```
# Install Java 8
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
```

```
# Download Spark 3.1.2 (compatible with most PySpark versions)
!wget -q https://archive.apache.org/dist/spark/spark-3.1.2/spark-3.1.2-bin-hadoop2.7.tgz

# Extract Spark
!tar xf spark-3.1.2-bin-hadoop2.7.tgz
```

```
import os

# Setting the environment variables for Java and Spark
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.2-bin-hadoop2.7"
```

```
# Adding Spark to the system path
os.environ["PATH"] += os.pathsep + "/content/spark-3.1.2-bin-hadoop2.7/bin"
```

```
# Installing PySpark 3.1.2
!pip install pyspark==3.1.2
```

```
Collecting pyspark==3.1.2
  Downloading pyspark-3.1.2.tar.gz (212.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 212.4/212.4 MB 2.5 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Collecting py4j==0.10.9 (from pyspark==3.1.2)
  Downloading py4j-0.10.9-py2.py3-none-any.whl.metadata (1.3 kB)
Downloading py4j-0.10.9-py2.py3-none-any.whl (198 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 198.6/198.6 kB 13.6 MB/s eta 0:00:00
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.1.2-py2.py3-none-any.whl size=212880745 sha256=3795ba7407726e05914e6340b2040978a1f9
  Stored in directory: /root/.cache/pip/wheels/ef/70/50/7882e1bcb5693225f7cc86698f10953201b48b3f36317c2d18
Successfully built pyspark
Installing collected packages: py4j, pyspark
  Attempting uninstall: py4j
    Found existing installation: py4j 0.10.9.7
    Uninstalling py4j-0.10.9.7:
      Successfully uninstalled py4j-0.10.9.7
  Attempting uninstall: pyspark
    Found existing installation: pyspark 3.5.3
    Uninstalling pyspark-3.5.3:
      Successfully uninstalled pyspark-3.5.3
Successfully installed py4j-0.10.9 pyspark-3.1.2
```

```
from pyspark.sql import SparkSession

# Initializing the Spark session
spark = SparkSession.builder \
    .appName("PySpark Preprocessing") \
    .getOrCreate()
```

```
print("Spark session initialized successfully!")
```

```
Spark session initialized successfully!
```

```
# Loading the CSV file into a Spark DataFrame from the specified path
dataset_path = '/content/Data.csv'  # Your dataset path
df = spark.read.csv(dataset_path, header=True, inferSchema=True)

# Displaying the first few rows of the DataFrame
df.show(5)
```

```
+-------+---+------+---------+
|Country|Age|Salary|Purchased|
+-------+---+------+---------+
| France| 44| 72000|      Yes|
|  Spain| 27| 48000|      Yes|
|Germany| 30| 54000|       No|
|  Spain| 38| 61000|       No|
|Germany| 40|  NULL|      Yes|
+-------+---+------+---------+
only showing top 5 rows
```

```
# Task 1:

from pyspark.sql.functions import col, mean, expr

# Counting the missing values
df.select([count(when(col(c).isNull(), c)).alias(c) for c in df.columns]).show()

# Imputing 'Salary' with mean and 'Age' with median
salary_mean = df.select(mean(col("Salary"))).first()[0]
age_median = df.approxQuantile("Age", [0.5], 0.0)[0]

df_filled = df.fillna({"Salary": salary_mean, "Age": age_median})

# Imputing categorical columns with mode
for cat_col in ["Country", "Purchased"]:
    mode_value = df_filled.groupBy(cat_col).count().orderBy("count", ascending=False).first()[0]
    df_filled = df_filled.fillna({cat_col: mode_value})

df_filled.show(5)
```

```
+-------+---+------+---------+
|Country|Age|Salary|Purchased|
+-------+---+------+---------+
|      0|  1|     1|        0|
+-------+---+------+---------+

+-------+---+------+---------+
|Country|Age|Salary|Purchased|
+-------+---+------+---------+
| France| 44| 72000|      Yes|
|  Spain| 27| 48000|      Yes|
|Germany| 30| 54000|       No|
|  Spain| 38| 61000|       No|
|Germany| 40| 63777|      Yes|
+-------+---+------+---------+
only showing top 5 rows
```

Justification : The best approach is imputing missing values: mean for Salary to retain average characteristics, median for Age to handle outliers, and mode for categorical data to maintain category distribution, preserving data integrity efficiently.

```
# Task 2 :

from pyspark.ml.feature import MinMaxScaler, StandardScaler, VectorAssembler

# Assembling numerical features into a vector
assembler = VectorAssembler(inputCols=["Age", "Salary"], outputCol="features")
df_vector = assembler.transform(df_filled).select("features")

#  Applying Min-Max Scaling
min_max_scaler = MinMaxScaler(inputCol="features", outputCol="scaled_min_max")
df_min_max_scaled = min_max_scaler.fit(df_vector).transform(df_vector)

#  Applying Standardization
standard_scaler = StandardScaler(inputCol="features", outputCol="scaled_standard", withMean=True, withStd=True)
df_standard_scaled = standard_scaler.fit(df_vector).transform(df_vector)

#  Showing the scaled data
df_min_max_scaled.select("scaled_min_max").show(5, truncate=False)
df_standard_scaled.select("scaled_standard").show(5, truncate=False)
```

```
+----------------------------------------+
|scaled_min_max                          |
+----------------------------------------+
|[0.7391304347826086,0.6857142857142857]  |
|(2,[],[])                               |
|[0.13043478260869565,0.17142857142857143]|
|[0.4782608695652174,0.37142857142857144] |
|[0.5652173913043478,0.45077142857142855] |
+----------------------------------------+
only showing top 5 rows

+-----------------------------------------+
|scaled_standard                          |
+-----------------------------------------+
|[0.7302341624998382,0.7110194845078297]   |
|[-1.612026358726056,-1.3643691084877927]  |
|[-1.1986862667450158,-0.8455219602388871] |
|[-0.09644602146224211,-0.2402002872818306]|
|[0.1791140398584513,-6.053216729545398E-5]|
+------------------------------------------+
only showing top 5 rows
```

Scaling affects algorithms differently: distance-based algorithms (e.g., KNN, SVM) benefit from Min-Max Scaling, while algorithms sensitive to variance (e.g., linear regression) perform better with Standardization, as it normalizes data distribution around the mean. We scaled features to ensure comparability, enhancing algorithm performance and accuracy by normalizing values within a consistent range or distribution.

```
# Task 3

from pyspark.ml.feature import StringIndexer, OneHotEncoder

# Applying Label Encoding
label_indexer = StringIndexer(inputCol="Country", outputCol="Country_Index")
df_encoded = label_indexer.fit(df_filled).transform(df_filled)

#  Applying One-Hot Encoding for Country
onehot_encoder = OneHotEncoder(inputCols=["Country_Index"], outputCols=["Country_OHE"])
df_encoded = onehot_encoder.fit(df_encoded).transform(df_encoded)

# Applying Label Encoding for Purchased
label_indexer_purchased = StringIndexer(inputCol="Purchased", outputCol="Purchased_Index")
df_encoded = label_indexer_purchased.fit(df_encoded).transform(df_encoded)

# Showing the encoded data
df_encoded.select("Country", "Country_Index", "Country_OHE", "Purchased", "Purchased_Index").show(5, truncate=False)
```

```
+-------+-------------+-------------+---------+---------------+
|Country|Country_Index|Country_OHE  |Purchased|Purchased_Index|
+-------+-------------+-------------+---------+---------------+
|France |0.0          |(2,[0],[1.0])|Yes      |0.0            |
|Spain  |2.0          |(2,[],[])    |Yes      |0.0            |
|Germany|1.0          |(2,[1],[1.0])|No       |1.0            |
|Spain  |2.0          |(2,[],[])    |No       |1.0            |
|Germany|1.0          |(2,[1],[1.0])|Yes      |0.0            |
+-------+-------------+-------------+---------+---------------+
only showing top 5 rows
```

Label Encoding:

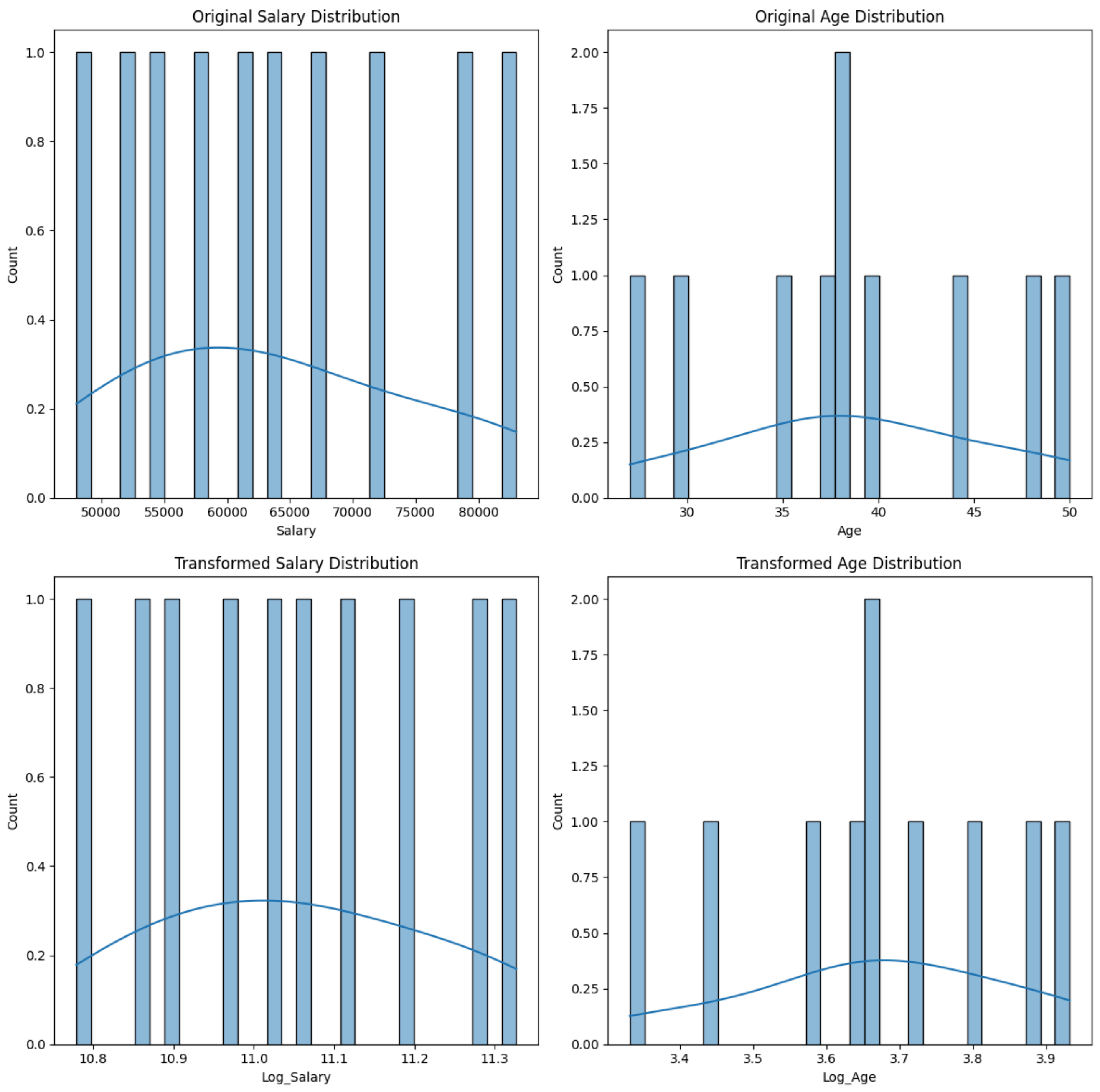
Converts categorical labels into numerical indices (e.g., "France" -> 0, "Spain" -> 1). Use it when the categorical variable is ordinal (i.e., has a natural order, such as "low," "medium," "high"). It retains the order and can work well with algorithms that can interpret the numeric representation.

One-Hot Encoding:

Converts categorical variables into a binary matrix (e.g., "France" becomes [1, 0, 0], "Spain" becomes [0, 1, 0]). Ideal for nominal categorical variables without an inherent order. It prevents the model from misinterpreting the numerical representation as having a rank or order.

```
# Task 4

from pyspark.sql.functions import log
import matplotlib.pyplot as plt
import seaborn as sns

# Creating a DataFrame for visualization
import pandas as pd

# Converting to Pandas DataFrame for visualization
df_pandas = df_encoded.select("Age", "Salary").toPandas()

# Visualizing original distribution
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
sns.histplot(df_pandas['Salary'], bins=30, kde=True)
plt.title('Original Salary Distribution')

plt.subplot(1, 2, 2)
sns.histplot(df_pandas['Age'], bins=30, kde=True)
plt.title('Original Age Distribution')

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

```
# Applying log transformation to reduce skewness
df_transformed = df_encoded.withColumn("Log_Salary", log(col("Salary"))) \
                            .withColumn("Log_Age", log(col("Age") + 1))  # Adding 1 to avoid log(0)

# Converting transformed DataFrame for visualization
df_transformed_pandas = df_transformed.select("Log_Salary", "Log_Age").toPandas()

#  Visualizing transformed distribution
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
sns.histplot(df_transformed_pandas['Log_Salary'], bins=30, kde=True)
plt.title('Transformed Salary Distribution')

plt.subplot(1, 2, 2)
sns.histplot(df_transformed_pandas['Log_Age'], bins=30, kde=True)
plt.title('Transformed Age Distribution')

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



Log transformation effectively reduces skewness by compressing the range of high values, making the distribution more normal. This enhances the performance of many machine learning models sensitive to feature scaling and distribution shape.

```
# Task 5

from pyspark.sql.functions import col

# Creating an interaction term between Age and Salary
df_features = df_encoded.withColumn("Age_Salary_Interaction", col("Age") * col("Salary"))
```

```
from pyspark.sql.functions import when

# Creating age groups
df_features = df_features.withColumn("Age_Group",
    when(col("Age") < 30, "18-29")
    .when((col("Age") >= 30) & (col("Age") < 40), "30-39")
    .when((col("Age") >= 40) & (col("Age") < 50), "40-49")
    .otherwise("50+"))
```

```
df_features = df_features.withColumn("Salary_per_Age", col("Salary") / col("Age"))
```

```
df_features = df_features.withColumn("Log_Salary", log(col("Salary")))
```

```
# Showing the updated DataFrame with the new features
df_features.show(5)
```

```
+-------+---+------+---------+-------------+-------------+---------------+----------------------+---------+------------------+-----
|Country|Age|Salary|Purchased|Country_Index|  Country_OHE|Purchased_Index|Age_Salary_Interaction|Age_Group|    Salary_per_Age|
+-------+---+------+---------+-------------+-------------+---------------+----------------------+---------+------------------+-----
| France| 44| 72000|      Yes|          0.0|(2,[0],[1.0])|            0.0|               3168000|    40-49|1636.3636363636363|11.18
|  Spain| 27| 48000|      Yes|          2.0|    (2,[],[])|            0.0|               1296000|    18-29|1777.7777777777778|10.77
|Germany| 30| 54000|       No|          1.0|(2,[1],[1.0])|            1.0|               1620000|    30-39|            1800.0|10.89
|  Spain| 38| 61000|       No|          2.0|    (2,[],[])|            1.0|               2318000|    30-39|1605.2631578947369|11.01
|Germany| 40| 63777|      Yes|          1.0|(2,[1],[1.0])|            0.0|               2551080|    40-49|          1594.425|11.06
+-------+---+------+---------+-------------+-------------+---------------+----------------------+---------+------------------+-----
only showing top 5 rows
```

By creating interaction terms and grouping continuous variables, we enable the model to capture relationships and patterns that may not be evident in the raw features alone. These engineered features can lead to better predictive performance and more robust models.

```
#Task 6
from pyspark.sql.functions import expr

# Calculating Q1 (25th percentile) and Q3 (75th percentile) for Salary
q1 = df_features.approxQuantile("Salary", [0.25], 0.0)[0]
q3 = df_features.approxQuantile("Salary", [0.75], 0.0)[0]
iqr = q3 - q1

# Determining outlier boundaries
lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr

# Identifying outliers
outliers = df_features.filter((col("Salary") < lower_bound) | (col("Salary") > upper_bound))
outliers_count = outliers.count()

print(f"Number of outliers detected in Salary: {outliers_count}")
```

```
Number of outliers detected in Salary: 0
```

```
# Option to cap outliers instead of removing
df_cleaned = df_features.withColumn("Salary_Capped",
    expr(f"CASE WHEN Salary < {lower_bound} THEN {lower_bound} WHEN Salary > {upper_bound} THEN {upper_bound} ELSE Salary END")
)
df_features.show(5)
```

```
+-------+---+------+---------+-------------+-------------+--------------+----------------------+---------+------------------+-----
|Country|Age|Salary|Purchased|Country_Index|  Country_OHE|Purchased_Index|Age_Salary_Interaction|Age_Group|    Salary_per_Age|
+-------+---+------+---------+-------------+-------------+--------------+----------------------+---------+------------------+-----
| France| 44| 72000|      Yes|          0.0|(2,[0],[1.0])|           0.0|               3168000|    40-49|1636.3636363636363|11.18
|  Spain| 27| 48000|      Yes|          2.0|    (2,[],[])|           0.0|               1296000|    18-29|1777.7777777777778|10.77
|Germany| 30| 54000|       No|          1.0|(2,[1],[1.0])|           1.0|               1620000|    30-39|            1800.0|10.89
|  Spain| 38| 61000|       No|          2.0|    (2,[],[])|           1.0|               2318000|    30-39|1605.2631578947369|11.01
|Germany| 40| 63777|      Yes|          1.0|(2,[1],[1.0])|           0.0|               2551080|    40-49|          1594.425|11.06
+-------+---+------+---------+-------------+-------------+--------------+----------------------+---------+------------------+-----
only showing top 5 rows
```

Removing or capping outliers ensures model stability and accuracy, preventing extreme values from skewing results and improving predictive performance.

```
# Task 7
# Splitting the data into training and testing sets (80-20 split)
train_df, test_df = df_cleaned.randomSplit([0.8, 0.2], seed=42)

# Print the count of records in each set
print(f"Training set count: {train_df.count()}")
print(f"Testing set count: {test_df.count()}")
```

```
Training set count: 7
Testing set count: 3
```

Data splitting is crucial to evaluate model performance on unseen data, ensuring it generalizes well and avoids overfitting.

Start coding or generate with AI.