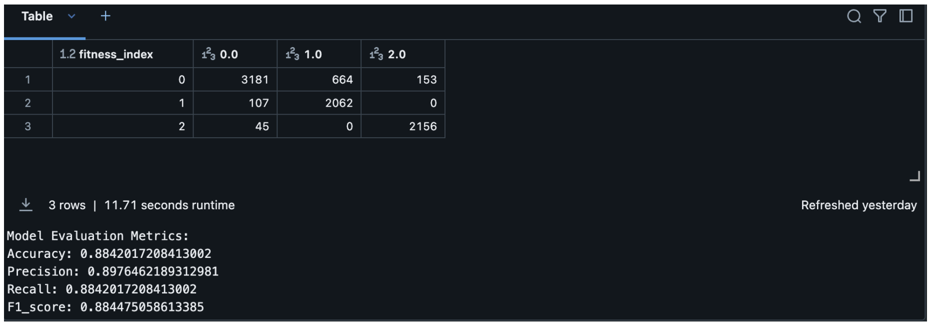
```
Yesterday (9s)                                   43
logreg = LogisticRegression(featuresCol="scaled_features", labelCol="fitness_index", regParam=0.1)
logreg_model = logreg.fit(balanced_train_data)
(16) Spark Jobs
```

```
Yesterday (<1s)                                  44
logreg_predictions = logreg_model.transform(test_data)
logreg_predictions: pyspark.sql.dataframe.DataFrame
```

Fig. 30. Logistic Regression

6.1 Evaluation - Logistic Regression

To evaluate the performance of the logistic regression model, I utilized several metrics to capture its overall effectiveness and classification accuracy. Accuracy served as a measure of the proportion of correct predictions out of the total predictions, providing a general sense of the model's reliability.



	fitness_index	0.0	1.0	2.0
1	0	3181	664	153
2	1	107	2062	0
3	2	45	0	2156

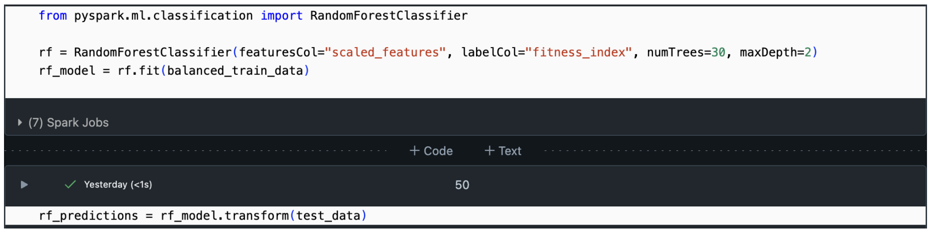
3 rows | 11.71 seconds runtime

Refreshed yesterday

```
Model Evaluation Metrics:
Accuracy: 0.8842017208413002
Precision: 0.8976462189312981
Recall: 0.8842017208413002
F1_score: 0.884475058613385
```

Fig. 31. Results- Logistic Regression

6.2 Applying Model - Random Forest



```
from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(featuresCol="scaled_features", labelCol="fitness_index", numTrees=30, maxDepth=2)
rf_model = rf.fit(balanced_train_data)
```

(7) Spark Jobs

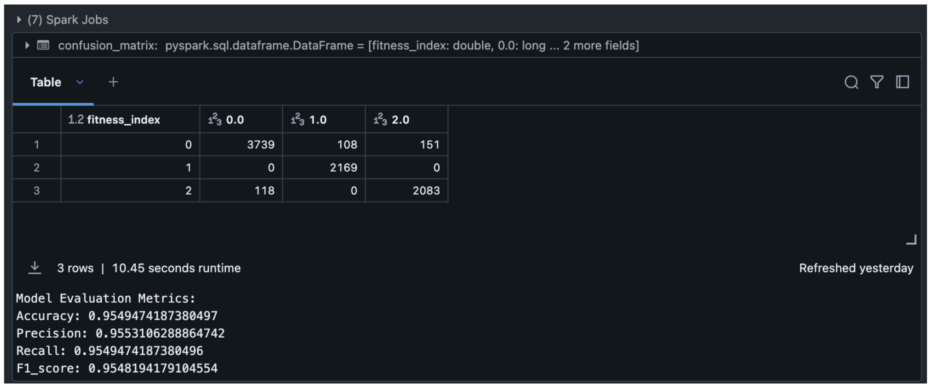
Yesterday (<1s) 50

```
rf_predictions = rf_model.transform(test_data)
```

Fig. 32. Random Forest

6.3 Evaluation - Random Forest

The performance of this Random Forest model will be evaluated similarly to the logistic regression model, using metrics such as accuracy, precision, recall, and F1-score. These metrics will measure how effectively the ensemble model distinguishes between fitness categories while balancing false positives and false negatives.Additionally, feature importance scores from the Random Forest can provide valuable insights.



(7) Spark Jobs

confusion_matrix: pyspark.sql.dataframe.DataFrame = [fitness_index: double, 0.0: long ... 2 more fields]

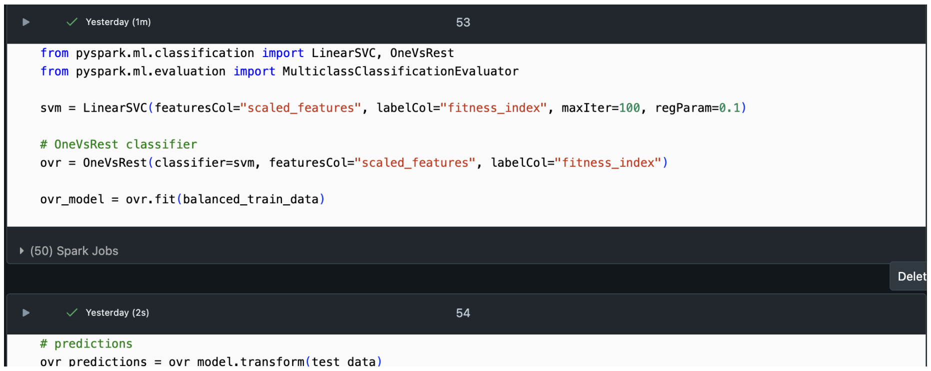
	fitness_index	0.0	1.0	2.0
1	0	3739	108	151
2	1	0	2169	0
3	2	118	0	2083

3 rows | 10.45 seconds runtime   Refreshed yesterday

```
Model Evaluation Metrics:
Accuracy: 0.9549474187380497
Precision: 0.9553106288864742
Recall: 0.9549474187380496
F1_score: 0.9548194179104554
```

Fig. 33. Evaluation - Random Forest

6.4 Applying Model - SVM (One vs Rest)



Yesterday (1m) 53

```
from pyspark.ml.classification import LinearSVC, OneVsRest
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

svm = LinearSVC(featuresCol="scaled_features", labelCol="fitness_index", maxIter=100, regParam=0.1)

# OneVsRest classifier
ovr = OneVsRest(classifier=svm, featuresCol="scaled_features", labelCol="fitness_index")

ovr_model = ovr.fit(balanced_train_data)
```

(50) Spark Jobs

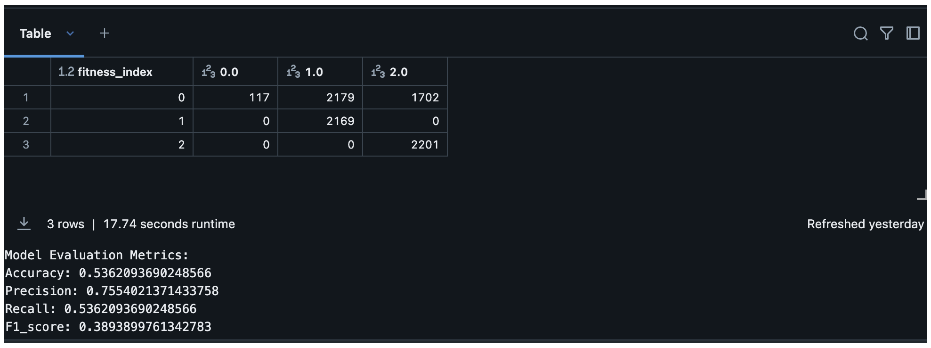
Yesterday (2s) 54

```
# predictions
ovr_predictions = ovr_model.transform(test_data)
```

Fig. 34. SVM

6.5 Evaluation - SVM (One vs Rest)

To explore the fitness classification further, a Support Vector Machine (SVM) model was employed using the One-vs-Rest (OvR) strategy. The Linear Support Vector Classifier (LinearSVC) was configured with scaled features as inputs, a maximum iteration limit of 100, and a regularization parameter (regParam) of 0.1 to control the trade-off between achieving a low error on the training data and minimizing model complexity. SVM is known for its ability to create robust decision boundaries by finding the hyperplane that maximizes the margin between classes, making it effective for high-dimensional data.



Table

	1.2 fitness_index	1²3 0.0	1²3 1.0	1²3 2.0
1	0	117	2179	1702
2	1	0	2169	0
3	2	0	0	2201

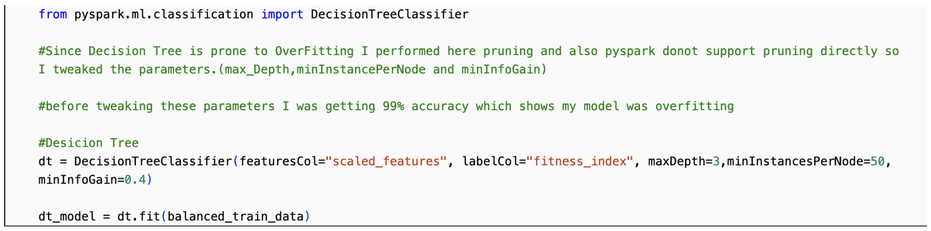
3 rows | 17.74 seconds runtime

Refreshed yesterday

```
Model Evaluation Metrics:
Accuracy: 0.5362093690248566
Precision: 0.7554021371433758
Recall: 0.5362093690248566
F1_score: 0.3893899761342783
```

Fig. 35. Evaluation - SVM

6.6 Applying Algorithm - Decision Trees

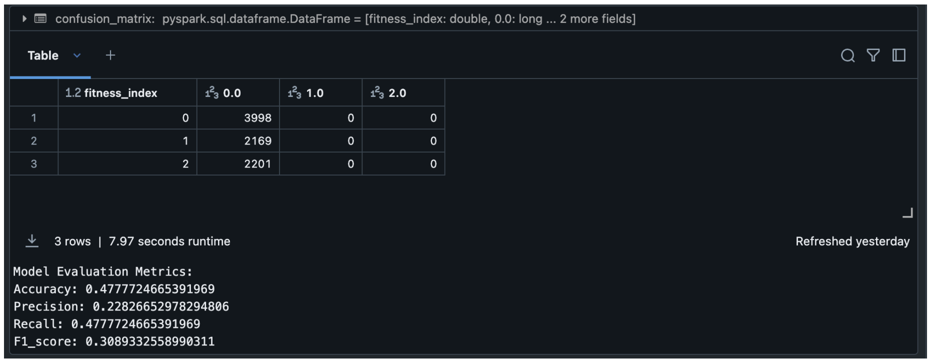


```
from pyspark.ml.classification import DecisionTreeClassifier

#Since Decision Tree is prone to OverFitting I performed here pruning and also pyspark donot support pruning directly so
I tweaked the parameters.(max_Depth,minInstancePerNode and minInfoGain)

#before tweaking these parameters I was getting 99% accuracy which shows my model was overfitting

#Desicion Tree
dt = DecisionTreeClassifier(featuresCol="scaled_features", labelCol="fitness_index", maxDepth=3,minInstancesPerNode=50,
minInfoGain=0.4)

dt_model = dt.fit(balanced_train_data)
```

Fig. 36. Decision Trees

6.7 Evaluation - Decision Trees

To address the risk of overfitting commonly associated with Decision Tree Classifiers, I implemented a tailored approach to pruning by tweaking key hyperparameters, as PySpark does not provide direct pruning functionality. Initially, the model achieved an excessively high accuracy of 99 percent, a clear indication of overfitting to the training data. To mitigate this, I constrained the tree's complexity by setting the maximum depth to 3, which limited how deep the tree could grow. I also adjusted the minimum instances per node to 50, ensuring that each leaf node contained a sufficient number of samples to avoid splitting on noise or small variations in the data. I set a minimum information gain threshold of 0.4, requiring significant improvement in predictive power before further splits could be made.These adjustments allowed the Decision Tree model to generalize better, striking a balance between predictive performance and overfitting. By training this adjusted model on the balanced dataset, the tree became more robust and fair in its classification of the fitness categories, reducing the likelihood of overfitting while retaining interpretability and accuracy.


confusion_matrix: pyspark.sql.dataframe.DataFrame = [fitness_index: double, 0.0: long ... 2 more fields]
Table

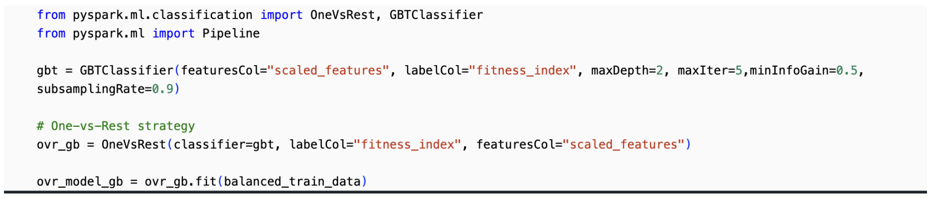
	fitness_index	0.0	1.0	2.0
1	0	3998	0	0
2	1	2169	0	0
3	2	2201	0	0

3 rows | 7.97 seconds runtime
Refreshed yesterday

```
Model Evaluation Metrics:
Accuracy: 0.4777724665391969
Precision: 0.22826652978294806
Recall: 0.4777724665391969
F1_score: 0.3089332558990311
```

Fig. 37. Evaluation - Decision Trees

6.8 Applying Algorithm - Gradient Boosting

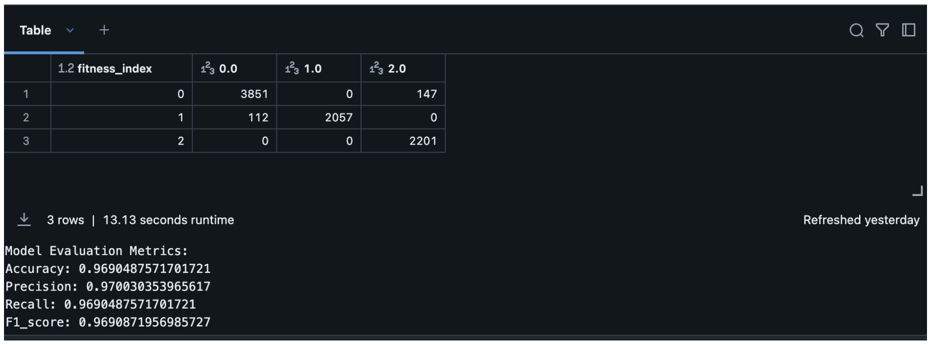


```
from pyspark.ml.classification import OneVsRest, GBTClassifier
from pyspark.ml import Pipeline

gbt = GBTClassifier(featuresCol="scaled_features", labelCol="fitness_index", maxDepth=2, maxIter=5,minInfoGain=0.5,
subsamplingRate=0.9)

# One-vs-Rest strategy
ovr_gb = OneVsRest(classifier=gbt, labelCol="fitness_index", featuresCol="scaled_features")

ovr_model_gb = ovr_gb.fit(balanced_train_data)
```

Fig. 38. Gradient Boosting One VS Rest

6.9 Evaluation - Gradient Boosting

To enhance the classification performance and handle multi-class classification effectively, I employed the One-vs-Rest (OvR) strategy in conjunction with a Gradient-Boosted Tree (GBT) classifier. The GBT classifier is a powerful ensemble learning method known for its ability to reduce bias and variance, making it particularly suitable for complex datasets. In this implementation, the GBT classifier was configured with a maximum depth of 2, limiting the depth of each individual tree to prevent overfitting, and a maximum of 5 iterations to control the number of boosting rounds. Additionally, I set the minimum information gain to 0.5, ensuring that splits were only made when they added significant predictive value, and used a subsampling rate of 0.9 to introduce randomness and improve generalization.The OvR strategy was used to extend the binary GBT classifier to a multi-class setting by training one classifier per class against all others. This approach allowed the model to handle multiple fitness categories effectively, leveraging the GBT's strengths for accurate predictions while maintaining computational efficiency.



	fitness_index	0.0	1.0	2.0
1	0	3851	0	147
2	1	112	2057	0
3	2	0	0	2201

```
Model Evaluation Metrics:
Accuracy: 0.9690487571701721
Precision: 0.970030353965617
Recall: 0.9690487571701721
F1_score: 0.9690871956985727
```

Fig. 39. Gradient Boosting One VS Rest

7 Cross Validation

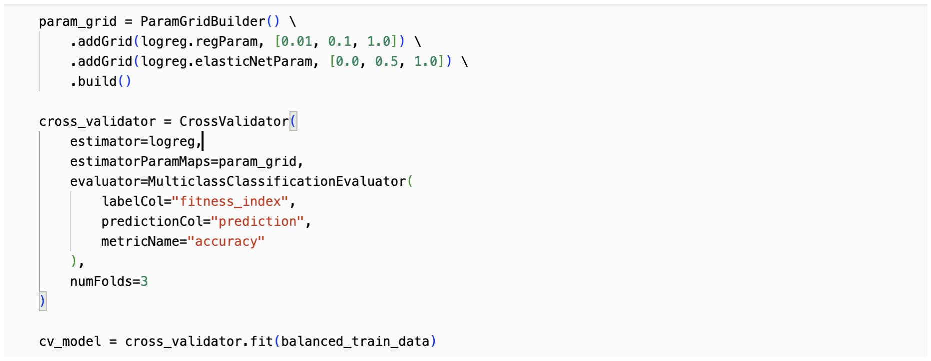
Cross-validation is a model evaluation technique used to assess the performance of a machine learning model. It involves splitting the dataset into multiple subsets or "folds." The model is then trained on a subset of the data (the training set) and tested on the remaining data (the validation set). This process is repeated several times, each time using a different fold as the validation set and the remaining folds as the training set. The results from each iteration are then averaged to provide a more reliable estimate of the model's performance.

The main purpose of cross-validation is to ensure that the model generalizes well to unseen data, reducing the risk of overfitting (where the model performs well on training data but poorly on new, unseen data). By using multiple train-test splits, cross-validation helps to obtain a more accurate and robust evaluation of the model's ability to make predictions on various data points. This technique is widely used in machine learning to optimize model selection and hyperparameters.

7.1 Logistic Regression with Best Parameters

To improve the performance and reliability of the logistic regression model, I implemented a cross-validation technique using PySpark's CrossValidator. Cross-validation helps evaluate the model's ability to generalize to unseen data by dividing the training data into multiple subsets (folds). For this task, I specified a 3-fold cross-validation, where the data is divided into three parts. The model is trained on two folds and validated on the third, rotating through all folds, and the results are averaged to ensure robustness.

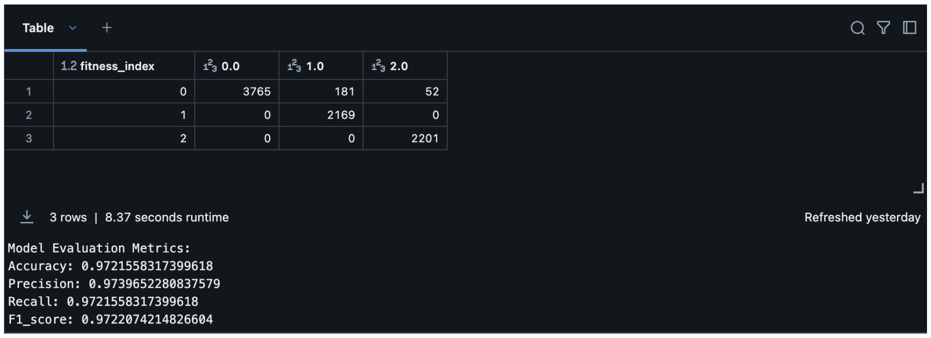
I constructed a parameter grid using ParamGridBuilder to fine-tune the model's hyperparameters. The grid includes different values for the regularization parameter (regParam), which controls overfitting by penalizing large coefficients, and the elasticNetParam, which balances between L1 and L2 regularization. The values for these parameters (e.g., regParam: [0.01, 0.1, 1.0] and elasticNetParam: [0.0, 0.5, 1.0]) ensure a comprehensive exploration of the parameter space.



```
param_grid = ParamGridBuilder() \
    .addGrid(logreg.regParam, [0.01, 0.1, 1.0]) \
    .addGrid(logreg.elasticNetParam, [0.0, 0.5, 1.0]) \
    .build()

cross_validator = CrossValidator(
    estimator=logreg,
    estimatorParamMaps=param_grid,
    evaluator=MulticlassClassificationEvaluator(
        labelCol="fitness_index",
        predictionCol="prediction",
        metricName="accuracy"
    ),
    numFolds=3
)

cv_model = cross_validator.fit(balanced_train_data)
```

Fig. 40. Logistic Regression Cross Validation

7.2 Evaluation - Logistic Regression with Best Parameters



	1.2 fitness_index	0.0	1.0	2.0
1	0	3765	181	52
2	1	0	2169	0
3	2	0	0	2201

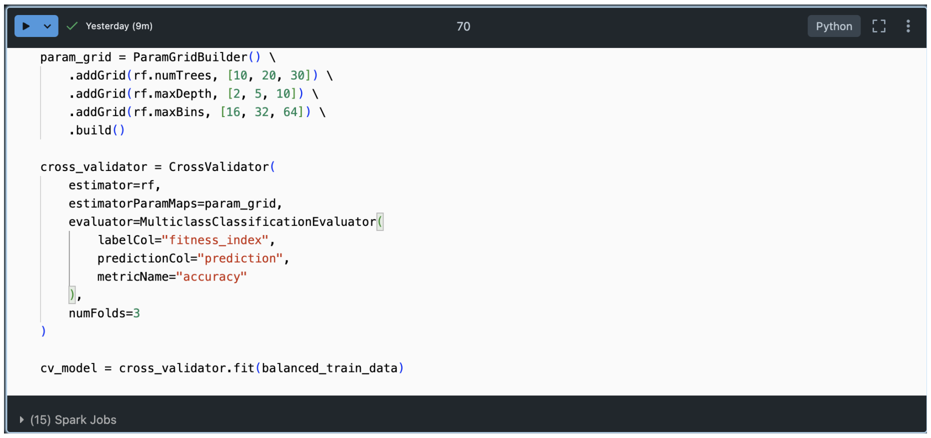
3 rows | 8.37 seconds runtime

```
Model Evaluation Metrics:
Accuracy: 0.9721558317399618
Precision: 0.9739652280837579
Recall: 0.9721558317399618
F1_score: 0.9722074214826604
```

Fig. 41. Evaluation - Logistic Regression

7.3 Random Forest with Best Parameters

To enhance the performance and generalization of the Random Forest model, I utilized a cross-validation approach in conjunction with hyperparameter tuning using PySpark's CrossValidator. The parameter grid was constructed with ParamGridBuilder, specifying ranges for critical hyperparameters of the Random Forest algorithm. These included the number of trees in the forest (numTrees), the maximum depth of the trees (maxDepth), and the maximum number of bins used for splitting data (maxBins). For each parameter, I explored multiple values: [10, 20, 30] for numTrees, [2, 5, 10] for maxDepth, and [16, 32, 64] for maxBins. This ensured a thorough search for the optimal combination of hyperparameters.

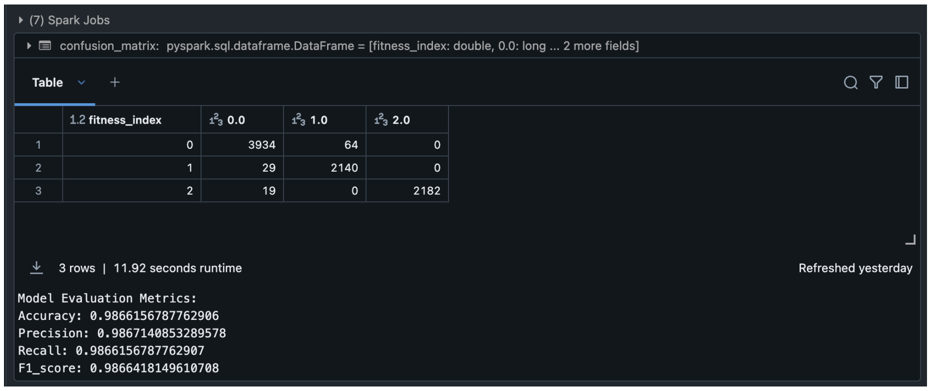


```
param_grid = ParamGridBuilder() \
    .addGrid(rf.numTrees, [10, 20, 30]) \
    .addGrid(rf.maxDepth, [2, 5, 10]) \
    .addGrid(rf.maxBins, [16, 32, 64]) \
    .build()

cross_validator = CrossValidator(
    estimator=rf,
    estimatorParamMaps=param_grid,
    evaluator=MulticlassClassificationEvaluator(
        labelCol="fitness_index",
        predictionCol="prediction",
        metricName="accuracy"
    ),
    numFolds=3
)

cv_model = cross_validator.fit(balanced_train_data)
```

```
(15) Spark Jobs
```

Fig. 42. Random Forest Cross Validation

7.4 Evaluation - Random Forest with Best Parameters



```
(7) Spark Jobs
confusion_matrix: pyspark.sql.dataframe.DataFrame = [fitness_index: double, 0.0: long ... 2 more fields]
```

	fitness_index	0.0	1.0	2.0
1	0	3934	64	0
2	1	29	2140	0
3	2	19	0	2182

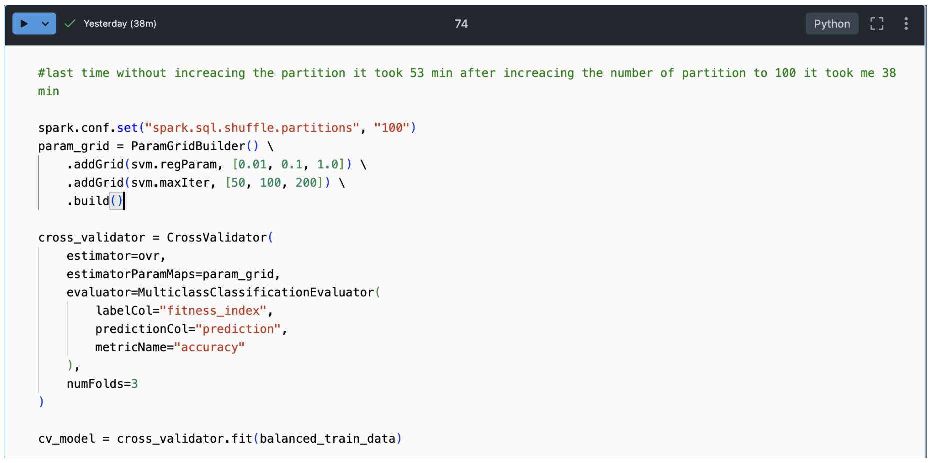
```
3 rows | 11.92 seconds runtime

Model Evaluation Metrics:
Accuracy: 0.9866156787762906
Precision: 0.9867140853289578
Recall: 0.9866156787762907
F1_score: 0.9866418149610708
```

Fig. 43. Evaluation - Random Forest Cross Validation

7.5 SVM with Best Parameters

To improve the efficiency and performance of the cross-validation process for the One-vs-Rest model using the Support Vector Machine (SVM) classifier, I increased the number of shuffle partitions in Spark from the default setting to 100. This adjustment significantly reduced the execution time, decreasing it from 53 minutes to 38 minutes. This optimization ensures a more balanced distribution of data across partitions, leading to better parallelism during the computation-intensive cross-validation.

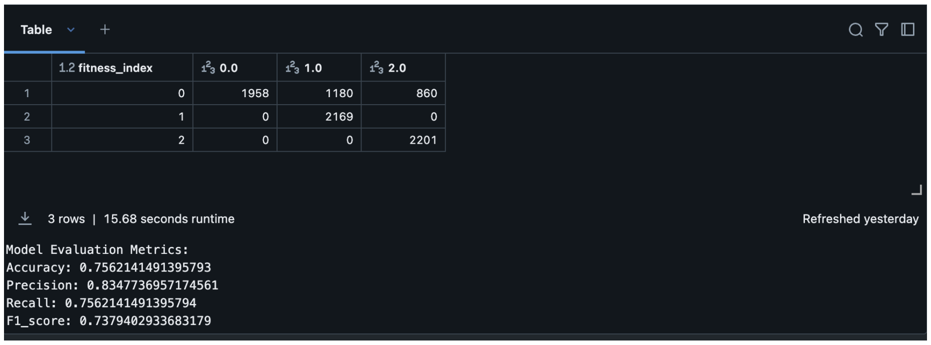
For hyperparameter tuning, a parameter grid was constructed using ParamGridBuilder. It included combinations of regularization parameter (regParam) values [0.01, 0.1, 1.0] and maximum iteration values [50, 100, 200] for the SVM classifier. These hyperparameters control the regularization strength and the convergence threshold of the algorithm, respectively, enabling a thorough search for the best configuration.



```
#last time without increacing the partition it took 53 min after increacing the number of partition to 100 it took me 38
min

spark.conf.set("spark.sql.shuffle.partitions", "100")
param_grid = ParamGridBuilder() \
    .addGrid(svm.regParam, [0.01, 0.1, 1.0]) \
    .addGrid(svm.maxIter, [50, 100, 200]) \
    .build()

cross_validator = CrossValidator(
    estimator=ovr,
    estimatorParamMaps=param_grid,
    evaluator=MulticlassClassificationEvaluator(
        labelCol="fitness_index",
        predictionCol="prediction",
        metricName="accuracy"
    ),
    numFolds=3
)

cv_model = cross_validator.fit(balanced_train_data)
```

Fig. 44. SVM

7.6 Evaluation - SVM with Best Parameters



	1.2 fitness_index	0.0	1.0	2.0
1	0	1958	1180	860
2	1	0	2169	0
3	2	0	0	2201

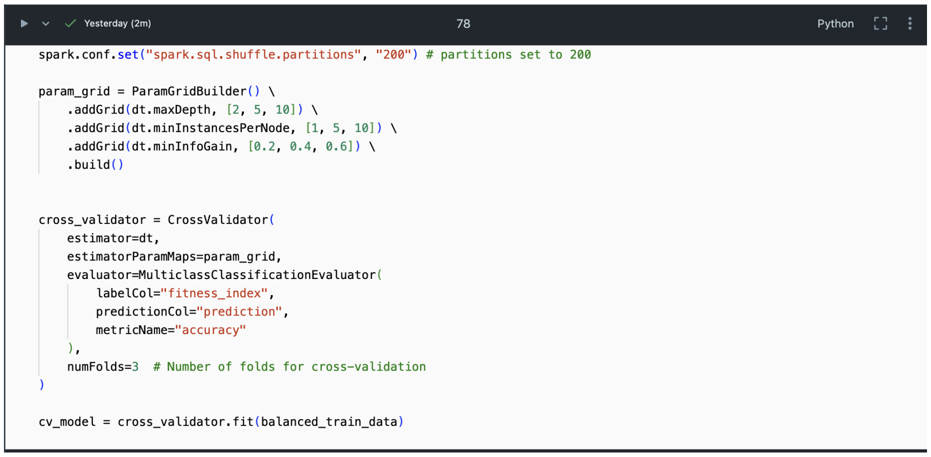
3 rows | 15.68 seconds runtime

Refreshed yesterday

```
Model Evaluation Metrics:
Accuracy: 0.7562141491395793
Precision: 0.8347736957174561
Recall: 0.7562141491395794
F1_score: 0.7379402933683179
```

Fig. 45. SVM

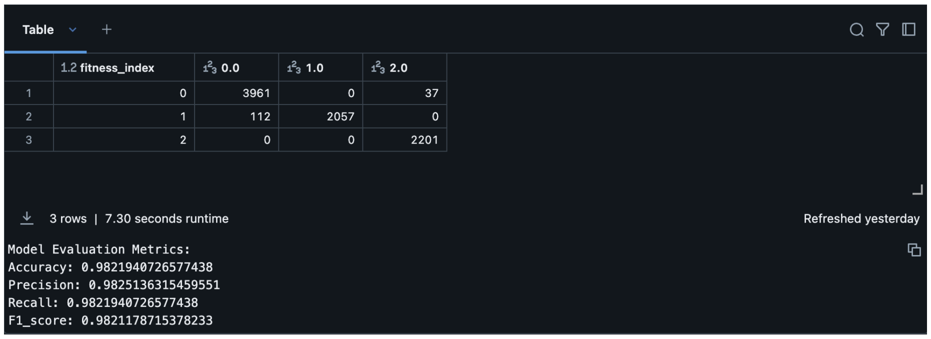
7.7 Decision Trees with best Parameters



```
spark.conf.set("spark.sql.shuffle.partitions", "200") # partitions set to 200

param_grid = ParamGridBuilder() \
    .addGrid(dt.maxDepth, [2, 5, 10]) \
    .addGrid(dt.minInstancesPerNode, [1, 5, 10]) \
    .addGrid(dt.minInfoGain, [0.2, 0.4, 0.6]) \
    .build()


cross_validator = CrossValidator(
    estimator=dt,
    estimatorParamMaps=param_grid,
    evaluator=MulticlassClassificationEvaluator(
        labelCol="fitness_index",
        predictionCol="prediction",
        metricName="accuracy"
    ),
    numFolds=3  # Number of folds for cross-validation
)

cv_model = cross_validator.fit(balanced_train_data)
```

Fig. 46. Decision Trees

7.8 Evaluation - Decision Trees with best Parameters



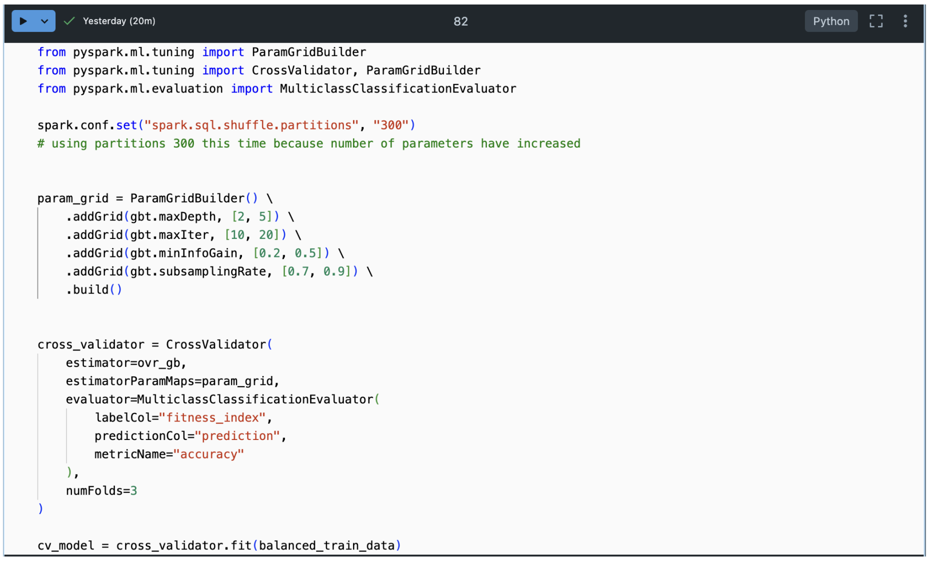
	fitness_index	0.0	1.0	2.0
1	0	3961	0	37
2	1	112	2057	0
3	2	0	0	2201

```
Model Evaluation Metrics:
Accuracy: 0.9821940726577438
Precision: 0.9825136315459551
Recall: 0.9821940726577438
F1_score: 0.9821178715378233
```

Fig. 47. Decision Trees

7.9 Gradient Boosting with best Parameters

To optimize the Gradient Boosted Trees (GBT) model for classification within the One-vs-Rest framework, I implemented cross-validation with an extensive parameter grid and increased the number of shuffle partitions in Spark to 300. The increase in partitions was necessary to handle the greater number of parameter combinations effectively, ensuring efficient parallelization and reducing computation time for the hyperparameter tuning process.



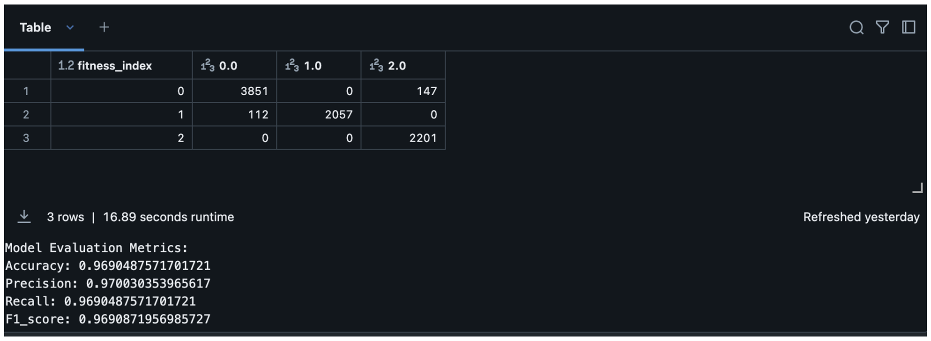
```python
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

spark.conf.set("spark.sql.shuffle.partitions", "300")
# using partitions 300 this time because number of parameters have increased


param_grid = ParamGridBuilder() \
    .addGrid(gbt.maxDepth, [2, 5]) \
    .addGrid(gbt.maxIter, [10, 20]) \
    .addGrid(gbt.minInfoGain, [0.2, 0.5]) \
    .addGrid(gbt.subsamplingRate, [0.7, 0.9]) \
    .build()


cross_validator = CrossValidator(
    estimator=ovr_gb,
    estimatorParamMaps=param_grid,
    evaluator=MulticlassClassificationEvaluator(
        labelCol="fitness_index",
        predictionCol="prediction",
        metricName="accuracy"
    ),
    numFolds=3
)

cv_model = cross_validator.fit(balanced_train_data)
```

Fig. 48. Gradient Boosting

7.10 Evaluation - Gradient Boosting with best Parameters

The evaluation was carried out using 3-fold cross-validation, where the dataset was divided into three subsets for training and testing in each fold. This procedure ensures that the model is robust and not overfitting to any particular portion of the data. The accuracy metric, computed through MulticlassClassificationEvaluator, was used to assess how well the model predicted the fitness index for the test data. Accuracy, as a classification metric, measures the proportion of correct predictions over the total number of predictions, making it a reliable indicator of the model's generalization performance.

The tuning of these hyperparameters was computationally intensive, and to enhance the process's efficiency, the number of shuffle partitions was set to 300, allowing Spark to handle the increased parameter grid and data splits more efficiently. The final model, represented by cvmodel, was optimized based on the highest accuracy achieved during the cross-validation, ensuring it is both accurate and efficient in predicting fitness categories.



	fitness_index	0.0	1.0	2.0
1	0	3851	0	147
2	1	112	2057	0
3	2	0	0	2201

```
Model Evaluation Metrics:
Accuracy: 0.9690487571701721
Precision: 0.970030353965617
Recall: 0.9690487571701721
F1_score: 0.9690871956985727
```

Fig. 49. Gradient Boosting

8 Conclusion

n conclusion, after evaluating multiple machine learning algorithms, the Random Forest model emerged as the most effective in predicting the fitness index, achieving an impressive accuracy of 98 percent. This result highlights the model's strong ability to generalize well on the dataset, especially after hyperparameter tuning.

The Support Vector Machine (SVM) model showed a notable improvement, with its accuracy jumping from 53 percent to 75 percent after performing cross-validation and optimizing hyperparameters. This demonstrates the potential of SVM in handling the classification task, especially with the right parameter configurations.

The Decision Tree classifier also displayed a significant enhancement, with its accuracy increasing from 47 percent to 97 percent after cross-validation and tuning. This improvement indicates that Decision Trees, when properly adjusted (e.g., through pruning and parameter optimization), can yield strong performance in predicting fitness categories.

Overall, while Random Forest performed the best, the improvements in both SVM and Decision Tree accuracy underscore the value of using cross-validation and parameter tuning to optimize model performance. These results emphasize the importance of model selection and fine-tuning in building an accurate predictive model for classification tasks.

9 References

1) Random Forest Classifier: scikit-learn. (2024). RandomForestClassifier. Retrieved from https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestClassifier.html

2) Support Vector Machine (SVM): scikit-learn. (2024). Support Vector Machines. Retrieved from https://scikit-learn.org/1.5/api/sklearn.svm.html

3) Decision Tree Classifier: scikit-learn. (2024). DecisionTreeClassifier. Retrieved from https://scikit-learn.org/1.5/modules/generated/sklearn.tree.DecisionTreeClassifier.html

4) Logistic Regression: scikit-learn. (2024). LogisticRegression. Retrieved from https://scikit-learn.org/1.5/modules/generated/sklearn.linearmodel.LogisticRegression.html

5) Gradient Boosting Classifier: scikit-learn. (2024). GradientBoostingClassifier. Retrieved from https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.GradientBoostingClassifi

6) CrossValidator: Apache Spark. (2024). CrossValidator (PySpark). Retrieved from https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.CrossValidato

7) StringIndexer: Apache Spark. (2024). StringIndexer (PySpark). Retrieved from https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StringIndexer.html