



CIS5560 Term Project Tutorial



Authors: [Akanksha Khaire](#), [Dhwani Vaishnav](#), [Manimozhi Neethinayagam](#)

Instructor: [Jongwook Woo](#)

Date: 05/15/2024

Lab Tutorial

Akanksha Khaire (akhaire3@calstatela.edu)

Dhwani Vaishnav (dvaishn2@calstatela.edu)

Manimozhi Neethinayagam (mneethi@calstatela.edu)

Consumer Financial Protection Bureau Predictive Analysis using machine learning models in Spark ML

Objectives:

The CFPB, U.S. government agency dedicated to making sure consumers are treated fairly by banks, lenders, and other financial institutions. It also maintains a public database of consumer complaints related to financial products and services, which can be used to inform policy decisions and regulatory actions. The objective of this tutorial is to guide you through developing predictive models using the Consumer Financial Protection Bureau (CFPB) dataset. The key goals are:

a. **Predict Timely Responses:**

Develop a model to predict the likelihood of complaints receiving timely responses, helping to proactively address potential delays.

b. **Anticipate Company Responses:**

Create a model to anticipate the type of response a company will provide (e.g., closed with monetary relief) for efficient resource allocation.

c. **Uncover Topic Discovery:**

Utilize clustering techniques to discover underlying topics within the complaints data, providing deeper insights into consumer issues.

Platform Specifications

- HADOOP VERSION: Hadoop 3.3.3
- PYSARK VERSION: 3.2.1
- CPU SPEED: 1995.309 MHz
- NUMBER OF CPU CORES: 8
- NUMBER OF NODES: 5 (2 master nodes, 3 worker nodes)
- MEMORY SIZE: 806.39 GB

Dataset Specifications

- DATASET NAME: Consumer Financial Protection Bureau Dataset
- DATASET URL: <https://catalog.data.gov/dataset/consumer-complaint-database>
- TOTAL SIZE: 4.9 GB
- COUNTRY CONSIDERED: USA
- NUMBER OF FILES: 1
- FILE FORMAT: JSON

Step 1: Download the Dataset

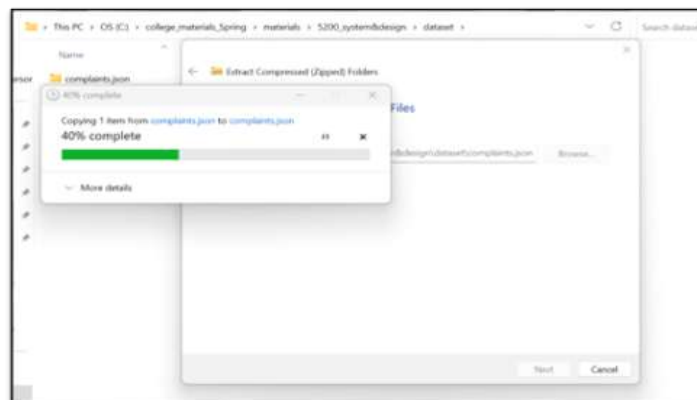
This step is to get data manually to the local system. Following are the steps to download:

1. [CFPB Dataset](#) – Download Dataset to local machine. You should have complaints.json in your local machine.

Note: This file contents can change depending on when it was downloaded since the complaints are registered daily.



2. Extract the complaints.json using zip(Extract All)



3. We have now extracted the json file

Name	Date modified	Type	Size
complaints.json	5/1/2023 4:35 PM	JSON File	3,602,460 KB

4. Now, you have to transfer the single json file from the local machine to HADOOP Cluster, which is shown in next section.

Step 2: Upload Files to Hadoop File system (HDFS)

Copy dataset using scp to the linux “tmp” directory.

Open a Gitbash session (Session 1) and execute the following command:

Remember to use your username and IP address of the cluster given.

```
scp C:/MSIS/CIS_5560/Project/Consumer_Complaints_Analysis/Dataset/complaints.json dvaishn2@129.153.214.22:/tmp
```

Now we must upload the file to HDFS folder. Run the following HDFS commands to create the directory in HDFS.

Step 3: Connect to Hadoop Spark cluster

For that open another shell terminal and paste the ssh command to connect to the Hadoop Spark cluster.

- You may download and install Git Bash: <https://git-scm.com/downloads>
 - If you use Linux or Mac computers, you can simply open “terminal”. You can search for “term” to find it.
- Then it will show you the terminal that you can use.

1. You must use your Calstate LA email account (dvaishn2) name to log in to the cluster, In a terminal, you need to type in the following ssh shell command to connect.

```
$ ssh dvaishn2@129.153.214.22
```

NOTE: Replace your username instead of “dvaishn2”

To enter password, type in your username as password and press enter.

Now you will be logged in.

```
AD+dvaishn2@STU-PF2XXWZK MINGW64 ~  
$ ssh dvaishn2@129.153.214.22  
dvaishn2@129.153.214.22's password:  
Last login: Fri Apr 26 01:56:43 2024 from 99-76-186-219.lightspeed.irvnca.sbcglobal.net
```

Step 4: Create a directory “5560_Complaints_DS” to put the file to HDFS.

- Run the following HDFS commands to create and list “5560_Complaints_DS” in HDFS:

```
hdfs dfs -mkdir 5560_Complaints_DS
```

```
hdfs dfs -ls
```

```
-bash-4.2$ hdfs dfs -ls  
Found 8 items  
drwx----- - dvaishn2 hdfs          0 2024-04-03 06:00 .Trash  
drwxr-xr-x - dvaishn2 hdfs          0 2024-04-25 22:09 .sparkStaging  
drwxr-xrwx - dvaishn2 hdfs          0 2024-04-02 21:32 5560_Complaints_DS  
drwxr-xr-x - dvaishn2 hdfs          0 2024-04-24 05:15 customer  
-rw-r--r-- 3 dvaishn2 hdfs 72088113 2024-03-26 23:54 flights.csv  
drwxr-xr-x - dvaishn2 hdfs          0 2024-04-24 05:15 movie  
drwxr-xr-x - dvaishn2 hdfs          0 2024-04-02 22:39 sample_data  
-rw-r--r-- 3 dvaishn2 hdfs 96368 2024-03-26 23:54 tweets.csv  
-bash-4.2$
```

- Once file is transferred to linux in step 2, the json file can be transferred from linux “tmp” location to the “5560_Complaints_DS” directory in HDFS using the “-put” command.

```
hdfs dfs -put /tmp/complaints.json 5560_Complaints_DS
```

```
hdfs dfs -ls 5560_Complaints_DS
```

Confirming the files are transferred to HDFS:

```
-bash-4.2$ hdfs dfs -ls 5560_Complaints_DS  
Found 2 items  
-rw-r--rw- 3 dvaishn2 hdfs 5088504860 2024-04-02 19:34 5560_Complaints_DS/complaints.json
```

Step 5: Download the python files from GitHub

Download the code files from GitHub and perform a spark-submit to run your Spark job on the cluster.

Navigate to Github Link: https://github.com/dvaishna/Consumer_Complaints_Machine_Learning

Download the 8 py files: complaints_narrative_LDA.py, decision_tree_final_class_8.py, desicion_tree_confusion_matrix.py, random_forest_confusion_matrix.py, random_forest_confusion_matrix.py, random_forest_final_class_8.py, timely_GBT.py, timely_LR.py, timely_SVM.py

Step 6: Connect to Hadoop Spark Cluster and Run PySpark

After downloading the code files from GitHub, follow these steps to run your PySpark job on the cluster:

1. **Download Code Files:** Clone or download the necessary code files from the provided GitHub repository.
2. **Connect to Hadoop Spark Cluster:**
 - a. Open a shell terminal.
 - b. Use the **ssh** command to connect to the Hadoop Spark cluster:

```
$ ssh dvaishn2@129.153.214.22
```

Note: Replace "dvaishn2" with your username.

3. **Run PySpark:** Start a PySpark session:

```
[~bash-4.2$ pyspark
```

```
PyTorch training
Welcome to
      _ _ _ _ _
     / _ _ _ \
    / _ _ _ \
   / _ _ _ \
  / _ _ _ \
 / _ _ _ \
/_ _ _ _ \
       version 3.2.1

Using Python version 3.6.8 (default, Jun  9 2023 11:59:08)
Spark context Web UI available at http://bigdaimn0.sub03291929060.trainingvcn.oraclevcn.com:4042
Spark context available as 'sc' (master = yarn, app id = application_1706556594272_2872).
SparkSession available as 'spark'.
>>> █
```

4. Step-by-Step Execution of Predictive Models.

A. Company Response Decision Tree:

This demonstrates how to perform classification on a complaints dataset using a Decision Tree model in Spark. The code covers several key steps: loading and preprocessing data, engineering and selecting features, building the model, tuning hyperparameters with CrossValidator, and evaluating model performance using various metrics.

```

"""This Spark code performs classification on a complaints dataset
using a Decision tree model trained with CrossValidator for
hyperparameter tuning.
Data Loading and Preprocessing:
Feature Engineering:
Feature Selection and Preprocessing:
Model Building:"""

import pandas as pd

from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.storagelevel import StorageLevel
from pyspark.sql.functions import lit
from pyspark.ml import Transformer
from pyspark.sql.functions import col
from pyspark.ml.tuning import TrainValidationSplit
from pyspark.mllib.evaluation import MulticlassMetrics

# PYSPARK_CLI = True
# if PYSPARK_CLI:
#     # sc = SparkContext.getOrCreate()
#     # spark = SparkSession(sc)

```

- **Data Preprocessing:**

- **Loading the Dataset:** The dataset is loaded from the HDFS into a Spark DataFrame.
- **Selecting Relevant Columns:** Only the columns necessary for model training are selected.
- **Handling Missing Values:** Any rows with missing values are dropped to ensure data quality.
- **Converting Data Types:** Ensure all columns are in the correct format for further processing.

```
#Data Loading and Preprocessing:
#-----
# Read the JSON file 'complaints.json' into a DataFrame named
'raw_complaints'
raw_complaints =
spark.read.json('/user/dvaishn2/5560_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records
complaint_df = raw_complaints.select('company', 'product',
'company_response' ,
'issue').filter(raw_complaints['_corrupt_record'].isNull())

complaint_df = complaint_df.filter(~(isNull(col("company")) |
(trim(col("company")) == "")))
complaint_df = complaint_df.filter(~(isNull(col("product")) |
(trim(col("product")) == "")))
complaint_df = complaint_df.filter(~(isNull(col("company_response")) |
(trim(col("company_response")) == "")))

# Show the first 10 rows of the DataFrame 'complaint_df'
complaint_df.show(10)

# Load dataset
df_company_response = complaint_df
```



```
# Select necessary columns and drop corrupt records
complaint_df = raw_complaints.select('company', 'product', 'company_response', 'issue').filter(raw_complaints['_corrupt_record'].isNull())

complaint_df = complaint_df.filter(~(isNull(col("company")) | (trim(col("company")) == "")))
complaint_df = complaint_df.filter(~(isNull(col("product")) | (trim(col("product")) == "")))
complaint_df = complaint_df.filter(~(isNull(col("company_response")) | (trim(col("company_response")) == "")))

>>> # Select necessary columns and drop corrupt records
.., complaint_df = raw_complaints.select('company', 'product', 'company_response', 'issue').filter(raw_complaints['_corrupt_record'].isNull())
>>>
>>> complaint_df = complaint_df.filter(~(isNull(col("company")) | (trim(col("company")) == "")))
>>> complaint_df = complaint_df.filter(~(isNull(col("product")) | (trim(col("product")) == "")))
>>> complaint_df = complaint_df.filter(~(isNull(col("company_response")) | (trim(col("company_response")) == "")))
```

- **Feature Engineering:**
 - **Indexing Categorical Columns:** Convert categorical features such as **product**, **issue**, and **company_response** into numerical indices using **StringIndexer**.
 - **Feature Extraction:** Extract additional features like **frequency_company** and **frequency_issue** to provide more context to the model.

```
#Feature Engineering:
#-----

# Calculate the frequency of each company

company_frequency =
df_company_response.groupBy("company").agg(count("*").alias("frequency_
company"))

# Join the frequency DataFrame with the original DataFrame on the
company column

df_response_with_frequency =
df_company_response.join(company_frequency, on="company", how="left")

# Calculate the frequency of each issue (corrected to avoid duplicate
calculation)

issue_frequency =
df_company_response.groupBy("issue").agg(count("*").alias("frequency_is
sue"))

# Join the issue frequency DataFrame with the existing DataFrame on the
issue column
```

```
df_response_with_frequency =  
df_response_with_frequency.join(issue_frequency, on="issue",  
how="left")  
  
# Show the result  
df_response_with_frequency.show(10)  
  
#Feature Selection and Preprocessing:  
#-----  
# Use the frequency column as a feature for modeling  
features = ["product", "frequency_company", "frequency_issue"]  
target = "company_response"  
  
from pyspark.storagelevel import StorageLevel  
  
df_response_with_frequency.persist(StorageLevel.MEMORY_ONLY)  
  
# String indexing for target variable  
target_indexer = StringIndexer(inputCol="company_response",  
outputCol="indexed_company_response")  
  
indexer_product = StringIndexer(inputCol="product",  
outputCol="indexed_product" , handleInvalid="skip")  
  
df_response_with_frequency = df_response_with_frequency.drop('company',  
'issue')
```

```
# Create VectorAssembler to combine the indexed product and hashed
company features

assembler = VectorAssembler(inputCols=["indexed_product",
"frequency_company", "frequency_issue"], outputCol="features")

# Create Decision Tree model

dt = DecisionTreeClassifier(labelCol="indexed_company_response",
featuresCol="features")
```

```
... df_company_response = complaint_df
>>> # Calculate the frequency of each company
... company_frequency = df_company_response.groupBy("company").agg(count("*").alias("frequency_company"))
>>> # Join the frequency DataFrame with the original DataFrame on the company column
... df_response_with_frequency = df_company_response.join(company_frequency, on="company", how="left")
>>> # Calculate the frequency of each issue (corrected to avoid duplicate calculation)
... issue_frequency = df_company_response.groupBy("issue").agg(count("*").alias("frequency_issue"))
>>> # Join the issue frequency DataFrame with the existing DataFrame on the issue column
... df_response_with_frequency = df_response_with_frequency.join(issue_frequency, on="issue", how="left")
>>> # Use the frequency column as a feature for modeling
... features = ["product", "frequency_company", "frequency_issue"]
>>> target = "company_response"
```

- **Create Indexers:** Index categorical columns.
- **Assemble Features:** Use **VectorAssembler** to combine feature columns into a single feature vector.

```
>>> from pyspark.storagelevel import StorageLevel
>>>
>>> df_response_with_frequency.persist(StorageLevel.MEMORY_ONLY)
DataFrame[issue: string, company: string, product: string, company_response: string, frequency_company: bigint, frequency_issue: bigint]
>>> # String indexing for target variable
... target_indexer = StringIndexer(inputCol="company_response", outputCol="indexed_company_response")
>>> indexer_product = StringIndexer(inputCol="product", outputCol="indexed_product", handleInvalid="skip")
>>>
>>> df_response_with_frequency = df_response_with_frequency.drop('company', 'issue')
>>> # Create VectorAssembler to combine the indexed product and hashed company features
... assembler = VectorAssembler(inputCols=["indexed_product", "frequency_company", "frequency_issue"], outputCol="features")
```

- **Balancing the Data:**
 - **Oversampling Minority Classes:** Duplicate instances of the minority class to ensure balanced representation.
 - **Undersampling Majority Classes:** Reduce the number of instances of the majority class to match the size of the minority class.
 - **Final Dataset:** A balanced dataset where each class has a roughly equal number of instances, improving model performance and fairness.

```
# Balancing the data_set:
```

```

#-----
# Define DataFrames for each response type

closed_with_explanation =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with explanation")

closed_with_non_monetary_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with non-monetary relief")

in_progress =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "In progress")

closed_with_monetary_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with monetary relief")

closed_without_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed without relief")

closed =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed")

untimely_response =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Untimely response")

closed_with_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with relief")


# Calculate current counts for each response type

counts = {

    "Closed with explanation": closed_with_explanation.count(),

```

```

    "Closed with non-monetary relief":
closed_with_non_monetary_relief.count(),

    "In progress": in_progress.count(),

    "Closed with monetary relief": closed_with_monetary_relief.count(),

    "Closed without relief": closed_without_relief.count(),

    "Closed": closed.count(),

    "Untimely response": untimely_response.count(),

    "Closed with relief": closed_with_relief.count()
}

# Calculate the oversampling factor for each category to achieve 15000
samples
target_count = 15000
oversampling_factors = {response: target_count / count for response,
count in counts.items()}

# Calculate sampling fractions to achieve the target count for each
category
sampling_fractions = {category: target_count / count for category,
count in counts.items()}

# Create an empty DataFrame to hold the balanced data
balanced_data = spark.createDataFrame([],
df_response_with_frequency.schema)

# Sample each category to achieve the target count
for category, count in counts.items():

    # Sample the category DataFrame with the calculated fraction

```

```

    sampled_df =
df_response_with_frequency.filter(df_response_with_frequency["company_r
response"] == category)\

                                .sample(withReplacement=True,
fraction=sampling_fractions[category], seed=42)

    # Union the sampled DataFrame with the balanced data

    balanced_data = balanced_data.union(sampled_df)

# Show the count of each category in the balanced data
balanced_data.groupBy("company_response").count().orderBy("company_resp
onse").show()

```

```

>>> # Define DataFrames for each response type
... closed_with_explanation = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with explanation")
... closed_with_non_monetary_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with non-monetary relief")
... in_progress = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "In progress")
... closed_with_monetary_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with monetary relief")
... closed_with_non_monetary_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with non-monetary relief")
... closed_without_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed without relief")
... in_progress = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "In progress")
... closed_with_monetary_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with monetary relief")
... closed_without_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed without relief")
... closed = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed")
... closed = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed")
... untimely_response = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Untimely response")
... closed_with_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with relief")
>>> # Calculate current counts for each response type
... counts = {
...     "Closed with explanation": closed_with_explanation.count(),
...     "Closed with non-monetary relief": closed_with_non_monetary_relief.count(),
...     "In progress": in_progress.count(),
...     "Closed with monetary relief": closed_with_monetary_relief.count(),
...     "Closed without relief": closed_without_relief.count(),
...     "Closed": closed.count(),
...     "Untimely response": untimely_response.count(),
...     "Closed with relief": closed_with_relief.count()
... }
>>> # Calculate the oversampling factor for each category to achieve 15000 samples
... target_count = 15000
>>> oversampling_factors = {response: target_count / count for response, count in counts.items()}
>>>
>>> # Calculate sampling fractions to achieve the target count for each category
... sampling_fractions = {category: target_count / count for category, count in counts.items()}
>>>
>>> # Create an empty DataFrame to hold the balanced data
... balanced_data = spark.createDataFrame([], df_response_with_frequency.schema)

# Sample each category to achieve the target count
for category, count in counts.items():
    # Sample the category DataFrame with the calculated fraction
    >>>
>>> # Sample each category to achieve the target count
... for category, count in counts.items():
...     # Sample the category DataFrame with the calculated fraction
...     sampled_df = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == category)\
...                 .sample(withReplacement=True, fraction=sampling_fractions[category], seed=42)
...     # Union the sampled DataFrame with the balanced data
...     balanced_data = balanced_data.union(sampled_df)

```

- **Prepare Pipeline**
 - **Build Pipeline:** Combine indexers and assembler into a **Pipeline** object for streamlined data transformation.

```
#pipeline:
#-----

# Create a pipeline with the VectorAssembler and Decision Tree model
pipeline = Pipeline(stages=[indexer_product, target_indexer, assembler,
dt])

# Split the data into training and testing sets
train_data, test_data = balanced_data.randomSplit([0.7, 0.3], seed=42)

train_rows = train_data.count()
test_rows = test_data.count()

# Print the counts
print("Training Rows:", train_rows, " Testing Rows:", test_rows)
```

```
>>> # Create a pipeline with the VectorAssembler and Decision Tree model
... pipeline = Pipeline(stages=[indexer_product, target_indexer, assembler, dt])
```

- **Model Building and Training:**

First, a Decision Tree model (Decision tree Classifier) is created. This model is part of a pipeline that also includes various feature preprocessing stages. The pipeline ensures that all preprocessing steps and the model are executed in sequence. For hyperparameter tuning, a ParamGridBuilder is set up to specify different combinations of hyperparameters. In this case, maxDepth and minInstancesPerNode are the hyperparameters being tuned, with multiple values tested for each.

An evaluator using `MulticlassClassificationEvaluator` is defined to measure the model's accuracy. The `CrossValidator` is then configured with the pipeline, parameter grid, and evaluator. It performs cross-validation with three folds to identify the best combination of hyperparameters.

To train the model, the time is recorded before and after fitting the `CrossValidator` to the training data. The difference gives the total training time, which is then formatted into minutes and seconds for readability.

Here's the code for this process:

```
#Model Building:
#-----

evaluator =
MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="indexed_company_response", metricName="accuracy")

paramGrid = ParamGridBuilder() \
    .addGrid(dt.maxDepth, [3, 5, 7]) \
    .addGrid(dt.minInstancesPerNode, [1, 5, 10]) \
    .build()

# Define CrossValidator
crossval = CrossValidator(estimator=pipeline,
                           estimatorParamMaps=paramGrid,
                           evaluator=evaluator,
                           numFolds=3)

#Model Training:
# Training the model and calculating its time

import time
# Start time
```



```

start_time = time.time()

# Fit the cross validator to the training data
cvModel = crossval.fit(train_data)

# End time
end_time = time.time()

print("Model trained!")

# Calculate training time
training_time = end_time - start_time

# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

# Format the time
training_time_formatted = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time
print("Training time CrossValidator:", training_time_formatted)

```

- **Dataset Split:** Split the data into training (83,763 rows) and test sets (36,153 rows).
- **Train Decision Tree Model:** Train the model using the training data.
- **Training Time:** The model training took approximately 29:33 minutes.

```
>>> # Split the data into training and testing sets
... train_data, test_data = balanced_data.randomSplit([0.7, 0.3], seed=42)

train_rows = train_data.count()
>>>
>>> train_rows = train_data.count()
test_rows = test_data.count()
>>> test_rows = test_data.count()
>>> # Print the counts
... print("Training Rows:", train_rows, " Testing Rows:", test_rows)
Training Rows: 83763 Testing Rows: 36153
```

```
>>> evaluator = MulticlassClassificationEvaluator(predictionCol="prediction", labelCol="indexed_company_response", metricName="accuracy")
>>> paramGrid = ParamGridBuilder() \
...     .addGrid(dt.maxDepth, [3, 5, 7]) \
...     .addGrid(dt.minInstancesPerNode, [1, 5, 10]) \
...     .build()
>>> # Define CrossValidator
... crossval = CrossValidator(estimatorPipeline,
...                             estimatorParamMaps=paramGrid,
...                             evaluator=evaluator,
...                             numFolds=3)
>>> # Training the model and calculating its time
... import time
>>>
>>> # Start time
... start_time = time.time()
>>> # Fit the cross validator to the training data
... cvModel = crossval.fit(train_data)
end_time = time.time()

print("Model trained!")

# Calculate training time
training_time = end_time - start_time

# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

# Format the time
training_time_formatted = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time
print("Training time CrossValidator:", training_time_formatted)
>>>
>>> # End time
... end_time = time.time()
```

```
>>> print("Model trained!")
Model trained!
>>>
>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>>
>>> # Format the time
... training_time_formatted = "{:02d}:{:02d}".format(minutes, seconds)
>>>
>>> # Print training time
... print("Training time CrossValidator:", training_time_formatted)
Training time CrossValidator: 29:33
>>> |
```

- **Feature Importance – Cross Validation:**

After training the model using cross-validation, the best model is selected from the CrossValidator results. The feature importances are accessed from the Random Forest model, which is assumed to be the last stage of the pipeline. The VectorAssembler provides the feature names, and these are paired with their

corresponding importance values to create a DataFrame. This DataFrame is then sorted in descending order of importance to highlight the most significant features. Finally, the sorted feature importance DataFrame is printed.

Here's the code for this process:

```
# Feature Importance crossvalidation:
#-----

# Get the fitted model from CrossValidator
bestModel = cvModel.bestModel

# Access the feature importances from the Random Forest model within
the pipeline
feature_importances = bestModel.stages[-1].featureImportances #
Assuming Random Forest is the last stage

# Get feature names from the VectorAssembler
feature_names = assembler.getInputCols()

# Create a DataFrame of feature importances
featureImp = pd.DataFrame(list(zip(feature_names,
feature_importances)), columns=["feature", "importance"])

# Sort the DataFrame by importance (descending order)
featureImp = featureImp.sort_values(by="importance", ascending=False)

# Print the DataFrame with feature importance
print("\nFeature Importance:")
print(featureImp.round(2).to_string(index=False))
```

- **frequency_company**: Importance = 0.55
- **product**: Importance = 0.41
- **frequency_issue**: Importance = 0.04

```
Feature Importance:
>>> print(featureImp.to_string(index=False))
      feature  importance
frequency_company  0.548747
indexed_product    0.406077
frequency_issue    0.045176
```

- **Model Evaluation – Cross Validation Decision Tree Confusion Matrix (Testing the Data):**

To evaluate the model's performance on unseen data, predictions are made using the best model selected by cross-validation. The predictions are made on the test dataset. The prediction results are then prepared for evaluation by selecting and casting the relevant columns to **FloatType** and ordering by prediction. This preparation ensures compatibility with the evaluation metrics.

The **MulticlassMetrics** from **pyspark.mllib.evaluation** is used to assess the model's performance. This involves converting the prediction results into an RDD of tuples, which **MulticlassMetrics** can then process to compute various performance metrics.

Here's the code for this process:

```
#Test the Data:
#-----

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

from pyspark.sql.types import FloatType
#important: need to cast to float type, and order by prediction, else
it won't work
preds_and_labels =
predictions.select(['prediction','indexed_company_response'])\
              .withColumn('indexed_company_response',
col('indexed_company_response')\
              .cast(FloatType()))\
```

```
.orderBy('prediction')
```

```
from pyspark.mllib.evaluation import MulticlassMetrics
metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))
```

```
>>> from pyspark.sql.types import FloatType
>>> #important: need to cast to float type, and order by prediction, else it won't work
>>> preds_and_labels = predictions.select(['prediction', 'indexed_company_response'])\
>>> .withColumn('indexed_company_response', col('indexed_company_response')\
>>> .cast(FloatType()))\
>>> .orderBy('prediction')
>>>
>>> from pyspark.mllib.evaluation import MulticlassMetrics
>>> metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))

/usr/odh/current/spark3-client/python/pyspark/sql/context.py:127: FutureWarning: Deprecated in 3.0.0: Use SparkSession.builder.getOrCreate() instead.
FutureWarning
>>>
>>> confusion_matrix = metrics.confusionMatrix().toArray()
>>>
>>> print(metrics.confusionMatrix().toArray())
[[3.730e+03 1.410e+02 2.000e+00 8.100e+01 6.200e+01 4.000e+00 4.630e+02
 0.000e+00]
 [2.710e+02 1.402e+03 1.024e+03 5.710e+02 3.990e+02 1.280e+02 5.030e+02
 3.240e+02]
 [3.900e+01 4.750e+02 2.231e+03 1.730e+02 8.700e+01 5.800e+01 1.520e+02
 1.234e+03]
 [1.290e+02 9.600e+01 5.600e+01 2.579e+03 4.690e+02 8.000e+02 3.660e+02
 0.000e+00]
 [1.000e+00 0.000e+00 0.000e+00 0.000e+00 2.957e+03 1.143e+03 3.780e+02
 0.000e+00]
 [0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.497e+03 2.473e+03 5.320e+02
 0.000e+00]
 [4.840e+02 3.300e+01 5.700e+01 2.800e+01 1.142e+03 2.420e+02 2.552e+03
 0.000e+00]
 [1.800e+01 8.800e+01 3.000e+00 7.800e+01 4.000e+00 7.000e+00 5.000e+01
 4.337e+03]]
```

```
>>> confusion_matrix = metrics.confusionMatrix().toArray()
>>>
>>> print(metrics.confusionMatrix().toArray())
[[3.730e+03 1.410e+02 2.000e+00 8.100e+01 6.200e+01 4.000e+00 4.630e+02
 0.000e+00]
 [2.710e+02 1.402e+03 1.024e+03 5.710e+02 3.990e+02 1.280e+02 5.030e+02
 3.240e+02]
 [3.900e+01 4.750e+02 2.231e+03 1.730e+02 8.700e+01 5.800e+01 1.520e+02
 1.234e+03]
 [1.290e+02 9.600e+01 5.600e+01 2.579e+03 4.690e+02 8.000e+02 3.660e+02
 0.000e+00]
 [1.000e+00 0.000e+00 0.000e+00 0.000e+00 2.957e+03 1.143e+03 3.780e+02
 0.000e+00]
 [0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.497e+03 2.473e+03 5.320e+02
 0.000e+00]
 [4.840e+02 3.300e+01 5.700e+01 2.800e+01 1.142e+03 2.420e+02 2.552e+03
 0.000e+00]
 [1.800e+01 8.800e+01 3.000e+00 7.800e+01 4.000e+00 7.000e+00 5.000e+01
 4.337e+03]]
>>> |
```

```
... print(confusion_matrix_2d)
[[3730.0, 141.0, 2.0, 81.0, 62.0, 4.0, 463.0, 0.0], [271.0, 1402.0, 1024.0, 571.0, 399.0, 128.0, 503.0, 324.0], [39.0, 475.0, 2231.0, 173.0, 87.0, 58.0, 152.0, 1234.0], [129.0, 96.0, 56.0, 2579.0, 469.0, 800.0, 366.0, 0.0], [1.0, 0.0, 0.0, 0.0, 2957.0, 1143.0, 378.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1497.0, 2473.0, 532.0, 0.0], [484.0, 33.0, 57.0, 28.0, 1142.0, 242.0, 2552.0, 0.0], [18.0, 88.0, 3.0, 78.0, 4.0, 7.0, 50.0, 4337.0]]
```

- **Model Evaluation – Cross Validator Results:**

The evaluation results are summarized by first printing the confusion matrix derived from the model's predictions. The confusion matrix provides insight into the performance across different classes. Diagonal elements represent true positives (TPs), while off-diagonal elements help calculate false positives (FPs), false negatives (FNs), and true negatives (TNs).

Precision, recall, and accuracy are calculated for each class based on the confusion matrix. These metrics provide a detailed view of the model's performance, highlighting how well it predicts each specific class. Class names are mapped to their respective numerical labels, making the results more interpretable. The precision, recall, and accuracy for each class are then printed.

Here's the code for this process:

```
#Result:crossvalidation
#-----

confusion_matrix = metrics.confusionMatrix().toArray()

print(metrics.confusionMatrix().toArray())

import numpy as np

tps = np.diag(confusion_matrix)
fps = np.sum(confusion_matrix, axis=0) - tps
fns = np.sum(confusion_matrix, axis=1) - tps
tns = np.sum(confusion_matrix) - tps - fps - fns

# Calculate precision, recall, and accuracy for each class
precision = tps / (tps + fps)
recall = tps / (tps + fns)
accuracy = (tps + tns) / (tps + tns + fps + fns)

"""Class 1: "Closed with explanation"
Class 2: "Closed with non-monetary relief"
Class 3: "In progress"
Class 4: "Closed with monetary relief"
Class 5: "Closed without relief"
Class 6: "Closed"
Class 7: "Untimely response"
Class 8: "Closed with relief" """
```

```
class_names = {
    1: "Closed with explanation",
    2: "Closed with non-monetary relief",
    3: "In progress",
    4: "Closed with monetary relief",
    5: "Closed without relief",
    6: "Closed",
    7: "Untimely response",
    8: "Closed with relief"
}

for i in range(len(precision)):
    class_name = class_names[i + 1]
    print(f"{class_name}:")
    print(f"Precision: {precision[i]:.2f}")
    print(f"Recall: {recall[i]:.2f}")
    print(f"Accuracy: {accuracy[i]:.2f}")
    print()

confusion_matrix_2d = confusion_matrix.tolist()

# Print the 2D list
print(confusion_matrix_2d)
```

```

>>> import numpy as np
>>>
>>> tps = np.diag(confusion_matrix)
>>> fps = np.sum(confusion_matrix, axis=0) - tps
>>> fns = np.sum(confusion_matrix, axis=1) - tps
>>> tns = np.sum(confusion_matrix) - tps - fps - fns
>>>
>>> # Calculate precision, recall, and accuracy for each class
>>> precision = tps / (tps + fps)
>>> recall = tps / (tps + fns)
>>> accuracy = (tps + tns) / (tps + tns + fps + fns)
>>>
>>> """Class 1: "Closed with explanation"
>>> Class 2: "Closed with non-monetary relief"
>>> Class 3: "In progress"
>>> Class 4: "Closed with monetary relief"
>>> Class 5: "Closed without relief"
>>> Class 6: "Closed"
>>> Class 7: "Untimely response"
>>> Class 8: "Closed with relief" """
>>> class_names = ["Class 1: \"Closed with explanation\"", "Class 2: \"Closed with non-monetary relief\"", "Class 3: \"In progress\"", "Class 4: \"Closed with monetary relief\"", "Class 5: \"Closed without relief\"", "Class 6: \"Closed\"", "Class 7: \"Untimely response\"", "Class 8: \"Closed with relief\""]
>>>
>>> class_names = {
>>>     1: "Closed with explanation",
>>>     2: "Closed with non-monetary relief",
>>>     3: "In progress",
>>>     4: "Closed with monetary relief",
>>>     5: "Closed without relief",
>>>     6: "Closed",
>>>     7: "Untimely response",
>>>     8: "Closed with relief"
>>> }
>>>
>>> for i in range(len(precision)):
>>>     class_name = class_names[i + 1]
>>>     print(f"Class {i+1}: {class_name}")
>>>     print(f"Precision: {precision[i]:.2f}")
>>>     print(f"Recall: {recall[i]:.2f}")
>>>     print(f"Accuracy: {accuracy[i]:.2f}")
>>>     print()

```

- **Evaluation Metrics:**

- **Closed with explanation:** Precision: 0.80, Recall: 0.83, Accuracy: 0.95
- **Closed with non-monetary relief:** Precision: 0.63, Recall: 0.30, Accuracy: 0.89
- **In progress:** Precision: 0.66, Recall: 0.50, Accuracy: 0.91
- **Closed with monetary relief:** Precision: 0.73, Recall: 0.57, Accuracy: 0.92
- **Closed without relief:** Precision: 0.45, Recall: 0.66, Accuracy: 0.86
- **Closed:** Precision: 0.51, Recall: 0.55, Accuracy: 0.88
- **Untimely response:** Precision: 0.51, Recall: 0.56, Accuracy: 0.88
- **Closed with relief:** Precision: 0.74, Recall: 0.95, Accuracy: 0.95


```
Closed with explanation:
Precision: 0.80
Recall: 0.83
Accuracy: 0.95

Closed with non-monetary relief:
Precision: 0.63
Recall: 0.30
Accuracy: 0.89

In progress:
Precision: 0.66
Recall: 0.50
Accuracy: 0.91

Closed with monetary relief:
Precision: 0.73
Recall: 0.57
Accuracy: 0.92

Closed without relief:
Precision: 0.45
Recall: 0.66
Accuracy: 0.86

Closed:
Precision: 0.51
Recall: 0.55
Accuracy: 0.88

Untimely response:
Precision: 0.51
Recall: 0.56
Accuracy: 0.88

Closed with relief:
Precision: 0.74
Recall: 0.95
Accuracy: 0.95
```

- **Train Validation Split:**

To evaluate the model's performance with a different validation approach, `TrainValidationSplit` is used. This method splits the data into training and validation sets based on the specified ratio (80% training and 20% validation). The same pipeline and parameter grid used in cross-validation are applied here for consistency. Training time is recorded by noting the start and end times of fitting the `TrainValidationSplit` model to the training data. The total training time is calculated and formatted for readability.

Finally, predictions are made on the test data using the best model identified by `TrainValidationSplit`.

Here's the code for this process:

```
#Train Validation
#-----
# Define TrainValidationSplit
from pyspark.ml.tuning import TrainValidationSplit
trainval = TrainValidationSplit(estimator=pipeline,
                                estimatorParamMaps=paramGrid,
                                evaluator=evaluator,
                                trainRatio=0.8)

#Training the model and Calculating its time
import time
# Start time
start_time = time.time()

# Fit the cross validator to the training data
tvModel = trainval.fit(train_data)

# End time
end_time = time.time()
print("Model trained!")

# Calculate training time
training_time = end_time - start_time
# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)
```

```

# Format the time

training_time_formatted = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time

print("Training time TrainValidator:", training_time_formatted)

# Make predictions on the test data using the best model

predictions = tvModel.transform(test_data)

```

- **Training Time:** Training using Train Validation Split took approximately 17:16 minutes.

```

>>> #Training the model and Calculating its time
... import time
>>>
>>> # Start time
... start_time = time.time()
>>>
>>> # Fit the cross validator to the training data
... tvModel = trainval.fit(train_data)

# End time
end_time = time.time()
print("Model trained!")

# Calculate training time
training_time = end_time - start_time

# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

# Format the time
training_time_formatted = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time
print("Training time TrainValidator:", training_time_formatted)
>>>
>>> # End time
... end_time = time.time()
>>>
>>> print("Model trained!")
Model trained!
>>>
>>>
>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>>
>>> # Format the time
... training_time_formatted = "{:02d}:{:02d}".format(minutes, seconds)
>>>
>>> # Print training time
... print("Training time TrainValidator:", training_time_formatted)
Training time TrainValidator: 17:16

```

- **Feature Importance – Train Validation Split:**

After training the model using TrainValidationSplit, the best model is selected for analysis. The feature importances are accessed from the Random Forest model within the pipeline, assuming it is the last stage. The VectorAssembler provides the feature names, which are paired with their corresponding importance values to create a DataFrame. This DataFrame is then sorted in descending order of importance to highlight the most significant features.

Here's the code for this process:

```
#Feature Importance:trainvalidation
#-----

# Get the fitted model from CrossValidator
bestModel = tvModel.bestModel

# Access the feature importances from the Random Forest model within
the pipeline
feature_importances_tv = bestModel.stages[-1].featureImportances #
Assuming Random Forest is the last stage

# Get feature names from the VectorAssembler
feature_names_tv = assembler.getInputCols()

# Create a DataFrame of feature importances
featureImp_tv = pd.DataFrame(list(zip(feature_names_tv,
feature_importances_tv)), columns=["feature", "importance"])

# Sort the DataFrame by importance (descending order)
featureImp_tv = featureImp_tv.sort_values(by="importance",
ascending=False)

# Print the DataFrame with feature importance
print("\nFeature Importance:")
print(featureImp_tv.round(2).to_string(index=False))
```

- **frequency_company**: Importance = 0.55
- **product**: Importance = 0.41
- **frequency_issue**: Importance = 0.05

```
... feature_importances_tv = bestModel.stages[-1].featureImportances
>>> # Get feature names from the VectorAssembler
... feature_names_tv = assembler.getInputCols()
>>> # Create a DataFrame of feature importances
... featureImp_tv = pd.DataFrame(list(zip(feature_names_tv, feature_importances_tv)), columns=["feature", "importance"])
>>> # Sort the DataFrame by importance (descending order)
... featureImp_tv = featureImp_tv.sort_values(by="importance", ascending=False)
>>> # Print the DataFrame with feature importance
... print("\nFeature Importance:")

Feature Importance:
>>> print(featureImp_tv.round(2).to_string(index=False))
      feature  importance
frequency_company      0.55
indexed_product        0.41
frequency_issue         0.05
```

```
Feature Importance:
>>> print(featureImp_tv.round(2).to_string(index=False))
      feature  importance
frequency_company      0.55
indexed_product        0.41
frequency_issue         0.05
```

- **Model Evaluation – Train Validation Split Results:**

The evaluation of the model trained using TrainValidationSplit involves making predictions on the test data and analyzing the results. Predictions are cast to FloatType and ordered by prediction to ensure compatibility with evaluation metrics.

The MulticlassMetrics class from pyspark.mllib.evaluation is used to compute various performance metrics from the prediction results. The confusion matrix is extracted and printed, providing a detailed view of the model's performance across different classes.

Key metrics such as precision, recall, and accuracy are calculated for each class. These metrics help in understanding how well the model predicts each class and overall performance.

Class names are mapped to their respective numerical labels to make the results more interpretable.

Here's the code for this process:

```
#Result:TrainValidationSplit

#-----

from pyspark.sql.types import FloatType
```

```

#important: need to cast to float type, and order by prediction, else
it won't work

preds_and_labels =
predictions.select(['prediction','indexed_company_response'])\
                .withColumn('indexed_company_response',
col('indexed_company_response')\
                .cast(FloatType()))\
                .orderBy('prediction')

metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))

confusion_matrix = metrics.confusionMatrix().toArray()

print(metrics.confusionMatrix().toArray())

import numpy as np

tps = np.diag(confusion_matrix)
fps = np.sum(confusion_matrix, axis=0) - tps
fns = np.sum(confusion_matrix, axis=1) - tps
tns = np.sum(confusion_matrix) - tps - fps - fns

# Calculate precision, recall, and accuracy for each class
precision = tps / (tps + fps)
recall = tps / (tps + fns)
accuracy = (tps + tns) / np.sum(confusion_matrix)

"""Class 1: "Closed with explanation"
Class 2: "Closed with non-monetary relief"

```

```
Class 3: "In progress"
Class 4: "Closed with monetary relief"
Class 5: "Closed without relief"
Class 6: "Closed"
Class 7: "Untimely response"
Class 8: "Closed with relief" ""

# Define a dictionary mapping class indices to class names
class_names = {
    1: "Closed with explanation",
    2: "Closed with non-monetary relief",
    3: "In progress",
    4: "Closed with monetary relief",
    5: "Closed without relief",
    6: "Closed",
    7: "Untimely response",
    8: "Closed with relief"
}
```

```

>>> import numpy as np
>>>
... tps = np.diag(confusion_matrix)
>>> fps = np.sum(confusion_matrix, axis=0) - tps
>>> fns = np.sum(confusion_matrix, axis=1) - tps
>>> tns = np.sum(confusion_matrix) - tps - fps - fns
>>>
>>> # Calculate precision, recall, and accuracy for each class
... precision = tps / (tps + fps)
... recall = tps / (tps + fns)
... accuracy = (tps + tns) / np.sum(confusion_matrix)
>>>
>>> """Class 1: "Closed with explanation"
... Class 2: "Closed with non-monetary relief"
... Class 3: "In progress"
... Class 4: "Closed with monetary relief"
... Class 5: "Closed without relief"
... Class 6: "Closed"
... Class 7: "Untimely response"
... Class 8: "Closed with relief" """
>>> "Class 1: "Closed with explanation"\nClass 2: "Closed with non-monetary relief"\nClass 3: "In progress"\nClass 4: "Closed with monetary relief"\nClass 5: "Closed without r
... lief"\nClass 6: "Closed"\nClass 7: "Untimely response"\nClass 8: "Closed with relief" "
>>>
>>>
>>> # Define a dictionary mapping class indices to class names
... class_names = {
...     1: "Closed with explanation",
...     2: "Closed with non-monetary relief",
...     3: "In progress",
...     4: "Closed with monetary relief",
...     5: "Closed without relief",
...     6: "Closed",
...     7: "Untimely response",
...     8: "Closed with relief"
... }
>>>
>>> for i in range(len(precision)):
...     class_name = class_names[i + 1]
...     print(f"{class_name}:")
...     print(f"Precision: {precision[i]:.2f}")
...     print(f"Recall: {recall[i]:.2f}")
...     print(f"Accuracy: {accuracy[i]:.2f}")
...     print()

```

- **Evaluation Metrics:**

The model's performance metrics are printed for each class to provide detailed insights. Precision, recall, and accuracy are calculated and displayed, giving a clear understanding of how well the model performs for each specific class.

The class names are mapped to their respective numerical labels, making the results more interpretable.

The confusion matrix, converted into a 2D list, is also printed to visualize the model's performance in a matrix format.

Here's the code for this process:

```

#Result:
#-----

for i in range(len(precision)):
    class_name = class_names[i + 1]
    print(f"{class_name}:")
    print(f"Precision: {precision[i]:.2f}")
    print(f"Recall: {recall[i]:.2f}")
    print(f"Accuracy: {accuracy[i]:.2f}")
    print()

confusion_matrix_2d = confusion_matrix.tolist()

# Print the 2D list
print(confusion_matrix_2d)

```


- **Closed with explanation:** Precision: 0.80, Recall: 0.83, Accuracy: 0.95
- **Closed with non-monetary relief:** Precision: 0.63, Recall: 0.30, Accuracy: 0.89
- **In progress:** Precision: 0.66, Recall: 0.50, Accuracy: 0.91
- **Closed with monetary relief:** Precision: 0.73, Recall: 0.57, Accuracy: 0.92
- **Closed without relief:** Precision: 0.45, Recall: 0.66, Accuracy: 0.86
- **Closed:** Precision: 0.51, Recall: 0.55, Accuracy: 0.88
- **Untimely response:** Precision: 0.51, Recall: 0.56, Accuracy: 0.88
- **Closed with relief:** Precision: 0.74, Recall: 0.95, Accuracy: 0.95

```
Closed with explanation:
Precision: 0.80
Recall: 0.83
Accuracy: 0.95

Closed with non-monetary relief:
Precision: 0.63
Recall: 0.30
Accuracy: 0.89

In progress:
Precision: 0.66
Recall: 0.50
Accuracy: 0.91

Closed with monetary relief:
Precision: 0.73
Recall: 0.57
Accuracy: 0.92

Closed without relief:
Precision: 0.45
Recall: 0.66
Accuracy: 0.86

Closed:
Precision: 0.51
Recall: 0.55
Accuracy: 0.88

Untimely response:
Precision: 0.51
Recall: 0.56
Accuracy: 0.88

Closed with relief:
Precision: 0.74
Recall: 0.95
Accuracy: 0.95
```

```
>>> from pyspark.sql.types import FloatType
>>> #important: need to cast to float type, and order by prediction, else it won't work
... preds_and_labels = predictions.select(['prediction', 'indexed_company_response'])\
...     .withColumn('indexed_company_response', col('indexed_company_response')\
...         .cast(FloatType()))\
...     .orderBy('prediction')
>>> metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))

[Stage 3350:=====] (187 + 5) / 1605]
/usr/odh/current/spark3-client/python/pyspark/sql/context.py:127: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
FutureWarning
>>>
>>> confusion_matrix = metrics.confusionMatrix().toArray()
>>>
>>> print(metrics.confusionMatrix().toArray())
[[3.730e+03 1.410e+02 2.000e+00 8.100e+01 6.200e+01 4.000e+00 4.630e+02
 0.000e+00]
 [2.710e+02 1.408e+03 1.023e+03 5.710e+02 3.990e+02 1.280e+02 5.030e+02
 3.190e+02]
 [3.900e+01 4.750e+02 2.231e+03 1.730e+02 8.700e+01 5.800e+01 1.520e+02
 1.234e+03]
 [1.290e+02 9.600e+01 5.600e+01 2.579e+03 4.690e+02 8.000e+02 3.660e+02
 0.000e+00]
 [1.000e+00 0.000e+00 0.000e+00 0.000e+00 2.957e+03 1.143e+03 3.780e+02
 0.000e+00]
 [0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.497e+03 2.473e+03 5.320e+02
 0.000e+00]
 [4.820e+02 3.300e+01 5.700e+01 2.800e+01 1.142e+03 2.420e+02 2.554e+03
 0.000e+00]
 [1.800e+01 8.800e+01 3.000e+00 7.800e+01 4.000e+00 7.000e+00 5.000e+01
 4.337e+03]]
```

```
>>> confusion_matrix = metrics.confusionMatrix().toArray()
>>>
>>> print(metrics.confusionMatrix().toArray())
[[3.730e+03 1.410e+02 2.000e+00 8.100e+01 6.200e+01 4.000e+00 4.630e+02
 0.000e+00]
 [2.710e+02 1.408e+03 1.023e+03 5.710e+02 3.990e+02 1.280e+02 5.030e+02
 3.190e+02]
 [3.900e+01 4.750e+02 2.231e+03 1.730e+02 8.700e+01 5.800e+01 1.520e+02
 1.234e+03]
 [1.290e+02 9.600e+01 5.600e+01 2.579e+03 4.690e+02 8.000e+02 3.660e+02
 0.000e+00]
 [1.000e+00 0.000e+00 0.000e+00 0.000e+00 2.957e+03 1.143e+03 3.780e+02
 0.000e+00]
 [0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.497e+03 2.473e+03 5.320e+02
 0.000e+00]
 [4.820e+02 3.300e+01 5.700e+01 2.800e+01 1.142e+03 2.420e+02 2.554e+03
 0.000e+00]
 [1.800e+01 8.800e+01 3.000e+00 7.800e+01 4.000e+00 7.000e+00 5.000e+01
 4.337e+03]]
```

```
>>> # Print the 2D list
... print(confusion_matrix_2d)
[[3730.0, 141.0, 2.0, 81.0, 62.0, 4.0, 463.0, 0.0], [271.0, 1408.0, 1023.0, 571.0, 399.0, 128.0, 503.0, 319.0], [39.0, 475.0, 2231.0, 173.0, 87.0, 58.0, 152.0, 1234.0], [129.0, 960.0, 560.0, 2579.0, 469.0, 800.0, 366.0, 0.0], [1.0, 0.0, 0.0, 0.0, 2957.0, 1143.0, 378.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1497.0, 2473.0, 532.0, 0.0], [482.0, 33.0, 57.0, 28.0, 1142.0, 242.0, 2554.0, 0.0], [18.0, 88.0, 3.0, 78.0, 4.0, 7.0, 50.0, 4337.0]]
```

Confusion Matrix:

To visualize confusion matrix for a Decision Tree model's cross-validation and train-validation results using Python. We import pandas, numpy, seaborn, and matplotlib in google colab as pyspark does not support visualization. Confusion matrix data is defined as 2D arrays (result retrieved from pyspark) for both cross-validation and train-validation results, then converted into pandas DataFrames. Using seaborn, we plot these matrices as heatmaps, setting figure size, annotations, color map, labels, and titles. This visualization provides a clear view of the model's classification performance.

Here is the code to perform confusion Matrix for Decision Tree:

```
"""Desicion tree_Matrix.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1Jc9Er4hvooPOq9oGVUgZbCjjRCP4pZpA

"""

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

# Sample data as a 2D array

confusion_matrix_dt_data= [[3.730e+03, 1.410e+02, 2.000e+00, 8.100e+01,
6.200e+01,4.000e+00, 4.630e+02,

0.000e+00],

[2.710e+02, 1.402e+03, 1.024e+03, 5.710e+02, 3.990e+02, 1.280e+02,
5.030e+0,

3.240e+02],

[3.900e+01, 4.750e+02, 2.231e+03, 1.730e+02 ,8.700e+01, 5.800e+01,
1.520e+02,

1.234e+03],
```

```

[1.290e+02, 9.600e+01, 5.600e+01, 2.579e+03, 4.690e+02
,8.000e+02,3.660e+02,

0.000e+00],

[1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 2.957e+03, 1.143e+03,
3.780e+02,

0.000e+00],

[0.000e+00, 0.000e+00 ,0.000e+00, 0.000e+00, 1.497e+03, 2.473e+03,
5.320e+02,

0.000e+00],

[4.840e+02, 3.300e+01, 5.700e+01, 2.800e+01, 1.142e+03 ,2.420e+02,
2.552e+03,

0.000e+00],

[1.800e+01, 8.800e+01 ,3.000e+00 ,7.800e+01, 4.000e+00,7.000e+00
,5.000e+01,

4.337e+03]]

# Convert to DataFrame

confusion_matrix_dt_cv = pd.DataFrame(confusion_matrix_dt_data)

# Set index and column names

confusion_matrix_dt_cv.index.name = 'Actual'

confusion_matrix_dt_cv.columns.name = 'Predicted'

confusion_matrix_dt_cv = pd.DataFrame(confusion_matrix_dt_data,
index=["1", "2", "3", "4", "5" , "6", "7", "8"], columns=["1", "2",
"3", "4", "5" , "6", "7", "8"])

```

```

# Plot heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(confusion_matrix_dt_cv, annot=True, fmt=".0f",
cmap="YlGnBu")

plt.xlabel("Predicted")

plt.ylabel("True Label")

plt.title(" Desicion Tree Cross Validation Confusion Matrix")

plt.show()


# Sample data as a 2D array

confusion_matrix_dt_tv_data= [[3.730e+03, 1.410e+02, 2.000e+00,
8.100e+01 ,6.200e+01, 4.000e+00, 4.630e+02,

    0.000e+00],

    [2.710e+02, 1.408e+03, 1.023e+03, 5.710e+02, 3.990e+02, 1.280e+02
,5.030e+02,

    3.190e+02],

    [3.900e+01, 4.750e+02, 2.231e+03, 1.730e+02 ,8.700e+01, 5.800e+01,
1.520e+02,

    1.234e+03],

    [1.290e+02, 9.600e+01 ,5.600e+01, 2.579e+03 ,4.690e+02, 8.000e+02
,3.660e+02,

    0.000e+00],

```

```
[1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 2.957e+03, 1.143e+03,
3.780e+02,

0.000e+00],

[0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 1.497e+03, 2.473e+03,
5.320e+02,

0.000e+00],

[4.820e+02 ,3.300e+01, 5.700e+01, 2.800e+01, 1.142e+03, 2.420e+02,
2.554e+03,

0.000e+00],

[1.800e+01, 8.800e+01, 3.000e+00, 7.800e+01, 4.000e+00, 7.000e+00,
5.000e+01,

4.337e+03]]
```

```
# Convert to DataFrame
```

```
confusion_matrix_dt_tv = pd.DataFrame(confusion_matrix_dt_tv_data)
```

```
# Set index and column names
```

```
confusion_matrix_dt_tv.index.name = 'Actual'
```

```
confusion_matrix_dt_tv.columns.name = 'Predicted'
```

```
confusion_matrix_dt_tv = pd.DataFrame(confusion_matrix_dt_tv_data,
index=["1", "2", "3", "4", "5" , "6", "7", "8"], columns=["1", "2",
"3", "4", "5" , "6", "7", "8"])
```

```
# Plot heatmap
```

```
plt.figure(figsize=(8, 6))
```

```

sns.heatmap(confusion_matrix_dt_tv, annot=True, fmt=".0f",
cmap="YlGnBu")

plt.xlabel("Predicted")

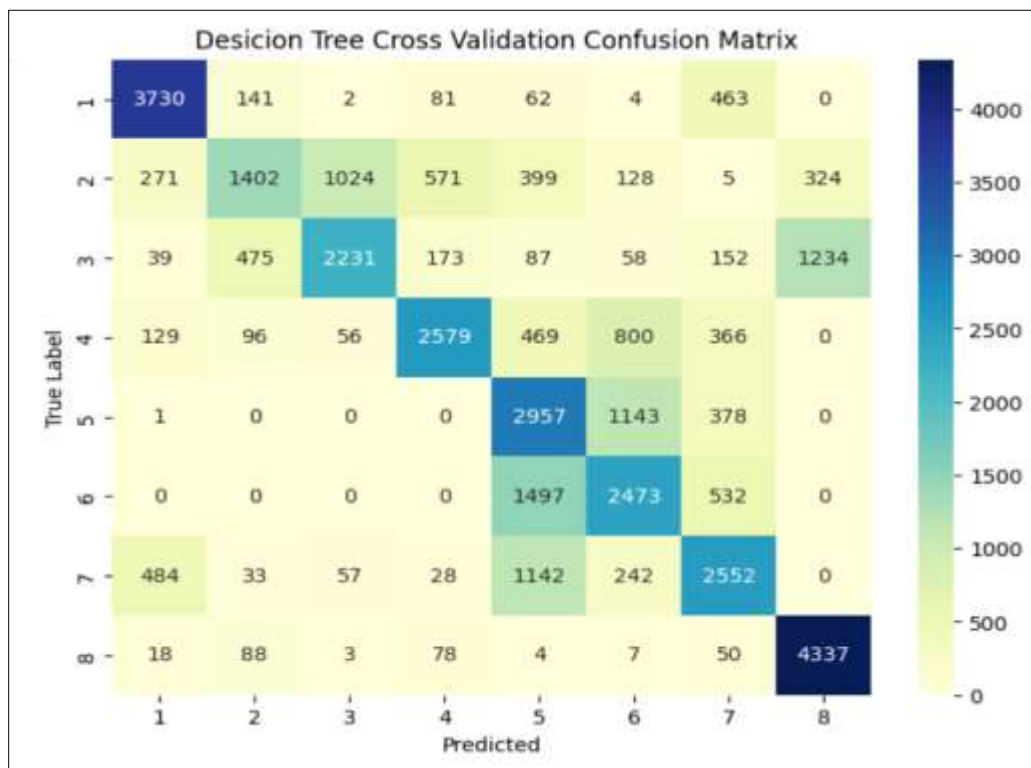
plt.ylabel("True Label")

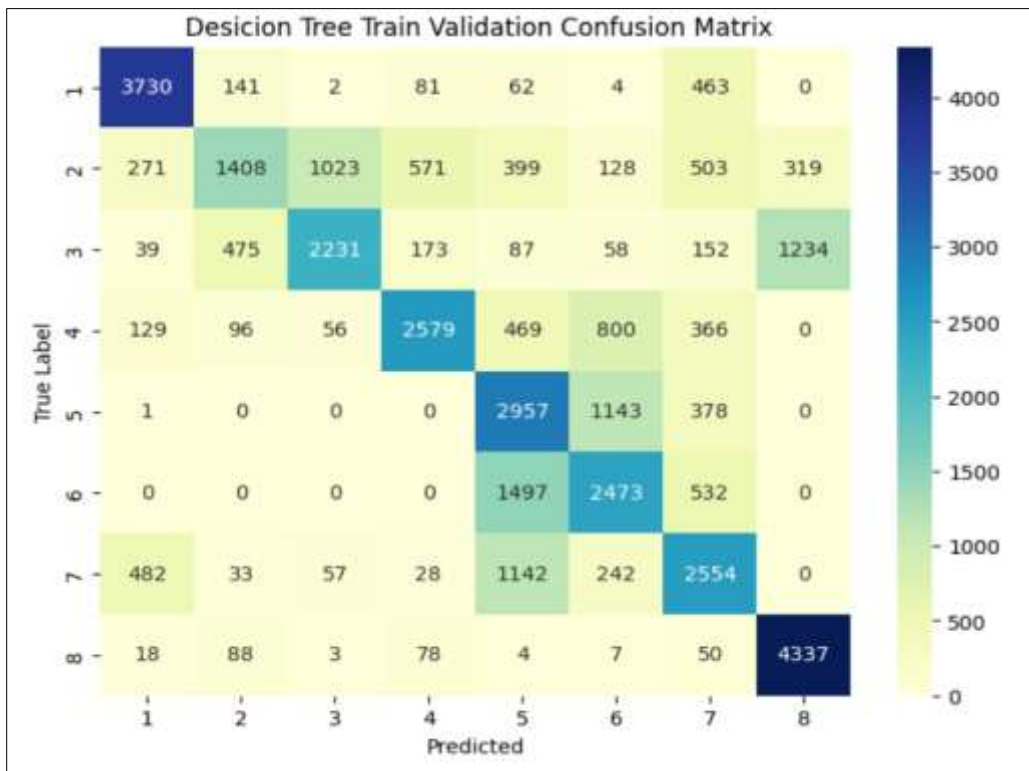
plt.title(" Desicion Tree Train Validation Confusion Matrix")

plt.show()

```

The Result:





B. Company Response using Random Forest:

This Spark code demonstrates how to perform classification on a complaints dataset using a Random Forest model. The model is trained with a CrossValidator for hyperparameter tuning. The code calculates feature importances and evaluates model performance using various metrics such as accuracy, precision, and recall. Additionally, it compares the results obtained from CrossValidator and TrainValidationSplit approaches.

```
"""This Spark code performs classification on a complaints dataset
using a Random Forest model trained with CrossValidator for
hyperparameter tuning.
```

```
Data Loading and Preprocessing:
```

```
Feature Engineering:
```

```
Feature Selection and Preprocessing:
```

```
Model Building:"""
```

```
import pandas as pd
```

```
from pyspark.sql.types import *
```

```
from pyspark.sql.functions import *
```

```
from pyspark.ml import Pipeline
```



```

from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.storagelevel import StorageLevel
from pyspark.sql.functions import lit
from pyspark.ml import Transformer
from pyspark.sql.functions import col
from pyspark.ml.tuning import TrainValidationSplit
from pyspark.ml.classification import RandomForestClassifier
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession

# PYSPARK_CLI = True
# if PYSPARK_CLI:
#     # sc = SparkContext.getOrCreate()
#     # spark = SparkSession(sc)

```

- **Data Preprocessing:**

We begin by reading the JSON file '**complaints.json**' into a DataFrame named **raw_complaints**. This file contains customer complaint data, which will be used for further analysis. We select the necessary columns: **company**, **product**, **company_response**, and **issue**, while filtering out any corrupt records based on the **_corrupt_record** column. Next, we clean the data by removing rows with missing or empty values in the **company**, **product**, or **company_response** columns. This ensures that the dataset is clean and free of incomplete records, providing a solid foundation for subsequent analysis and modeling. The cleaned dataset is stored in a DataFrame named **complaint_df**.

```

#Data Loading and Preprocessing:

#-----

# Read the JSON file 'complaints.json' into a DataFrame named
'raw_complaints'

```

```

raw_complaints =
spark.read.json('/user/dvaishn2/5560_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records

complaint_df = raw_complaints.select('company', 'product',
'company_response' ,
'issue').filter(raw_complaints['_corrupt_record'].isNull())

complaint_df = complaint_df.filter(~(isNull(col("company")) |
(trim(col("company")) == "")))

complaint_df = complaint_df.filter(~(isNull(col("product")) |
(trim(col("product")) == "")))

complaint_df = complaint_df.filter(~(isNull(col("company_response")) |
(trim(col("company_response")) == "")))

# Load dataset (assuming `complaint_df` is already defined)

df_company_response = complaint_df

```

```

>>> # Read the JSON file 'complaints.json' into a DataFrame named 'raw_complaints'
... raw_complaints = spark.read.json('/user/dvaishn2/5560_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records
complaint_df = raw_complaints.select('company', 'product', 'company_response', 'issue').filter(raw_complaints['_corrupt_record'].isNull())

complaint_df = complaint_df.filter(~(isNull(col("company")) | (trim(col("company")) == "")))
complaint_df = complaint_df.filter(~(isNull(col("product")) | (trim(col("product")) == "")))
complaint_df = complaint_df.filter(~(isNull(col("company_response")) | (trim(col("company_response")) == "")))

# Load dataset (assuming complaint_df is already defined)
df_company_response = complaint_df

>>> # Select necessary columns and drop corrupt records
... complaint_df = raw_complaints.select('company', 'product', 'company_response', 'issue').filter(raw_complaints['_corrupt_record'].isNull())
>>>
>>> complaint_df = complaint_df.filter(~(isNull(col("company")) | (trim(col("company")) == "")))
>>> complaint_df = complaint_df.filter(~(isNull(col("product")) | (trim(col("product")) == "")))
>>> complaint_df = complaint_df.filter(~(isNull(col("company_response")) | (trim(col("company_response")) == "")))
>>>
>>> # Load dataset (assuming complaint_df is already defined)
... df_company_response = complaint_df

```

- **Feature Engineering:**

In the feature engineering step, we enhance our dataset by calculating the frequency of occurrences for specific features and integrating these as new columns. First, we calculate the frequency of each company by grouping the data by the **company** column and counting the number of records for each company. This frequency information is then joined with the original DataFrame on the **company** column. Similarly, we calculate the frequency of each issue by grouping the data by the **issue** column and counting the records.

This frequency data is also joined with the original DataFrame on the **issue** column. These additional frequency features provide the model with valuable context about the relative occurrence of companies and issues, which can improve the model's predictive performance.

```
#Feature Engineering:

#-----

# Calculate the frequency of each company

company_frequency =
df_company_response.groupBy("company").agg(count("*").alias("frequency_
company"))

# Join the frequency DataFrame with the original DataFrame on the
company column

df_response_with_frequency =
df_company_response.join(company_frequency, on="company", how="left")

# Calculate the frequency of each issue (corrected to avoid duplicate
calculation)

issue_frequency =
df_company_response.groupBy("issue").agg(count("*").alias("frequency_is
sue"))

# Join the issue frequency DataFrame with the existing DataFrame on the
issue column

df_response_with_frequency =
df_response_with_frequency.join(issue_frequency, on="issue",
how="left")
```

```

>>> # Select necessary columns and drop corrupt records
... complaint_df = raw_complaints.select('company', 'product', 'company_response', 'issue').filter(raw_complaints['_corrupt_record'].isNull())
>>>
>>> complaint_df = complaint_df.filter(~(isNull(col('company')) | (trim(col('company')) == '')))
>>> complaint_df = complaint_df.filter(~(isNull(col('product')) | (trim(col('product')) == '')))
>>> complaint_df = complaint_df.filter(~(isNull(col('company_response')) | (trim(col('company_response')) == '')))
>>>
>>> # Load dataset (assuming 'complaint_df' is already defined)
... df_company_response = complaint_df
>>> # Calculate the frequency of each company
... company_frequency = df_company_response.groupBy("company").agg(count("*").alias("frequency_company"))
>>>
>>> # Join the frequency DataFrame with the original DataFrame on the company column
>>>
>>> # Join the frequency DataFrame with the original DataFrame on the company column
... df_response_with_frequency = df_company_response.join(company_frequency, on="company", how="left")
>>> issue_frequency = df_company_response.groupBy("issue").agg(count("*").alias("frequency_issue"))
>>>
>>> # Calculate the frequency of each issue (corrected to avoid duplicate calculation)
... issue_frequency = df_company_response.groupBy("issue").agg(count("*").alias("frequency_issue"))
>>> # Join the issue frequency DataFrame with the existing DataFrame on the issue column
df_response_with_frequency = df_response_with_frequency.join(issue_frequency, on="issue", how="left")>>>
>>> # Join the issue frequency DataFrame with the existing DataFrame on the issue column
... df_response_with_frequency = df_response_with_frequency.join(issue_frequency, on="issue", how="left")
>>> # Use the frequency column as a feature for modeling
... features = ["product", "frequency_company", "frequency_issue"]
>>> target = "company_response"

```

- **Prepare Pipeline:**

In this step, we define the features and target variable for the model, perform string indexing, and create a vector assembler to combine the features into a single vector. We start by specifying **product**, **frequency_company**, and **frequency_issue** as the features and **company_response** as the target variable. We persist the DataFrame in memory to improve performance during processing. String indexing is applied to the **company_response** (target) and **product** columns to convert categorical values into numerical format using **StringIndexer**. Next, we use **VectorAssembler** to combine the indexed features into a single feature vector, which will be used as input for the model.

```

#Feature Selection and Preprocessing:

#-----

# Use the frequency column as a feature for modeling

features = ["product", "frequency_company", "frequency_issue"]

target = "company_response"

from pyspark.storagelevel import StorageLevel

df_response_with_frequency.persist(StorageLevel.MEMORY_ONLY)

# String indexing for target variable

```

```
target_indexer = StringIndexer(inputCol="company_response",
outputCol="indexed_company_response")

indexer_product = StringIndexer(inputCol="product",
outputCol="indexed_product", handleInvalid="skip")

df_response_with_frequency = df_response_with_frequency.drop('company'
, 'issue')

# Create VectorAssembler to combine the indexed product and hashed
company features

assembler = VectorAssembler(inputCols=["indexed_product",
"frequency_company", "frequency_issue"], outputCol="features")

# Create Random Forest model

rf = RandomForestClassifier(labelCol="indexed_company_response",
featuresCol="features")

#pipeline:

#-----

# Create a pipeline with the VectorAssembler and Random Forest model

pipeline = Pipeline(stages=[indexer_product, target_indexer, assembler,
rf])

# Split the data into training and testing sets

train_data, test_data = balanced_data.randomSplit([0.7, 0.3], seed=42)

train_rows = train_data.count()

test_rows = test_data.count()

# Print the counts
```

```
print("Training Rows:", train_rows, " Testing Rows:", test_rows)
```

```
>>>
... df_response_with_frequency.persist(storageLevel=MEMORY_ONLY)
# String indexing for target variable
target_indexer = StringIndexer(inputCol="company_response", outputCol="indexed_company_response")
indexer_product = StringIndexer(inputCol="product", outputCol="indexed_product", handleInvalid="skip")
df_response_with_frequency = df_response_with_frequency.drop('company', 'issue')

# Create VectorAssembler to combine the indexed product and hashed company features
assembler = VectorAssembler(inputCols=["indexed_product", "frequency_company", "frequency_issue"], outputCol="features")

# Create Random Forest model
rf = RandomForestClassifier(labelCol="indexed_company_response", featuresCol="features")Dataframe[issue: string, company: string, product: string, company_response: string,
frequency_company: bigint, frequency_issue: bigint]
>>>
... # String indexing for target variable
... target_indexer = StringIndexer(inputCol="company_response", outputCol="indexed_company_response")
...
... indexer_product = StringIndexer(inputCol="product", outputCol="indexed_product", handleInvalid="skip")
...
... df_response_with_frequency = df_response_with_frequency.drop('company', 'issue')
...
... # Create VectorAssembler to combine the indexed product and hashed company features
... assembler = VectorAssembler(inputCols=["indexed_product", "frequency_company", "frequency_issue"], outputCol="features")
...
... # Create Random Forest model
... rf = RandomForestClassifier(labelCol="indexed_company_response", featuresCol="features")
```

```
>>> # Create a pipeline with the VectorAssembler and Random Forest model
... pipeline = Pipeline(stages=[indexer_product, target_indexer, assembler, rf])
>>>
```

- **Balancing the Data by Oversampling Minority Class (company response 8 categories):**

To ensure balanced representation of each response type in the dataset, we perform several steps. First, we define separate DataFrames for each response type, such as "Closed with explanation", "Closed with non-monetary relief", "In progress", and others. We then count the number of complaints in each category. Using these counts, we calculate oversampling factors to achieve a target count of 15,000 samples per category. We compute the sampling fractions required to reach this target count for each response type. Next, we sample each category DataFrame using the calculated fractions, performing sampling with replacement to balance the dataset. The sampled DataFrames are then combined using the union operation to create a single balanced dataset. Finally, we display the count of each category in the balanced data and show the first 20 rows to confirm the balancing process.

```
# Balancing the data_set:

#-----

# Define DataFrames for each response type
```

```
closed_with_explanation =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with explanation")

closed_with_non_monetary_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with non-monetary relief")

in_progress =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "In progress")

closed_with_monetary_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with monetary relief")

closed_without_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed without relief")

closed =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed")

untimely_response =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Untimely response")

closed_with_relief =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == "Closed with relief")

# Calculate current counts for each response type

# Calculate current counts for each response type

counts = {
```

```

    "Closed with explanation": closed_with_explanation.count(),

    "Closed with non-monetary relief":
closed_with_non_monetary_relief.count(),

    "In progress": in_progress.count(),

    "Closed with monetary relief": closed_with_monetary_relief.count(),

    "Closed without relief": closed_without_relief.count(),

    "Closed": closed.count(),

    "Untimely response": untimely_response.count(),

    "Closed with relief": closed_with_relief.count()
}

# Calculate the oversampling factor for each category to achieve 15000
samples

target_count = 15000

oversampling_factors = {response: target_count / count for response,
count in counts.items()}

# Calculate sampling fractions to achieve the target count for each
category

sampling_fractions = {category: target_count / count for category,
count in counts.items()}

# Create an empty DataFrame to hold the balanced data

balanced_data = spark.createDataFrame([],
df_response_with_frequency.schema)

# Sample each category to achieve the target count

```



```

for category, count in counts.items():

# Sample the category DataFrame with the calculated fraction

sampled_df =
df_response_with_frequency.filter(df_response_with_frequency["company_r
esponse"] == category)\
.sample(withReplacement=True, fraction=sampling_fractions[category],
seed=42)

# Union the sampled DataFrame with the balanced data

balanced_data = balanced_data.union(sampled_df)

```

```

... rf = RandomForestClassifier(labelCol="indexed_company_response", featuresCol="features")
... # Define DataFrames for each response type
... closed_with_explanation = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with explanation")
... closed_with_non_monetary_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with non-monetary relief")
... in_progress = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "In progress")
... closed_with_monetary_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with monetary relief")
... closed_without_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed without relief")
... closed = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed")
... untimely_response = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Untimely response")
... closed_with_relief = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == "Closed with relief")
...
...
... # Calculate current counts for each response type
... # Calculate current counts for each response type
... counts = {
...     "Closed with explanation": closed_with_explanation.count(),
...     "Closed with non-monetary relief": closed_with_non_monetary_relief.count(),
...     "In progress": in_progress.count(),
...     "Closed with monetary relief": closed_with_monetary_relief.count(),
...     "Closed without relief": closed_without_relief.count(),
...     "Closed": closed.count(),
...     "Untimely response": untimely_response.count(),
...     "Closed with relief": closed_with_relief.count()
... }

# Calculate the oversampling factor for each category to achieve 15000 samples
target_count = 15000
oversampling_factors = {response: target_count / count for response, count in counts.items()}

# Calculate sampling fractions to achieve the target count for each category
sampling_fractions = {category: target_count / count for category, count in counts.items()}

# Create an empty DataFrame to hold the balanced data
balanced_data = spark.createDataFrame([], df_response_with_frequency.schema)

# Sample each category to achieve the target count
for category, count in counts.items():
    # Sample the category DataFrame with the calculated fraction
    sampled_df = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == category)\
        .sample(withReplacement=True, fraction=sampling_fractions[category], seed=42)
    # Union the sampled DataFrame with the balanced data
    balanced_data = balanced_data.union(sampled_df)
...
...
... # Calculate the oversampling factor for each category to achieve 15000 samples
... target_count = 15000
... oversampling_factors = {response: target_count / count for response, count in counts.items()}

```

```

# Calculate the oversampling factor for each category to achieve 15000 samples
target_count = 15000
oversampling_factors = {response: target_count / count for response, count in counts.items()}

# Calculate sampling fractions to achieve the target count for each category
sampling_fractions = {category: target_count / count for category, count in counts.items()}

# Create an empty DataFrame to hold the balanced data
balanced_data = spark.createDataFrame([], df_response_with_frequency.schema)

# Sample each category to achieve the target count
for category, count in counts.items():
    # Sample the category DataFrame with the calculated fraction
    sampled_df = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == category)\
        .sample(withReplacement=True, fraction=sampling_fractions[category], seed=42)
    # Union the sampled DataFrame with the balanced data
    balanced_data = balanced_data.union(sampled_df)
>>>
>>>
>>> # Calculate the oversampling factor for each category to achieve 15000 samples
... target_count = 15000
>>> oversampling_factors = {response: target_count / count for response, count in counts.items()}
>>>
>>> # Calculate sampling fractions to achieve the target count for each category
... sampling_fractions = {category: target_count / count for category, count in counts.items()}
>>>
>>> # Create an empty DataFrame to hold the balanced data
... balanced_data = spark.createDataFrame([], df_response_with_frequency.schema)
>>>
>>> # Sample each category to achieve the target count
... for category, count in counts.items():
...     # Sample the category DataFrame with the calculated fraction
...     sampled_df = df_response_with_frequency.filter(df_response_with_frequency["company_response"] == category)\
...         .sample(withReplacement=True, fraction=sampling_fractions[category], seed=42)
...     # Union the sampled DataFrame with the balanced data
...     balanced_data = balanced_data.union(sampled_df)
...

```

- **Model Training:**

To build the model, we start by creating a Random Forest classifier (**RandomForestClassifier**). We then define a pipeline that includes all the necessary feature preprocessing stages along with the Random Forest model. To optimize the model, we set up a **ParamGridBuilder** for hyperparameter tuning. The parameter grid includes different values for the number of trees (**numTrees**), the maximum depth of the trees (**maxDepth**), and the minimum instances per node (**minInstancesPerNode**). The model training begins by fitting the **CrossValidator** to the training data, which performs cross-validation to select the best model based on the specified hyperparameters. We measure the training time and format it for easy interpretation. The best model from the cross-validation process is then retrieved for further evaluation and use. This comprehensive approach ensures that the model is well-tuned and performs optimally on the given dataset.

```
#Model Building:
```

```
#-----
```

```
evaluator =
MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="indexed_company_response", metricName="accuracy")

paramGrid = ParamGridBuilder() \

    .addGrid(rf.numTrees, [10, 20]) \

    .addGrid(rf.maxDepth, [2, 4]) \

    .addGrid(rf.minInstancesPerNode, [5, 10]) \

    .build()

# Define CrossValidator

crossval = CrossValidator(estimator=pipeline,

                           estimatorParamMaps=paramGrid,

                           evaluator=evaluator,

                           numFolds=3)

from pyspark.sql.functions import col

#Training the model and Calculating its time

import time

# Start time

start_time = time.time()

# Fit the cross validator to the training data

cvModel = crossval.fit(train_data)

# End time
```

```

end_time = time.time()

print("Model trained!")

# Calculate training time

training_time = end_time - start_time

# Calculate minutes and seconds

minutes = int(training_time // 60)

seconds = int(training_time % 60)

# Format the time

training_time_formatted_cv = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time

print("Training time crossvalidation:", training_time_formatted_cv)

# Get the fitted model from CrossValidator

bestModel = cvModel.bestModel

```

```

>>> # Split the data into training and testing sets
... train_data, test_data = balanced_data.randomSplit([0.7, 0.3], seed=42)

train_rows = train_data.count()
>>>
>>> train_rows = train_data.count()
test_rows = test_data.count()

# Print the counts
print("Training Rows:", train_rows, " Testing Rows:", test_rows)
>>> test_rows = test_data.count()
>>>
>>> # Print the counts
... print("Training Rows:", train_rows, " Testing Rows:", test_rows)
Training Rows: 83763 Testing Rows: 36153

```

```
>>> evaluator = MulticlassClassificationEvaluator(predictionCol="prediction", labelCol="indexed_company_response", metricName="accuracy")
>>>
... paramGrid = ParamGridBuilder() \
...     .addGrid(rf.numTrees, [10, 20]) \
...     .addGrid(rf.maxDepth, [2, 4]) \
...     .addGrid(rf.minInstancesPerNode, [5, 10]) \
...     .build()
>>> # Define CrossValidator
... crossval = CrossValidator(estimator=pipeline,
...                             estimatorParamMaps=paramGrid,
...                             evaluator=evaluator,
...                             numFolds=3)
>>> from pyspark.sql.functions import col
>>>
```

```
... #Training the model and calculating its time
... import time
>>>
>>> # Start time
... start_time = time.time()
>>>
>>> # Fit the cross validator to the training data
... cvModel = crossval.fit(train_data)
# End time
end_time = time.time()

print("Model trained!")

# Calculate training time
training_time = end_time - start_time

# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

# Format the time
training_time_formatted_cv = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time
print("Training time crossvalidation:", training_time_formatted_cv)
>>>
>>> # End time
... end_time = time.time()
>>>
>>> print("Model trained!")
Model trained!
>>>
>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>>
>>> # Format the time
... training_time_formatted_cv = "{:02d}:{:02d}".format(minutes, seconds)
>>>
>>> # Print training time
... print("Training time crossvalidation:", training_time_formatted_cv)
Training time crossvalidation: 26:52
>>> |
```

Training Time: The training process takes approximately 26.52 minutes.

- **Feature Importance – Cross Validation:**

After training the model, we assess the importance of each feature to understand their impact on the predictions. We access the feature importances from the Random Forest model, which is the final stage in

the pipeline. The VectorAssembler used in the pipeline provides the feature names. We create a DataFrame that pairs each feature with its corresponding importance value. This DataFrame is then sorted in descending order of importance to highlight the most influential features. Finally, we print the DataFrame to present the feature importances clearly.

```
# Feature Importance crossvalidation:

#-----

# Access the feature importances from the Random Forest model within
the pipeline

feature_importances = bestModel.stages[-1].featureImportances

# Assuming Random Forest is the last stage

# Get feature names from the VectorAssembler

feature_names = assembler.getInputCols()

# Create a DataFrame of feature importances

featureImp = pd.DataFrame(list(zip(feature_names,
feature_importances)), columns=["feature", "importance"])

# Sort the DataFrame by importance (descending order)

featureImp = featureImp.sort_values(by="importance", ascending=False)

# Print the DataFrame with feature importance

print("\nFeature Importance_crossvalidation:")

print(featureImp.round(2).to_string(index=False))
```

- frequency_company: 0.49
- product: 0.39
- frequency_issue: 0.12

```
>>> # Get the fitted model from CrossValidator
... bestModel = cvModel.bestModel
>>> # Access the feature importances from the Random Forest model within the pipeline
... feature_importances = bestModel.stages[-1].featureImportances # Assuming Random Forest is the last stage
>>>
>>> # Get feature names from the VectorAssembler
... feature_names = assembler.getInputCols()
>>>
>>> # Create a DataFrame of feature importances
... featureImp = pd.DataFrame(list(zip(feature_names, feature_importances)), columns=["feature", "importance"])
>>>
>>> # Sort the DataFrame by importance (descending order)
... featureImp = featureImp.sort_values(by="importance", ascending=False)
>>>
>>> # Print the DataFrame with feature importance
... print("\nFeature Importance_crossvalidation:")

Feature Importance_crossvalidation:
>>> print(featureImp.round(2).to_string(index=False))
   feature  importance
frequency_company    0.49
indexed_product      0.39
frequency_issue      0.12
```

- **Confusion Matrix: Cross Validation (Test the data):**

To evaluate the performance of the best model on unseen data, we make predictions on the test dataset using the trained model. First, we transform the test data with the best model obtained from cross-validation. The predicted labels and true labels are selected and cast to float type, then ordered by prediction to ensure compatibility with evaluation metrics. We use MulticlassMetrics to evaluate the model's performance, which includes generating a confusion matrix. The confusion matrix provides detailed insights into the classification performance across different classes. We calculate precision, recall, and accuracy for each response category to understand the model's effectiveness. The results are printed, showing the performance metrics for each class, such as "Closed with explanation" and "Closed with non-monetary relief". This comprehensive evaluation helps in assessing the model's reliability and areas for improvement.

```
#Test the Data :

#-----

# Make predictions on the test data using the best model

predictions = cvModel.transform(test_data)

from pyspark.sql.types import FloatType
```

```
#important: need to cast to float type, and order by prediction, else
it won't work

preds_and_labels =
predictions.select(['prediction','indexed_company_response'])\

                .withColumn('indexed_company_response',
col('indexed_company_response')\

                .cast(FloatType()))\

                .orderBy('prediction')

from pyspark.mllib.evaluation import MulticlassMetrics

metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))

confusion_matrix = metrics.confusionMatrix().toArray()

print("***Confusion matrix_crossvalidation***")

print(metrics.confusionMatrix().toArray())

import numpy as np

tps = np.diag(confusion_matrix)

fps = np.sum(confusion_matrix, axis=0) - tps

fns = np.sum(confusion_matrix, axis=1) - tps

tns = np.sum(confusion_matrix) - tps - fps - fns

# Calculate precision, recall, and accuracy for each class

precision = tps / (tps + fps)

recall = tps / (tps + fns)
```



```
accuracy = (tps + tns) / np.sum(confusion_matrix)

"""Class 1: "Closed with explanation"

Class 2: "Closed with non-monetary relief"

Class 3: "In progress"

Class 4: "Closed with monetary relief"

Class 5: "Closed without relief"

Class 6: "Closed"

Class 7: "Untimely response"

Class 8: "Closed with relief" """

class_names = {

    1: "Closed with explanation",

    2: "Closed with non-monetary relief",

    3: "In progress",

    4: "Closed with monetary relief",

    5: "Closed without relief",

    6: "Closed",

    7: "Untimely response",

    8: "Closed with relief"

}

for i in range(len(precision)):
```

```

class_name = class_names[i + 1]

print(f"{class_name}:")

print(f"Precision: {precision[i]:.2f}")

print(f"Recall: {recall[i]:.2f}")

print(f"Accuracy: {accuracy[i]:.2f}")

print()

confusion_matrix_2d = confusion_matrix.tolist()

# Print the 2D list

print(confusion_matrix_2d)

```

```

>>> #important: need to cast to float type, and order by prediction, else it won't work
... preds_and_labels = predictions.select(['prediction', 'indexed_company_response'])\
...     .withColumn('indexed_company_response', col('indexed_company_response'))\
...     .cast(FloatType())\
...     .orderBy("prediction")
metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))

confusion_matrix = metrics.confusionMatrix().toArray()
>>>
>>> from pyspark.mllib.evaluation import MulticlassMetrics
>>> metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))

print("***Confusion matrix_crossvalidation***")

[Stage 1915:=> (66 + 5) / 1605]
/usr/odh/current/spark3-client/python/pyspark/sql/context.py:127: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  FutureWarning
>>>
>>> confusion_matrix = metrics.confusionMatrix().toArray()
>>>
>>> print("***Confusion matrix_crossvalidation***")
***Confusion matrix_crossvalidation***
>>>
>>> print(metrics.confusionMatrix().toArray())
[[3.9854e+03 8.400e+01 4.000e+00 3.500e+01 7.900e+01 1.300e+01 2.790e+02
 1.000e+00]
 [3.690e+02 1.323e+03 9.790e+02 5.380e+02 4.800e+02 3.130e+02 3.110e+02
 9.090e+02]
 [5.400e+01 4.820e+02 2.200e+03 1.610e+02 1.390e+02 1.180e+02 7.000e+01
 1.221e+03]
 [1.900e+02 8.600e+01 4.100e+01 2.406e+03 2.360e+02 1.361e+03 1.670e+02
 8.000e+00]
 [1.100e+01 0.000e+00 0.000e+00 0.000e+00 2.181e+03 2.048e+03 2.390e+02
 0.000e+00]
 [1.000e+01 0.000e+00 0.000e+00 0.000e+00 8.540e+02 3.398e+03 2.400e+02
 0.000e+00]
 [7.970e+02 1.500e+01 5.000e+01 1.400e+01 1.822e+03 6.940e+02 1.626e+03
 0.000e+00]
 [2.700e+01 5.100e+01 7.600e+01 4.700e+01 3.900e+01 5.900e+01 9.000e+00
 4.277e+03]]

>>> confusion_matrix_2d = confusion_matrix.tolist()
>>>
>>> # Print the 2D list
... print(confusion_matrix_2d)
[[3985.0, 84.0, 4.0, 35.0, 79.0, 13.0, 279.0, 1.0], [369.0, 1323.0, 979.0, 538.0, 480.0, 313.0, 311.0, 909.0], [54.0, 482.0, 2200.0, 161.0, 139.0, 118.0, 70.0, 1225.0], [190.0, 86.0, 41.0, 2406.0, 236.0, 1361.0, 167.0, 8.0], [11.0, 0.0, 0.0, 0.0, 2181.0, 2048.0, 239.0, 0.0], [10.0, 0.0, 0.0, 0.0, 854.0, 3398.0, 240.0, 0.0], [797.0, 15.0, 50.0, 34.0, 1322.0, 694.0, 1626.0, 0.0], [27.0, 51.0, 76.0, 47.0, 39.0, 59.0, 9.0, 4277.0]]

```

- **Model Evaluation – Cross Validator:**

The model's performance is evaluated using cross-validation, focusing on precision, recall, and accuracy for each response category. The results indicate how well the model can predict each category, with precision and recall values providing insight into the model's accuracy and sensitivity.

```

... tps = np.diag(confusion_matrix)
... fps = np.sum(confusion_matrix, axis=0) - tps
... fns = np.sum(confusion_matrix, axis=1) - tps
... tns = np.sum(confusion_matrix) - tps - fps - fns
...
... # Calculate precision, recall, and accuracy for each class
... precision = tps / (tps + fps)
... recall = tps / (tps + fns)
... accuracy = (tps + tns) / np.sum(confusion_matrix)
...
... """Class 1: "Closed with explanation"
... Class 2: "Closed with non-monetary relief"
... Class 3: "In progress"
... Class 4: "Closed with monetary relief"
... Class 5: "Closed without relief"
... Class 6: "Closed"
... Class 7: "Untimely response"
... Class 8: "Closed with relief" """
...
... "Class 1: "Closed with explanation"\nClass 2: "Closed with non-monetary relief"\nClass 3: "In progress"\nClass 4: "Closed with monetary relief"\nClass 5: "Closed without relief"\nClass 6: "Closed"\nClass 7: "Untimely response"\nClass 8: "Closed with relief"
...
... class_names = {
...     1: "Closed with explanation",
...     2: "Closed with non-monetary relief",
...     3: "In progress",
...     4: "Closed with monetary relief",
...     5: "Closed without relief",
...     6: "Closed",
...     7: "Untimely response",
...     8: "Closed with relief"
... }
...
... for i in range(len(precision)):
...     class_name = class_names[i + 1]
...     print(f"{class_name}:")
...     print(f"Precision: {precision[i]:.2f}")
...     print(f"Recall: {recall[i]:.2f}")
...     print(f"Accuracy: {accuracy[i]:.2f}")
...     print()

```

- **Model Evaluation – Cross Validator Results:**

The detailed evaluation metrics for each response category are as follows:

- **Closed with explanation:** Precision: 0.73, Recall: 0.89, Accuracy: 0.95
- **Closed with non-monetary relief:** Precision: 0.65, Recall: 0.29, Accuracy: 0.89
- **In progress:** Precision: 0.66, Recall: 0.49, Accuracy: 0.91
- **Closed with monetary relief:** Precision: 0.75, Recall: 0.54, Accuracy: 0.92
- **Closed without relief:** Precision: 0.41, Recall: 0.49, Accuracy: 0.85
- **Closed:** Precision: 0.42, Recall: 0.75, Accuracy: 0.84
- **Untimely response:** Precision: 0.55, Recall: 0.36, Accuracy: 0.88
- **Closed with relief:** Precision: 0.73, Recall: 0.93, Accuracy: 0.95

```
Closed with explanation:
Precision: 0.73
Recall: 0.89
Accuracy: 0.95

Closed with non-monetary relief:
Precision: 0.65
Recall: 0.29
Accuracy: 0.89

In progress:
Precision: 0.66
Recall: 0.49
Accuracy: 0.91

Closed with monetary relief:
Precision: 0.75
Recall: 0.54
Accuracy: 0.92

Closed without relief:
Precision: 0.41
Recall: 0.49
Accuracy: 0.85

Closed:
Precision: 0.42
Recall: 0.75
Accuracy: 0.84

Untimely response:
Precision: 0.55
Recall: 0.36
Accuracy: 0.88

Closed with relief:
Precision: 0.73
Recall: 0.93
Accuracy: 0.95
```

- **Train Validation Split:**

To validate the model's performance and ensure it generalizes well to new data, we use the Train-Validation Split method. We set up a **TrainValidationSplit** with the previously defined pipeline, parameter grid, and evaluator. The training ratio is set to 0.8, meaning 80% of the data is used for training and 20% for validation. After training, we make predictions on the test data using the best model obtained from the Train-Validation Split. We also evaluate feature importances to understand which features contributed most to the model's predictions.

```
#Train Validation
```

```
#-----

from pyspark.ml.tuning import TrainValidationSplit

from pyspark.sql.functions import col

# Define TrainValidationSplit

trainval = TrainValidationSplit(estimator=pipeline,

                                estimatorParamMaps=paramGrid,

                                evaluator=evaluator,

                                trainRatio=0.8)


#Training the model and Calculating its time

import time

# Start time

start_time = time.time()

# Fit the cross validator to the training data

tvModel = trainval.fit(train_data)

# End time

end_time = time.time()

print("Model trained!")
```

```
# Calculate training time

training_time = end_time - start_time

# Calculate minutes and seconds

minutes = int(training_time // 60)

seconds = int(training_time % 60)

# Format the time

training_time_formatted_tv = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time

print("Training time_tv:", training_time_formatted_tv)


#Make predictions on the test data using the best model

predictions = tvModel.transform(test_data)


# Get the fitted model from CrossValidator

bestModel = tvModel.bestModel
```

```

>>> # Start time
... start_time = time.time()
>>>
>>> # Fit the cross validator to the training data
... tvModel = trainval.fit(train_data)

# End time
end_time = time.time()

print("Model trained!")

# Calculate training time
training_time = end_time - start_time

# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

# Format the time
training_time_formatted_tv = "{:02d}:{:02d}".format(minutes, seconds)

# Print training time
print("Training time_tv:", training_time_formatted_tv)
>>>
>>> # End time
... end_time = time.time()
>>>
>>> print("Model trained!")
Model trained!
>>>
>>>
>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>>
>>> # Format the time
... training_time_formatted_tv = "{:02d}:{:02d}".format(minutes, seconds)
>>>
>>> # Print training time
... print("Training time_tv:", training_time_formatted_tv)
Training time_tv: 12:17
>>> |

```

Training Time: The training process with the train-validation split takes approximately 12.17 minutes.

- **Feature Importance – Train Validation Split:**

After training, we make predictions on the test data using the best model obtained from the Train-Validation Split. We also evaluate feature importances to understand which features contributed most to the model's predictions. We access the feature importances from the Random Forest model within the pipeline. The **VectorAssembler** provides the feature names, and we create a DataFrame to pair each feature with its importance value. This DataFrame is sorted in descending order of importance to highlight the most influential features.

```
#Feature Importance:trainvalidation
```

```
#-----
```

```

# Access the feature importances from the Random Forest model within
the pipeline

feature_importances_tv = bestModel.stages[-1].featureImportances #
Assuming Random Forest is the last stage

# Get feature names from the VectorAssembler

feature_names_tv = assembler.getInputCols()

# Create a DataFrame of feature importances

featureImp_tv = pd.DataFrame(list(zip(feature_names_tv,
feature_importances_tv)), columns=["feature", "importance"])

# Sort the DataFrame by importance (descending order)

featureImp_tv = featureImp_tv.sort_values(by="importance",
ascending=False)

# Print the DataFrame with feature importance

print("\nFeature Importance_transvalidation:")

print(featureImp_tv.round(2).to_string(index=False))

```

- **frequency_company:** 0.53
- **product:** 0.37
- **frequency_issue:** 0.10


```

>>> #Make predictions on the test data using the best model
... predictions = tvModel.transform(test_data)

# Get the fitted model from CrossValidator
bestModel = tvModel.bestModel

>>>
>>> # Get the fitted model from CrossValidator
... bestModel = tvModel.bestModel
>>>
>>> #Feature Importance:trainvalidation
... #-----
...
>>>
>>> # Access the Feature importances from the Random Forest model within the pipeline
... feature_importances_tv = bestModel.stages[-1].featureImportances # Assuming Random Forest is the last stage
>>>
>>> # Get feature names from the VectorAssembler
... feature_names_tv = assembler.getInputCols()
>>>
>>> # Create a DataFrame of feature importances
... featureImp_tv = pd.DataFrame(list(zip(feature_names_tv, feature_importances_tv)), columns=["feature", "importance"])
>>>
>>> # Sort the DataFrame by importance (descending order)
... featureImp_tv = featureImp_tv.sort_values(by="importance", ascending=False)
>>>
>>> # Print the DataFrame with feature importance
... print("\nFeature Importance_transvalidation:")

Feature Importance_transvalidation:
>>> print(featureImp_tv.round(2).to_string(index=False))
      feature  importance
frequency_company    0.53
indexed_product      0.37
frequency_issue      0.10
>>>

```

- **Model Evaluation – Train Validation Split:**

After training the model using the Train-Validation Split method, we evaluate its performance on the test data. We first make predictions using the best model and prepare the predicted and actual labels for evaluation by casting them to float type and ordering them by prediction. This setup is essential for accurate computation of performance metrics. The confusion matrix is printed to provide a detailed view of the classification results, showing the counts of true positives, false positives, true negatives, and false negatives for each class.

We then calculate key performance metrics—precision, recall, and accuracy—for each response category. These metrics help us understand the model's effectiveness in classifying each type of company response. To provide a clear understanding, we map the numerical labels to the corresponding response category names and print the performance metrics for each class.

```

#Result:TrainValidationSplit

#-----

from pyspark.sql.types import FloatType

#important: need to cast to float type, and order by prediction, else
it won't work

```

```

preds_and_labels =
predictions.select(['prediction', 'indexed_company_response'])\

                                .withColumn('indexed_company_response',
col('indexed_company_response')\

                                .cast(FloatType()))\

                                .orderBy('prediction')

metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))

confusion_matrix = metrics.confusionMatrix().toArray()

print("***Confusion matrix_transvalidation***")

print(metrics.confusionMatrix().toArray())

import numpy as np

tps = np.diag(confusion_matrix)

fps = np.sum(confusion_matrix, axis=0) - tps

fns = np.sum(confusion_matrix, axis=1) - tps

tns = np.sum(confusion_matrix) - tps - fps - fns

# Calculate precision, recall, and accuracy for each class

precision = tps / (tps + fps)

recall = tps / (tps + fns)

accuracy = (tps + tns) / np.sum(confusion_matrix)

"""Class 1: "Closed with explanation"

Class 2: "Closed with non-monetary relief"

```

```
Class 3: "In progress"

Class 4: "Closed with monetary relief"

Class 5: "Closed without relief"

Class 6: "Closed"

Class 7: "Untimely response"

Class 8: "Closed with relief" ""

class_names = {

    1: "Closed with explanation",

    2: "Closed with non-monetary relief",

    3: "In progress",

    4: "Closed with monetary relief",

    5: "Closed without relief",

    6: "Closed",

    7: "Untimely response",

    8: "Closed with relief"

}

for i in range(len(precision)):

    class_name = class_names[i + 1]

    print(f"{class_name}:")

    print(f"Precision: {precision[i]:.2f}")
```

```

    print(f"Recall: {recall[i]:.2f}")

    print(f"Accuracy: {accuracy[i]:.2f}")

    print()

confusion_matrix_2d = confusion_matrix.tolist()

# Print the 2D list

print(confusion_matrix_2d)

```

```

>>> #Train Validation
... #-----
...
>>> from pyspark.ml.tuning import TrainValidationSplit
>>> from pyspark.sql.functions import col
>>>
>>>
>>> # Define TrainValidationSplit
... trainval = TrainValidationSplit(estimator=pipeline,
...                                 estimatorParamMaps=paramGrid,
...                                 evaluator=evaluator,
...                                 trainRatio=0.8)
>>>
...
>>>
... #Training the model and Calculating its time
... import time
>>>
>>> # Start time
... start_time = time.time()
>>>
>>> # Fit the cross validator to the training data
... tvModel = trainval.fit(train_data)

```

- **Model Evaluation – Train Validation Split Results:**

The detailed evaluation metrics for each response category are as follows:

- **Closed with explanation:** Precision: 0.73, Recall: 0.89, Accuracy: 0.95
- **Closed with non-monetary relief:** Precision: 0.65, Recall: 0.29, Accuracy: 0.89
- **In progress:** Precision: 0.66, Recall: 0.49, Accuracy: 0.91
- **Closed with monetary relief:** Precision: 0.75, Recall: 0.54, Accuracy: 0.92
- **Closed without relief:** Precision: 0.41, Recall: 0.49, Accuracy: 0.85
- **Closed:** Precision: 0.42, Recall: 0.75, Accuracy: 0.84

- **Untimely response:** Precision: 0.55, Recall: 0.36, Accuracy: 0.88
- **Closed with relief:** Precision: 0.73, Recall: 0.93, Accuracy: 0.95

```
... tps = np.diag(confusion_matrix)
>>> fps = np.sum(confusion_matrix, axis=0) - tps
>>> fns = np.sum(confusion_matrix, axis=1) - tps
>>> tns = np.sum(confusion_matrix) - tps - fps - fns
>>>
>>> # Calculate precision, recall, and accuracy for each class
... precision = tps / (tps + fps)
>>> recall = tps / (tps + fns)
>>> accuracy = (tps + tns) / np.sum(confusion_matrix)
>>>
>>> """Class 1: "Closed with explanation"
... Class 2: "Closed with non-monetary relief"
... Class 3: "In progress"
... Class 4: "Closed with monetary relief"
... Class 5: "Closed without relief"
... Class 6: "Closed"
... Class 7: "Untimely response"
... Class 8: "Closed with relief" """
>>> "Class 1: "Closed with explanation"\nClass 2: "Closed with non-monetary relief"\nClass 3: "In progress"\nClass 4: "Closed with monetary relief"\nClass 5: "Closed without r
tier"\nClass 6: "Closed"\nClass 7: "Untimely response"\nClass 8: "Closed with relief" "
>>>
>>>
>>> # Define a dictionary mapping class indices to class names
... class_names = {
...     1: "Closed with explanation",
...     2: "Closed with non-monetary relief",
...     3: "In progress",
...     4: "Closed with monetary relief",
...     5: "Closed without relief",
...     6: "Closed",
...     7: "Untimely response",
...     8: "Closed with relief"
... }
>>>
>>> #Result:
... #-----
... for i in range(len(precision)):
...     class_name = class_names[i + 1]
...     print(f"Class_name: {class_name}")
...     print(f"Precision: {precision[i]:.2f}")
...     print(f"Recall: {recall[i]:.2f}")
...     print(f"Accuracy: {accuracy[i]:.2f}")
...     print()
... 
```

```
Closed with explanation:
Precision: 0.73
Recall: 0.89
Accuracy: 0.95

Closed with non-monetary relief:
Precision: 0.65
Recall: 0.29
Accuracy: 0.89

In progress:
Precision: 0.66
Recall: 0.49
Accuracy: 0.91

Closed with monetary relief:
Precision: 0.75
Recall: 0.54
Accuracy: 0.92

Closed without relief:
Precision: 0.41
Recall: 0.49
Accuracy: 0.85

Closed:
Precision: 0.42
Recall: 0.75
Accuracy: 0.84

Untimely response:
Precision: 0.55
Recall: 0.36
Accuracy: 0.88

Closed with relief:
Precision: 0.73
Recall: 0.93
Accuracy: 0.95
```

- Train validation: confusion matrix:

```
>>> #important: need to cast to float type, and order by prediction, else it won't work
... preds_and_labels = predictions.select(['prediction', 'indexed_company_response'])\
...                               .withColumn('indexed_company_response', col('indexed_company_response'))\
...                               .cast(FloatType())\
...                               .orderBy('prediction')
...
metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))
>>>
...
>>> metrics = MulticlassMetrics(preds_and_labels.rdd.map(tuple))
confusion_matrix = metrics.confusionMatrix().toArray()

[Stage 2591:==>                                     (109 + 5) / 1605]
>>>
>>> confusion_matrix = metrics.confusionMatrix().toArray()
>>>
>>> print(metrics.confusionMatrix().toArray())
[[3.988e+03 8.400e+01 4.000e+00 3.500e+01 7.900e+01 1.300e+01 2.790e+02
 1.000e+00]
 [3.690e+02 1.323e+03 9.790e+02 5.380e+02 4.800e+02 3.130e+02 3.110e+02
 3.090e+02]
 [5.400e+01 4.820e+02 2.200e+03 1.610e+02 1.390e+02 1.180e+02 7.000e+01
 1.225e+03]
 [1.900e+02 8.600e+01 4.100e+01 2.406e+03 2.360e+02 1.361e+03 1.670e+02
 8.000e+00]
 [1.100e+01 0.000e+00 0.000e+00 0.000e+00 2.151e+03 2.048e+03 2.390e+02
 0.000e+00]
 [1.000e+01 0.000e+00 0.000e+00 0.000e+00 8.540e+02 3.398e+03 2.400e+02
 0.000e+00]
 [7.970e+02 1.500e+01 5.000e+01 3.400e+01 1.322e+03 6.940e+02 1.626e+03
 0.000e+00]
 [2.700e+01 5.100e+01 7.600e+01 4.700e+01 3.900e+01 5.900e+01 9.000e+00
 4.277e+03]]

>>> confusion_matrix_2d = confusion_matrix.toList()
>>>
>>> # Print the 2d list
... print(confusion_matrix_2d)
[[3988.0, 84.0, 4.0, 35.0, 79.0, 13.0, 279.0, 1.0], [369.0, 1323.0, 979.0, 538.0, 480.0, 313.0, 311.0, 309.0], [54.0, 482.0, 2200.0, 161.0, 139.0, 118.0, 70.0, 1275.0], [190.0, 36.0, 41.0, 2406.0, 236.0, 1361.0, 167.0, 5.0], [11.0, 0.0, 0.0, 0.0, 2151.0, 2048.0, 239.0, 0.0], [10.0, 0.0, 0.0, 0.0, 854.0, 3398.0, 240.0, 0.0], [797.0, 15.0, 50.0, 0.0, 1322.0, 694.0, 1626.0, 0.0], [27.0, 51.0, 76.0, 47.0, 39.0, 59.0, 9.0, 4277.0]]
```

Confusion Matrix:

To visualize confusion matrices for a Random Forest model's cross-validation and train-validation results using Python. We import pandas, numpy, seaborn, and matplotlib in google colab as pyspark does not support visualization. Confusion matrix data is defined as 2D arrays (results retrieved from pyspark) for both cross-validation and train-validation results, then converted into pandas DataFrames. Using seaborn, we plot these matrices as heatmaps, setting figure size, annotations, color map, labels, and titles. This visualization provides a clear view of the model's classification performance.

Here is the code to perform Confusion Matrix:

```
"""Random_Forest.ipynb
```

Automatically generated by Colab.

Original file is located at

<https://colab.research.google.com/drive/1i9ed4zG2sHosWXgQK76M3oiEeirfK41H>

"""

```
import pandas as pd
```

```
import numpy as np
```

```
import seaborn as sns
```

```
import matplotlib.pyplot as plt
```

```
# Sample data as a 2D array
```

```
confusion_matrix_rf_data= [[3.988e+03, 8.400e+01, 4.000e+00, 3.500e+01,  
7.900e+01, 1.300e+01, 2.790e+02,
```

```
1.000e+00],
```

```
[3.690e+02, 1.323e+03, 9.790e+02, 5.380e+02, 4.800e+02, 3.130e+02,  
3.110e+02,
```

```
3.090e+02],
```

```
[5.400e+01, 4.820e+02, 2.200e+03, 1.610e+02, 1.390e+02, 1.180e+02,  
7.000e+01,
```

```
1.225e+03],
```

```
[1.900e+02, 8.600e+01, 4.100e+01, 2.406e+03, 2.360e+02, 1.361e+03,  
1.670e+02,
```

```
8.000e+00],
```

```
[1.100e+01, 0.000e+00, 0.000e+00, 0.000e+00, 2.181e+03, 2.048e+03,  
2.390e+02,
```

```
0.000e+00],
```

```
[1.000e+01, 0.000e+00, 0.000e+00, 0.000e+00 ,8.540e+02, 3.398e+03,  
2.400e+02,
```

```
0.000e+00],
```

```
[7.970e+02, 1.500e+01, 5.000e+01, 3.400e+01, 1.322e+03, 6.940e+02,
1.626e+03,
    0.000e+00],
[2.700e+01, 5.100e+01, 7.600e+01, 4.700e+01, 3.900e+01, 5.900e+01,
9.000e+00,
    4.277e+03]]

# Convert to DataFrame
confusion_matrix_rf_cv = pd.DataFrame(confusion_matrix_rf_data)

# Set index and column names
confusion_matrix_rf_cv.index.name = 'Actual'
confusion_matrix_rf_cv.columns.name = 'Predicted'
confusion_matrix_rf_cv = pd.DataFrame(confusion_matrix_rf_data,
index=["1", "2", "3", "4", "5" , "6", "7", "8"], columns=["1", "2",
"3", "4", "5" , "6", "7", "8"])

# Plot heatmap
plt.figure(figsize=(8, 6))

sns.heatmap(confusion_matrix_rf_cv, annot=True, fmt=".0f",
cmap="YlGnBu")

plt.xlabel("Predicted")
plt.ylabel("True Label")

plt.title(" Random Forest Cross Validation Confusion Matrix")

plt.show()

# Sample data as a 2D array
```



```

confusion_matrix_rf_data= [[3.988e+03, 8.400e+01, 4.000e+00 ,3.500e+01,
7.900e+01, 1.300e+01 ,2.790e+02,
    1.000e+00],
    [3.690e+02, 1.323e+03, 9.790e+02, 5.380e+02, 4.800e+02, 3.130e+02,
3.110e+02,
    3.090e+02],
    [5.400e+01, 4.820e+02, 2.200e+03, 1.610e+02, 1.390e+02, 1.180e+02,
7.000e+01,
    1.225e+03],
    [1.900e+02, 8.600e+01, 4.100e+01, 2.406e+03, 2.360e+02, 1.361e+03,
1.670e+02,
    8.000e+00],
    [1.100e+01, 0.000e+00, 0.000e+00 ,0.000e+00, 2.181e+03, 2.048e+03,
2.390e+02,
    0.000e+00],
    [1.000e+01, 0.000e+00, 0.000e+00, 0.000e+00, 8.540e+02, 3.398e+03
,2.400e+02,
    0.000e+00],
    [7.970e+02, 1.500e+01, 5.000e+01, 3.400e+01, 1.322e+03, 6.940e+02,
1.626e+03,
    0.000e+00],
    [2.700e+01 ,5.100e+01, 7.600e+01, 4.700e+01, 3.900e+01 ,5.900e+01,
9.000e+00,
    4.277e+03]]

# Convert to DataFrame
confusion_matrix_rf_tv = pd.DataFrame(confusion_matrix_rf_data)

# Set index and column names

```

```

confusion_matrix_rf_tv.index.name = 'Actual'

confusion_matrix_rf_tv.columns.name = 'Predicted'

confusion_matrix_rf_tv = pd.DataFrame(confusion_matrix_rf_data,
index=["1", "2", "3", "4", "5" , "6", "7", "8"], columns=["1", "2",
"3", "4", "5" , "6", "7", "8"])

# Plot heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(confusion_matrix_rf_tv, annot=True, fmt=".0f",
cmap="YlGnBu")

plt.xlabel("Predicted")

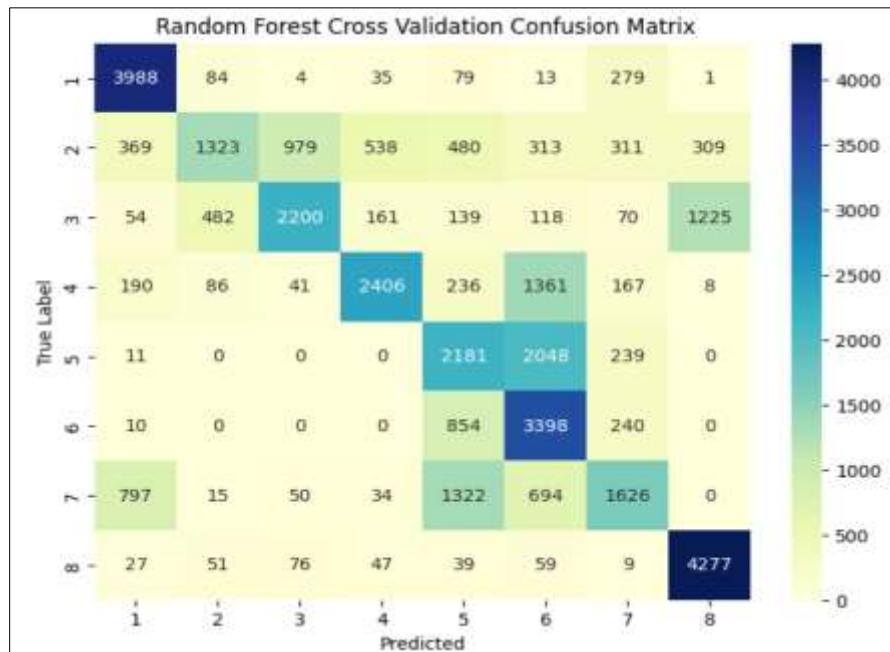
plt.ylabel("True Label")

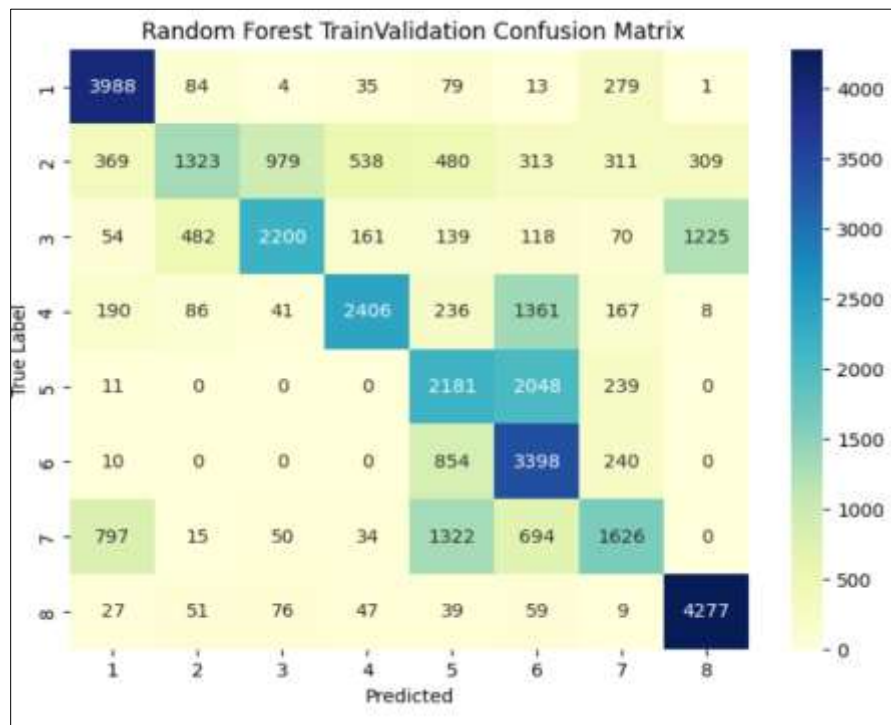
plt.title(" Random Forest TrainValidation Confusion Matrix")

plt.show()

```

The Result:





C. Timely Response using Gradient Boosting Tree:

To set up the environment for your Gradient Boosting Tree (GBT) model using PySpark, you'll begin by importing the necessary libraries and configuring logging for informational messages. You start by importing essential modules, including SparkSession for managing Spark applications, VectorAssembler and StringIndexer for feature transformation, GBTClassifier for the classification model, and BinaryClassificationEvaluator for model evaluation. You'll also import various functions and types to handle data manipulation and transformations.

Here is the code:

```
import logging

logging.basicConfig(level=logging.INFO, format='%(message)s')

from pyspark.sql import SparkSession
```

```
from pyspark.ml.feature import VectorAssembler, StringIndexer

from pyspark.ml.classification import GBTClassifier

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.sql.types import StringType, TimestampType

from pyspark.sql.functions import year, month, dayofmonth, count,
col, when, lit

from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from pyspark.ml import Pipeline

from pyspark.context import SparkContext

from pyspark.sql.session import SparkSession

import pandas as pd


PYSPARK_CLI = True

if PYSPARK_CLI:

    sc = SparkContext.getOrCreate()

    spark = SparkSession(sc)
```

- **Data Preprocessing:**

This step involves cleaning and preparing your data for analysis. It typically includes handling missing values, encoding categorical variables, normalizing numerical features, and ensuring the data is in a suitable format for the model. In this step, we preprocess the data to prepare it for model training. We

start by reading the JSON file 'complaints.json' into a DataFrame named 'raw_complaints'. This file contains the customer complaint data that we will use for analysis. Next, we select the necessary columns from the 'raw_complaints' DataFrame, specifically 'company', 'product', 'timely', 'issue', 'state', and 'date_sent_to_company'. We also filter out any records that are corrupted. To ensure the data is clean, we use 'zipWithIndex' to index the rows and then remove the first row, which may contain header information or be otherwise unsuitable for analysis. We further clean the data by dropping any rows where the 'timely' column is empty. Next, we convert the 'date_sent_to_company' column to 'TimestampType' to facilitate date-based transformations. Finally, we extract the year, month, and day from the 'date_sent_to_company' column. These new columns will help us understand temporal patterns in the data.

```
# Data Preprocessing

# -----

# Read the JSON file 'complaints.json' into a DataFrame named
'raw_complaints'

raw_complaints = spark.read.json('5560_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records

complaint_df = raw_complaints.select('company', 'product', 'timely',
'issue', 'state',
'date_sent_to_company').filter(raw_complaints['_corrupt_record'].isNull())

# ZipWithIndex and remove first row

complaint_df = complaint_df.rdd.zipWithIndex().filter(lambda x: x[1] >
0).map(lambda x: x[0]).toDF()

# Drop rows with timely=None

df_timely_initial = complaint_df.filter(col("timely") != "")

# Cast date_sent_to_company to TimestampType
```

```

df_timely_initial =
df_timely_initial.withColumn("date_sent_to_company",
col("date_sent_to_company").cast(TimestampType()))

# Extract year, month, and day from date_sent_to_company

df_timely_initial = df_timely_initial.withColumn("year",
year("date_sent_to_company")) \

                                .withColumn("month",
month("date_sent_to_company")) \

                                .withColumn("day",
dayofmonth("date_sent_to_company"))

```

```

... # sc = SparkContext.getOrCreate()
... # spark = SparkSession(sc)
...
>>> # Data Preprocessing
... # -----
...
>>> # Read the JSON file 'complaints.json' into a DataFrame named 'raw_complaints'
... raw_complaints = spark.read.json('5560_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records
complaint_df = raw_complaints.select('company', 'product', 'timely', 'issue', 'state', 'date_sent_to_company').
filter(raw_complaints['_corrupt_record'].isNull())

# ZipWithIndex and remove first row
complaint_df = complaint_df.rdd.zipWithIndex().filter(lambda x: x[1] > 0).map(lambda x: x[0]).toDF()

# Drop rows with timely=None
df_timely_initial = complaint_df.filter(col("timely") != "")

# Cast date_sent_to_company to TimestampType
df_timely_initial = df_timely_initial.withColumn("date_sent_to_company", col("date_sent_to_company").cast(TimestampType()))

# Extract year, month, and day from date_sent_to_company
df_timely_initial = df_timely_initial.withColumn("year", year("date_sent_to_company")) \
                                .withColumn("month", month("date_sent_to_company")) \
                                .withColumn("day", dayofmonth("date_sent_to_company"))

```

- **Feature Engineering:**

In feature engineering, you create new features or modify existing ones to improve the performance of your model. This can include creating new variables, transforming existing variables, or encoding categorical variables. In this feature engineering step, we enhance our dataset by calculating the frequency of occurrences for specific features and adding these as new columns. First, we calculate the frequency of each company by grouping the data by the **company** column and counting the number of records for each

company. This frequency is then added to the original DataFrame by performing a left join on the **company** column. Next, we calculate the frequency of each issue by grouping the data by the **issue** column and counting the records. This frequency is added to the DataFrame by joining on the **issue** column. Similarly, we calculate the frequency of each state by grouping the data by the **state** column and counting the number of records. This frequency is added to the DataFrame by joining on the **state** column.

Feature Engineering

```
# -----
```

```
# Calculate the frequency of each company
```

```
company_frequency =
```

```
df_timely_initial.groupBy("company").agg(count("*").alias("frequency_c  
ompany"))
```

```
# Join the frequency DataFrame with the original DataFrame on the  
company column
```

```
df_timely = df_timely_initial.join(company_frequency, on="company",  
how="left")
```

```
# Calculate the frequency of each issue
```

```
issue_frequency =
```

```
df_timely_initial.groupBy("issue").agg(count("*").alias("frequency_iss  
ue"))
```

```
# Join the issue frequency DataFrame with the existing DataFrame on the  
issue column
```

```

df_timely = df_timely.join(issue_frequency, on="issue", how="left")

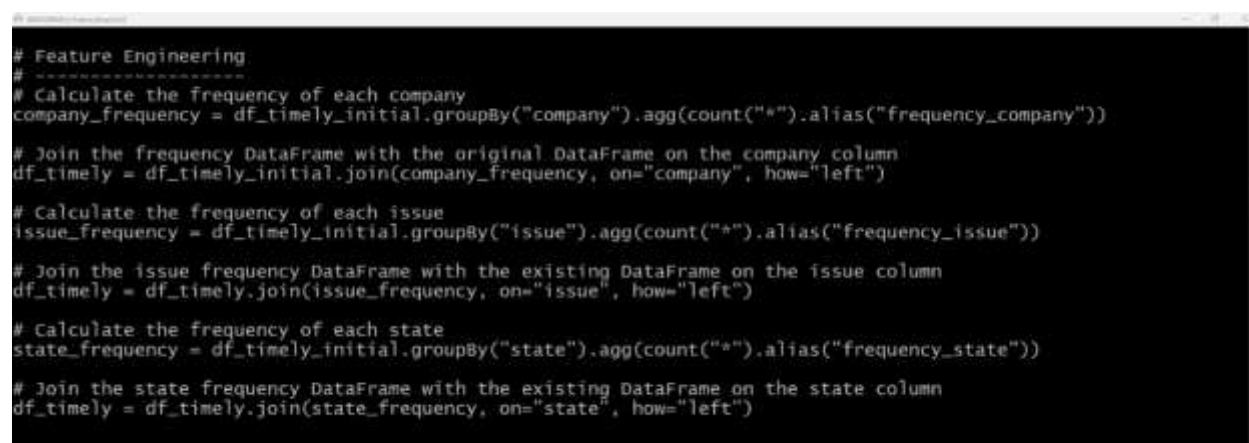
# Calculate the frequency of each state

state_frequency =
df_timely_initial.groupBy("state").agg(count("*").alias("frequency_state"))

# Join the state frequency DataFrame with the existing DataFrame on the
state column

df_timely = df_timely.join(state_frequency, on="state", how="left")

```



```

# Feature Engineering
# -----
# Calculate the frequency of each company
company_frequency = df_timely_initial.groupBy("company").agg(count("*").alias("frequency_company"))

# Join the frequency DataFrame with the original DataFrame on the company column
df_timely = df_timely_initial.join(company_frequency, on="company", how="left")

# Calculate the frequency of each issue
issue_frequency = df_timely_initial.groupBy("issue").agg(count("*").alias("frequency_issue"))

# Join the issue frequency DataFrame with the existing DataFrame on the issue column
df_timely = df_timely.join(issue_frequency, on="issue", how="left")

# Calculate the frequency of each state
state_frequency = df_timely_initial.groupBy("state").agg(count("*").alias("frequency_state"))

# Join the state frequency DataFrame with the existing DataFrame on the state column
df_timely = df_timely.join(state_frequency, on="state", how="left")

```

- **Prepare Pipeline:**

A pipeline automates the workflow of data preprocessing and model training. It ensures that the same sequence of steps is applied consistently to both the training and test datasets. In this step, we prepare a machine learning pipeline to streamline the model training process. We start by defining the features for the model, which include **product**, **frequency_company**, **frequency_issue**, and **frequency_state**. We then create a list of stages for the pipeline. The first stage involves string indexing for the **product** feature, converting it into numerical form. The second stage is a **VectorAssembler**, which combines the indexed

product, frequency features, and date-related features (**year**, **month**, **day**) into a single feature vector called **assembledFeatures**. The third stage involves string indexing for the **timely** label to convert it into a numerical format. Finally, the fourth stage is the GBT (Gradient Boosting Tree) model, which uses the assembled features and the indexed label for classification. This pipeline ensures all necessary transformations and model training are executed systematically and efficiently.

```
# Prepare Pipeline

# -----

# Define features_for_model

features_for_model = ["product", "frequency_company",
"frequency_issue", "frequency_state"]

# Create a list of stages for the pipeline

stages = []

# Stage 1: String indexing for categorical features

indexer_product = StringIndexer(inputCol="product",
outputCol="indexed_product")

stages.append(indexer_product)

# Stage 2: Assemble features

assembler = VectorAssembler(inputCols=["indexed_product",
"frequency_company", "frequency_issue", "frequency_state", "year",
"month", "day"], outputCol="assembledFeatures")

stages.append(assembler)

# Stage 3: String indexing for label

label_indexer = StringIndexer(inputCol="timely", outputCol="label")
```

```

stages.append(label_indexer)

# Stage 4: GBT model

gbt = GBTClassifier(featuresCol="assembledFeatures", labelCol="label",
maxIter=10)

stages.append(gbt)

```



```

# Prepare Pipeline
# -----

# Define features_for_model
features_for_model = ["product", "frequency_company", "frequency_issue", "frequency_state"]

# Create a list of stages for the pipeline
stages = []

# Stage 1: String indexing for categorical features
indexer_product = StringIndexer(inputCol="product", outputCol="indexed_product")
stages.append(indexer_product)

# Stage 2: Assemble features
assembler = VectorAssembler(inputCols=["indexed_product", "frequency_company", "frequency_issue", "frequency_state", "year", "month", "day"], outputCol="assembledFeatures")
stages.append(assembler)

# Stage 3: String indexing for label
label_indexer = StringIndexer(inputCol="timely", outputCol="label")
stages.append(label_indexer)

# Stage 4: GBT model
gbt = GBTClassifier(featuresCol="assembledFeatures", labelCol="label", maxIter=10)
stages.append(gbt)

```

- **Balancing the Data by Oversampling Minority Class (timely = No):**

Since your target variable is imbalanced (e.g., more "Yes" than "No"), you use oversampling to increase the number of "No" instances. This helps the model learn to recognize both classes more effectively. To address the class imbalance in the dataset where the minority class is **timely = No**, we first drop unnecessary columns such as **company**, **issue**, **state**, and **date_sent_to_company**. We then isolate the minority class (**timely = No**) into a separate DataFrame. To balance the dataset, we calculate the ratio of the majority class (**timely = Yes**) to the minority class. Using this ratio, we oversample the minority class by creating a sample with replacement that matches the calculated ratio. Finally, we combine the oversampled minority class DataFrame with the majority class DataFrame to achieve a balanced dataset, ensuring both classes are represented equally in the training data.

```

# Balancing the data by Oversampling minority class (timely = No)

# -----

```

```

# Dropping additional columns

df_timely = df_timely.drop('company', 'issue', 'state',
'date_sent_to_company')

# Oversample the minority class

negative_df = df_timely.filter(col("timely") == "No")

balanced_ratio = df_timely.filter(col("timely") == "Yes").count() /
negative_df.count()

oversampled_negative_df = negative_df.sample(withReplacement=True,
fraction=balanced_ratio)

df_timely = df_timely.filter(col("timely") ==
"Yes").union(oversampled_negative_df)

```

```

# Balancing the data by Oversampling minority class (timely = No)
# -----

# Dropping additional columns
df_timely = df_timely.drop('company', 'issue', 'state', 'date_sent_to_company')

# Oversample the minority class
negative_df = df_timely.filter(col("timely") == "No")
balanced_ratio = df_timely.filter(col("timely") == "Yes").count() / negative_df.count()
oversampled_negative_df = negative_df.sample(withReplacement=True, fraction=balanced_ratio)
df_timely = df_timely.filter(col("timely") == "Yes").union(oversampled_negative_df)

```

- **Model Training:**

To train the model, we start by persisting the balanced DataFrame in memory for efficient access. We then split the data into training and testing sets, with 70% of the data allocated for training and 30% for testing, ensuring reproducibility with a fixed seed. We log the number of rows in each set to confirm the split. Next, we combine the preprocessing and modeling stages into a single pipeline. We define a binary classification evaluator to assess the model's performance using the area under the ROC curve (AUC). A parameter grid is created to specify hyperparameters for tuning, including different values for the

maximum depth and the number of iterations for the Gradient Boosting Tree (GBT) classifier. We use a cross-validator to perform three-fold cross-validation, optimizing the model based on the parameter grid. Finally, we measure and log the training time, ensuring the process is efficient and manageable.

```
# Model Training
# -----

# Persist the DataFrame in memory
from pyspark.storagelevel import StorageLevel
df_timely.persist(StorageLevel.MEMORY_ONLY)

# Split data into training and testing sets
train, test = df_timely.randomSplit([0.7, 0.3], seed=42)

# Print the number of rows in train and test DataFrames
logging.info("Number of rows in train DataFrame:
{}".format(train.count()))
logging.info("Number of rows in test DataFrame:
{}".format(test.count()))

# Combine stages into a pipeline
pipeline = Pipeline(stages=stages)

# Define evaluator
evaluator = BinaryClassificationEvaluator(labelCol="label",
rawPredictionCol="rawPrediction", metricName="areaUnderROC")

# Define paramGrid
paramGrid = ParamGridBuilder() \
    .addGrid(gbt.maxDepth, [3, 5]) \
    .addGrid(gbt.maxIter, [10, 20]) \
    .build()

# Create a CrossValidator
cv = CrossValidator(estimator=pipeline, evaluator=evaluator,
estimatorParamMaps=paramGrid, numFolds=3)
```

```
# Measure training time
import time
start_time = time.time()
model = cv.fit(train)
end_time = time.time()
training_time = end_time - start_time
minutes = int(training_time // 60)
seconds = int(training_time % 60)
logging.info("Training time: %02d:%02d" % (minutes, seconds))
```

- **Training Data Size:** 6,851,091 rows (approximately 7 million rows)
- **Test Data Size:** 2,935,114 rows (approximately 3 million rows)
- **Training Time:** 35.45 minutes

```
>>>
>>>
>>> # Model Training
>>> # -----
>>>
>>> # Persist the DataFrame in memory
... from pyspark.storagelevel import StorageLevel
>>> df_timely.persist(StorageLevel.MEMORY_ONLY)
DataFrame[product: string, timely: string, year: int, month: int, day: int, frequency_company: bigint, frequency_issue: bigint, frequency_state: b
igint]
>>>
>>> # Split data into training and testing sets
... train, test = df_timely.randomSplit([0.7, 0.3], seed=42)
>>>
>>> # Print the number of rows in train and test DataFrames
... logging.info("Number of rows in train DataFrame: {}".format(train.count()))
Number of rows in train DataFrame: 6852061
>>> logging.info("Number of rows in test DataFrame: {}".format(test.count()))
Number of rows in test DataFrame: 2935114
>>>
>>>
>>> # Combine stages into a pipeline
... pipeline = Pipeline(stages=stages)
>>>
>>> # Define evaluator
... evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
>>>
>>> # Define paramGrid
... paramGrid = ParamGridBuilder() \
...     .addGrid(gbt.maxDepth, [1, 5]) \
...     .addGrid(gbt.maxIter, [10, 20]) \
...     .build()
>>>
>>> # Create a CrossValidator
... cv = CrossValidator(estimator=pipeline, evaluator=evaluator, estimatorParamMaps=paramGrid, numFolds=3)
>>>
>>> # Measure training time
... import time
>>> start_time = time.time()
>>> model = cv.fit(train)
Closing down clientserver connection
```

- **Feature Importance – Cross Validation:**

To train the model, we start by persisting the balanced DataFrame in memory for efficient access. We then split the data into training and testing sets, with 70% of the data allocated for training and 30% for testing, ensuring reproducibility with a fixed seed. We log the number of rows in each set to confirm the split. Next, we combine the preprocessing and modeling stages into a single pipeline. We define a binary classification evaluator to assess the model's performance using the area under the ROC curve (AUC). A parameter grid is created to specify hyperparameters for tuning, including different values for the maximum depth and the number of iterations for the Gradient Boosting Tree (GBT) classifier. We use a

cross-validator to perform three-fold cross-validation, optimizing the model based on the parameter grid. Finally, we measure and log the training time, ensuring the process is efficient and manageable.

```
# Feature Importance - Cross Validation
# -----

# Calculate feature importances
featureImportances = model.bestModel.stages[-1].featureImportances

# Get feature names
#featureNames = df_timely.select(features_for_model).columns
featureNames = df_timely.select(features_for_model + ['year', 'month',
'year', 'month', 'day']).columns

# Filter featureNames to only include the features that are present in
featureImportances
importancesList = [(name, round(importance, 2)) for name, importance in
zip(featureNames, featureImportances) if importance > 0]

# Sort importancesList by importance in descending order
importancesList = sorted(importancesList, key=lambda x: x[1],
reverse=True)

# Print the feature importances
for feature, importance in importancesList:
    logging.info("Feature: {}, Importance: {}".format(feature,
importance))
```

```
>>> # training_time_formatted = "{:02d}:{:02d}".format(minutes, seconds)
... # print("Training time:", training_time_formatted)
... logging.info("Training time: %02d:%02d" % (minutes, seconds))
Training time: 35:45
```

- **frequency_company:** 0.69
- **year:** 0.14
- **product:** 0.09
- **month:** 0.05
- **day:** 0.02
- **frequency_issue:** 0.0
- **frequency_state:** 0.0

```
...
Feature: frequency_company, Importance: 0.69
Feature: year, Importance: 0.14
Feature: product, Importance: 0.09
Feature: month, Importance: 0.05
Feature: day, Importance: 0.02
Feature: frequency_issue, Importance: 0.0
Feature: frequency_state, Importance: 0.0
```

- **Model Evaluation – Cross Validation:**

After training the model, we evaluate its performance using the test set. We make predictions on the test data and calculate key metrics: true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). These metrics are derived by filtering the predictions based on the actual and predicted labels. We then calculate the Area Under the ROC Curve (AUC) using the evaluator. Precision and recall are computed to assess the model's accuracy and sensitivity, respectively. These metrics, along with the AUC, are compiled into a DataFrame for clarity. Finally, we log the evaluation results, providing a comprehensive view of the model's performance during cross-validation. This step ensures that the model is evaluated rigorously, highlighting its strengths and areas for improvement.

```
# Model Evaluation - Cross Validation
# -----

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

# Calculate metrics
tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
auc = evaluator.evaluate(predictions)

# Create DataFrame with evaluation metrics
metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),
    ("Precision", round(tp / (tp + fp), 2)),
    ("Recall", round(tp / (tp + fn), 2)),
    ("AUC", round(auc, 2))
```

```
], ["metric", "value"])

# Print evaluation metrics
logging.info("***** GBT CrossValidator Results *****")
metrics.show()
```

- **True Positives (TP):** 1,360,216
- **False Positives (FP):** 233,426
- **True Negatives (TN):** 1,234,651
- **False Negatives (FN):** 107,280
- **Precision:** 0.85
- **Recall:** 0.93
- **AUC (Area Under the Curve):** 0.94


```

>>> # Model Evaluation
... # -----
...
>>> # Make predictions on the test set
... predictions = model.transform(test)

# Calculate metrics
tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
fp = float(predictions.filter("p>>>
>>> # Calculate metrics
... tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
auc = evaluator.evaluate(predictions)

# Create DataFrame with evaluation metrics
metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),
    ("Precision", round(tp / (tp + fp), 2)),
    ("Recall", round(tp / (tp + fn), 2)),
    ("AUC", round(auc, 2))
], ["metric", "value"])

# Print evaluation metrics
#print("*****CrossValidator Results *****")
logging.info("*****CrossValidator Results *****")
>>> fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
>>> tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
>>> fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
>>> auc = evaluator.evaluate(predictions)
>>>
>>> # Create DataFrame with evaluation metrics
... metrics = spark.createDataFrame([
...     ("TP", tp),
...     ("FP", fp),
...     ("TN", tn),
...     ("FN", fn),
...     ("Precision", round(tp / (tp + fp), 2)),
...     ("Recall", round(tp / (tp + fn), 2)),
...     ("AUC", round(auc, 2))
... ], ["metric", "value"])
>>>
>>> # Print evaluation metrics
... #print("*****CrossValidator Results *****")
... logging.info("*****CrossValidator Results *****")
*****CrossValidator Results *****
>>> metrics.show()

```

```

>>> logging.info("***** GBT CrossValidator Results *****")
***** GBT CrossValidator Results *****
>>> metrics.show()
+-----+-----+
| metric | value |
+-----+-----+
| TP     | 1360216.0 |
| FP     | 233426.0 |
| TN     | 1234651.0 |
| FN     | 107280.0 |
| Precision | 0.85 |
| Recall  | 0.93 |
| AUC     | 0.94 |
+-----+-----+

```

- **Train Validation Split:**

To further validate the model, we use Train-Validation Split, a simpler alternative to cross-validation. We define a TrainValidationSplit with the pipeline as the estimator, the previously defined parameter grid, and the evaluator. The training ratio is set to 0.8, meaning 80% of the data is used for training and 20% for validation. We measure the training time by recording the start and end times of the model fitting process. The TrainValidationSplit model (tvModel) is then fit to the training data. After training, we calculate and log the total training time, providing the duration in minutes and seconds. This approach offers a balance between model validation and computational efficiency, ensuring the model is well-tuned while reducing the time required for training.

```
#Train Split Validation
#-----

from pyspark.ml.tuning import TrainValidationSplit
from pyspark.sql.functions import col

# Define TrainValidationSplit
trainval = TrainValidationSplit(estimator=pipeline,
                                estimatorParamMaps=paramGrid,
                                evaluator=evaluator,
                                trainRatio=0.8)

# Training the model and Calculating its time
import time
start_time = time.time()

# Fit the cross validator to the training data
tvModel = trainval.fit(train)
end_time = time.time()
print("Model trained!")

# Calculate training time
training_time = end_time - start_time

# Calculate minutes and seconds
```

```

minutes = int(training_time // 60)

seconds = int(training_time % 60)

logging.info("Training time: %02d:%02d" % (minutes, seconds))

```

```

>>> # Define TrainValidationSplit
... trainval = TrainValidationSplit(estimator=pipeline,
...                               estimatorParamMaps=paramGrid,
...                               evaluator=evaluator,
...                               trainRatio=0.8)
>>>
>>> # Training the model and Calculating its time
... import time
>>>
>>> start_time = time.time()
>>>
>>> # Fit the cross validator to the training data
... tvModel = trainval.fit(train)
end_time = time.time()

print("Model trained!")

# Calculate training time
training_time = end_time - start_time

# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

logging.info("Training time: %02d:%02d" % (minutes, seconds))
Closing down clientserver connection
24/05/01 02:57:16 WARN BlockManager: Asked to remove block broadcast_2895_piece0, which does not exist
24/05/01 02:57:16 WARN BlockManager: Asked to remove block broadcast_2895, which does not exist
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
>>>
>>> end_time = time.time()
>>>
>>> print("Model trained!")
Model trained!
>>>
>>>
>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)

```

- **Training Time:** 13.48 minutes

```

>>>
>>> logging.info("Training time: %02d:%02d" % (minutes, seconds))
Training time: 13:48

```

- **Feature Importance – Train Validation Split:**

To understand which features are most influential in our model, we calculate feature importances using the best model from the Train-Validation Split. We extract the feature importances from the model's final stage and obtain the corresponding feature names from the DataFrame. We then create a list of feature

names and their respective importances, rounding the importance values to two decimal places. This list is filtered to include only the features with non-zero importance. We sort the list in descending order by importance to highlight the most significant features. Finally, we log each feature's name and its importance, providing insight into which features contribute most to the model's predictions. This step is crucial for interpreting the model and understanding the driving factors behind its decisions.

```
# Feature Importance - Train Validation Split

# -----

# Calculate feature importances

featureImportances = tvModel.bestModel.stages[-1].featureImportances

# Get feature names

featureNames = df_timely.select(features_for_model + ['year', 'month',
'day']).columns

# Filter featureNames to only include the features that are present in
featureImportances

importancesList = [(name, round(importance, 2)) for name, importance in
zip(featureNames, featureImportances) if importance > 0]

# Sort importancesList by importance in descending order

importancesList = sorted(importancesList, key=lambda x: x[1],
reverse=True)

# Print the feature importances

for feature, importance in importancesList:

    logging.info("Feature: {}, Importance: {}".format(feature,
importance))
```

- **frequency_company**: 0.69
- **year**: 0.14
- **product**: 0.09
- **month**: 0.05
- **day**: 0.02
- **frequency_issue**: 0.0
- **frequency_state**: 0.0

```
>>> # Calculate feature importances
.. featureImportances = tvModel.bestModel.stages[-1].featureImportances
>>>
>>> # Get feature names
.. featureNames = df_timely.select(features_for_model + ['year', 'month', 'day']).columns
>>> # Filter featureNames to only include the features that are present in featureImportances
.. importancesList = [(name, round(importance, 2)) for name, importance in zip(featureNames, featureImportances) if importance > 0]
>>> # Sort importancesList by importance in descending order
.. importancesList = sorted(importancesList, key=lambda x: x[1], reverse=True)
>>>
>>> # Print the feature importances
.. for feature, importance in importancesList:
..     logging.info("Feature: {}, Importance: {}".format(feature, importance))
..
Feature: frequency_company, Importance: 0.69
Feature: year, Importance: 0.14
Feature: product, Importance: 0.09
Feature: month, Importance: 0.05
Feature: day, Importance: 0.02
Feature: frequency_issue, Importance: 0.0
Feature: frequency_state, Importance: 0.0
```

- **Model Evaluation – Train Validation Split:**

To evaluate the performance of our model trained using the Train-Validation Split, we make predictions on the test data using the best model from the split. We then assess the model's accuracy using the defined evaluator. Key metrics such as true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) are extracted from the predictions. Precision and recall are calculated to gauge the model's accuracy and sensitivity, respectively. These metrics, along with the Area Under the ROC Curve (AUC), are compiled into a DataFrame for better visualization and interpretation. Finally, we log the evaluation results to provide a detailed overview of the model's performance. This evaluation step is essential for validating the model's effectiveness and ensuring it generalizes well to unseen data.

```
# Model Evaluation - Train Validation Split
# -----

# Make predictions on the test data using the best model

predictions = tvModel.transform(test)

# Model Evaluate
```

```
accuracy = evaluator.evaluate(predictions)

# Extract TP, FP, TN, FN

tp = float(predictions.filter("prediction == 1.0 AND label ==
1").count())

fp = float(predictions.filter("prediction == 1.0 AND label ==
0").count())

tn = float(predictions.filter("prediction == 0.0 AND label ==
0").count())

fn = float(predictions.filter("prediction == 0.0 AND label ==
1").count())

# Calculate precision and recall

precision = round(tp / (tp + fp), 2)

recall = round(tp / (tp + fn), 2)

# Create DataFrame with evaluation metrics

metrics = spark.createDataFrame([

    ("TP", tp),

    ("FP", fp),

    ("TN", tn),

    ("FN", fn),
```

```
    ("Precision", precision),  
  
    ("Recall", recall),  
  
    ("AUC", round(accuracy, 2))  
], ["metric", "value"])  
  
#print("*****TrainValidator Results *****")  
  
logging.info("*****GBT TrainValidator Results *****")  
  
metrics.show()
```

Evaluation metrics using the train-validation split.

- **True Positives (TP):** 1,370,704
- **False Positives (FP):** 259,229
- **True Negatives (TN):** 1,208,848
- **False Negatives (FN):** 96,792
- **Precision:** 0.84
- **Recall:** 0.93
- **AUC (Area Under the Curve):** 0.94

```

MINGW64/c/Users/dvaishn2
>>> # Model Evaluation - Train Validation Split
... # -----
...
>>>
>>> # Make predictions on the test data using the best model
... predictions = tvModel.transform(test)

# Model Evaluate
accuracy = evaluator.evaluate(predictions)
>>>
>>> # Model Evaluate
... accuracy = evaluator.evaluate(predictions)

# Extract TP, FP, TN, FN
tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())

# Calculate precision and recall
precision = round(tp / (tp + fp), 2)
recall = round(tp / (tp + fn), 2)

# Create DataFrame with evaluation metrics
metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),
    ("Precision", precision),
    ("Recall", recall),
    ("AUC", round(accuracy, 2))
], ["metric", "value"])

#print("*****TrainValidator Results *****")
logging.info("*****GBT TrainValidator Results *****")

>>>
>>> # Extract TP, FP, TN, FN
... tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
>>> fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
>>> tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
>>> fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
>>>
>>> # Calculate precision and recall
... precision = round(tp / (tp + fp), 2)
>>> recall = round(tp / (tp + fn), 2)
>>>
>>> # Create DataFrame with evaluation metrics
... metrics = spark.createDataFrame([
...     ("TP", tp),
...     ("FP", fp),
...     ("TN", tn),
...     ("FN", fn),
...     ("Precision", precision),
...     ("Recall", recall),
...     ("AUC", round(accuracy, 2))
... ], ["metric", "value"])
>>>

```

- Model Evaluation – Train Validation Split Results:

```

>>>
... #print("*****TrainValidator Results *****")
... logging.info("*****GBT TrainValidator Results *****")
*****GBT TrainValidator Results *****
>>>
>>> metrics.show()
+-----+
| metric | value |
+-----+
| TP     | 1370704.0 |
| FP     | 259229.0 |
| TN     | 1208848.0 |
| FN     | 96792.0 |
| Precision | 0.84 |
| Recall  | 0.93 |
| AUC     | 0.94 |
+-----+

```


D. Timely Response using Linear Regression:

To set up the environment for building a Logistic Regression model with PySpark, we start by configuring logging to display informational messages, which helps in tracking the execution flow and debugging. Next, we import essential PySpark modules including **SparkSession** for managing Spark applications, **VectorAssembler** and **StringIndexer** for feature transformation, and **LogisticRegression** for the classification model. We also import various functions and types to handle data manipulation and transformations, such as extracting year, month, and day from dates. For model tuning and evaluation, we use **CrossValidator**, **ParamGridBuilder**, and **BinaryClassificationEvaluator**. Finally, we initialize the **SparkContext** and **SparkSession** if running in a PySpark CLI environment to manage Spark operations effectively.

```
import logging

logging.basicConfig(level=logging.INFO, format='%(message)s')

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler, StringIndexer

from pyspark.ml.classification import LogisticRegression

from pyspark.sql.types import StringType

from pyspark.sql.functions import year, month, dayofmonth

from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from pyspark.sql.functions import col

from pyspark.sql.types import *

from pyspark.sql.functions import *

from pyspark.ml import Pipeline

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.context import SparkContext

from pyspark.sql.session import SparkSession

import pandas as pd
```

```
# PYSPARK_CLI = True

# if PYSPARK_CLI:

    # sc = SparkContext.getOrCreate()

    # spark = SparkSession(sc)
```

- **Data Preprocessing:**

In the data preprocessing step, we start by reading the JSON file '**complaints.json**' into a DataFrame named **raw_complaints**. This file contains customer complaint data, which will be used for analysis. We select the necessary columns—**company**, **product**, **timely**, **issue**, **state**, and **date_sent_to_company**—while filtering out any corrupt records based on the **_corrupt_record** column. To ensure clean data, we use **zipWithIndex** to index the rows and remove the first row, which may contain header information or be otherwise unsuitable for analysis. Next, we drop rows where the **timely** column is empty, creating a DataFrame for predicting timely responses. We cast the **date_sent_to_company** column to **TimestampType** to facilitate date-based transformations. Finally, we extract the year, month, and day from the **date_sent_to_company** column, which will help in understanding temporal patterns in the data.

```
# Data Preprocessing

# -----

# Read the JSON file 'complaints.json' into a DataFrame named
# 'raw_complaints'

raw_complaints = spark.read.json('5560_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records

complaint_df = raw_complaints.select('company', 'product', 'timely' ,
    'issue', 'state',
    'date_sent_to_company').filter(raw_complaints['_corrupt_record'].isNull())

complaint_df = complaint_df.rdd.zipWithIndex().filter(lambda x: x[1] >
0).map(lambda x: x[0]).toDF()

# drop 1 row having timely = None and Create a dataframe for prediction
of timely_response

df_timely_initial = complaint_df.filter(col("timely") != "")
```

```
# Cast date_sent_to_company to a suitable type 'timestamp'

df_timely_initial =
df_timely_initial.withColumn("date_sent_to_company",
col("date_sent_to_company").cast(TimestampType()))

# Extracting year, month, and day from 'date_sent_to_company' column

df_timely_initial = df_timely_initial.withColumn("year",
year("date_sent_to_company")) \

                                .withColumn("month",
month("date_sent_to_company")) \

                                .withColumn("day",
dayofmonth("date_sent_to_company"))
```

```
>>> # PYSPARK_CLI = True
... # If PYSPARK_CLI:
...     # sc = SparkContext.getOrCreate()
...     # spark = SparkSession(sc)
...
... # Data Preprocessing
... # =====
...
... # Read the JSON file 'complaints.json' into a dataframe named 'raw_complaints'
... raw_complaints = spark.read.json('5540_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records
complaint_df = raw_complaints.select('company', 'product', 'timely', 'issue', 'state', 'date_sent_to_company').filter(raw_complaints['corrupt_record'].isNull())

# Show the first 100 rows of the Dataframe 'complaint_df'
complaint_df.show(100)

# complaint_df = complaint_df.drop("corrupt_record")

complaint_df = complaint_df.rdd.zipWithIndex().filter(lambda x: x[1] > 0).map(lambda x: x[0]).toDF()

# and Create a dataframe for prediction of timely_response (0 + 1) / 100
df_timely_initial = complaint_df.filter(in("timely") != "")

# Cast date_sent_to_company to a suitable type 'timestamp'
df_timely_initial = df_timely_initial.withColumn("date_sent_to_company", col("date_sent_to_company").cast(TimestampType()))

# Extracting year, month, and day from 'date_sent_to_company' column
df_timely_initial = df_timely_initial.withColumn("year", year("date_sent_to_company")) \
                                .withColumn("month", month("date_sent_to_company")) \
                                .withColumn("day", dayofmonth("date_sent_to_company"))

...
... # Select necessary columns and drop corrupt records
... complaint_df = raw_complaints.select('company', 'product', 'timely', 'issue', 'state', 'date_sent_to_company').filter(raw_complaints['corrupt_record'].isNull())
...
... # Show the first 100 rows of the Dataframe 'complaint_df'
... complaint_df.show(100)
```

company	product	timely	issue	state	date_sent_to_company
TRANSUNION INTER...	Credit reporting ...	Yes	Incorrect Informa...	OH	2024-02-20
EQUIFAX, INC.	Credit reporting ...	Yes	Problem with a co...	FL	2024-02-20
EQUIFAX, INC.	Credit reporting ...	Yes	Problem with a co...	FL	2024-02-20
WELLS FARGO & CO...	Credit card	Yes	Problem with a pu...	NC	2024-02-20

- **Feature Engineering:**

In the feature engineering step, we enhance our dataset by calculating the frequency of occurrences for specific features and integrating these as new columns. First, we calculate the frequency of each company by grouping the data by the **company** column and counting the number of records for each company. This frequency year information is then joined with the original DataFrame on the **company** column. Next, we calculate the frequency of each issue by grouping the data by the **issue** column and counting the records. This frequency data is also joined with the DataFrame on the **issue** column. Similarly, we calculate the frequency of each state by grouping the data by the **state** column and counting the number of records. This frequency information is joined with the DataFrame on the **state** column. These additional frequency

features provide the model with valuable context about the relative occurrence of companies, issues, and states, which can improve the model's predictive performance.

```
# Feature Engineering

# -----

# Calculate the frequency of each company

company_frequency =
df_timely_initial.groupBy("company").agg(count("*").alias("frequency_c
ompany"))

# Join the frequency DataFrame with the original DataFrame on the
company column

df_timely = df_timely_initial.join(company_frequency, on="company",
how="left")

# Calculate the frequency of each issue (corrected to avoid duplicate
calculation)

issue_frequency =
df_timely_initial.groupBy("issue").agg(count("*").alias("frequency_iss
ue"))

# Join the issue frequency DataFrame with the existing DataFrame on the
issue column

df_timely = df_timely.join(issue_frequency, on="issue", how="left")

# Calculate the frequency of each issue (corrected to avoid duplicate
calculation)

state_frequency =
df_timely_initial.groupBy("state").agg(count("*").alias("frequency_sta
te"))

# Join the issue frequency DataFrame with the existing DataFrame on the
issue column

df_timely = df_timely.join(state_frequency, on="state", how="left")
```

```

>>> # Feature Engineering
... # -----
>>> # Calculate the frequency of each company
... company_frequency = df_timely_initial.groupby("company").agg(count("*").alias("frequency_company"))
>>>
>>> # Join the frequency DataFrame with the original DataFrame on the company column
... df_timely = df_timely_initial.join(company_frequency, on="company", how="left")
>>>
>>> # Calculate the frequency of each issue (corrected to avoid duplicate calculation)
... issue_frequency = df_timely_initial.groupby("issue").agg(count("*").alias("frequency_issue"))
>>>
>>> # Join the issue frequency DataFrame with the existing DataFrame on the issue column
... df_timely = df_timely.join(issue_frequency, on="issue", how="left")
>>>
>>> # Calculate the frequency of each state (corrected to avoid duplicate calculation)
... state_frequency = df_timely_initial.groupby("state").agg(count("*").alias("frequency_state"))
>>>
>>> # Join the issue frequency DataFrame with the existing DataFrame on the issue column
... df_timely = df_timely.join(state_frequency, on="state", how="left")
>>>
>>> # Show the result
... df_timely.show(10)

```

state	issue	company	product	timely	date_sent_to_company	year	month	day	frequency_company	frequency_issue	frequency_state
WV	Improper use of y...	Experian Informat...	Credit reporting...	Yes	2023-08-08 00:00:00	2023	8	8	852148	742713	167070
WV	Improper use of y...	HYUNDAI CAPITAL A...	Credit reporting...	Yes	2023-08-08 00:00:00	2023	8	8	47571	742713	120233
WV	Getting the loan	LDL Holdings, LLC	Payday loan, titl...	Yes	2023-11-21 00:00:00	2023	11	21	8031	2652	2930
CA	Unauthorized tran...	Paypal Holdings, Inc	Money transfer, v...	Yes	2024-02-15 00:00:00	2024	2	15	22452	6140	568982
MD	Fees or interest	CAPITAL ONE FINAN...	Credit card	Yes	2023-09-01 00:00:00	2023	9	1	97045	23259	130254
IL	Incorrect informa...	Experian Informat...	Credit reporting...	Yes	2023-08-08 00:00:00	2023	8	8	852148	1421023	205026
OH	Problem with a cr...	Experian Informat...	Credit reporting...	Yes	2023-08-08 00:00:00	2023	8	8	852148	38937	223021
OH	Problem caused by...	JP MORGAN CHASE & CO.	Checking or savin...	Yes	2023-11-03 00:00:00	2023	11	3	117116	12020	325021
PA	Managing the loan...	HYUNDAI MOTOR CRE...	Vehicle loan or l...	Yes	2023-11-21 00:00:00	2023	11	21	4872	33291	234009
TX	Incorrect informa...	Experian Informat...	Credit reporting...	Yes	2023-08-08 00:00:00	2023	8	8	852148	1421023	523258

only showing top 10 rows

- **Prepare Pipeline:**

In the pipeline preparation step, we define the features for the model and create a series of stages to preprocess the data and train the model. We start by specifying the features: **product**, **frequency_company**, **frequency_issue**, **frequency_state**, **year**, **month**, and **day**. We then create a list of stages for the pipeline. The first stage involves string indexing the **product** feature to convert it into a numerical format using **StringIndexer**. Next, we use a **VectorAssembler** to combine the indexed **product**, frequency features, and date-related features into a single feature vector called **assembledFeatures**. The third stage involves string indexing the **timely** label to convert it into a numerical format. Finally, we define the Logistic Regression model with the assembled features and indexed label. These stages are appended to the pipeline, ensuring a streamlined and consistent workflow for data preprocessing and model training.

```

# Prepare Pipeline

# -----

# Define features_for_model directly with data types
features_for_model = ["product", "frequency_company",
"frequency_issue", "frequency_state"]

# Create a list of stages for the pipeline

stages = []

# String indexing for categorical features

```

```

indexer_product = StringIndexer(inputCol="product",
outputCol="indexed_product")

# Stage 1: Append all indexers to the stages list

stages.append(indexer_product)

# Create the VectorAssembler instance

assembler = VectorAssembler(inputCols=["indexed_product",
"frequency_company", "frequency_issue", "frequency_state", "year",
"month", "day"], outputCol="assembledFeatures")

# Stage 2: Assemble features

stages.append(assembler)

# Stage 3: String indexing for label

label_indexer = StringIndexer(inputCol="timely", outputCol="label")

stages.append(label_indexer)

# Stage 4: Logistic Regression model

lr = LogisticRegression(featuresCol="assembledFeatures",
labelCol="label")

stages.append(lr)

```

```

>>> # Prepare Pipeline
... #
...
>>> # Define features for model directly with data types
... features_for_model = ["product", "frequency_company", "frequency_issue", "frequency_state"]
...
>>> # Create a list of stages for the pipeline
... stages = []
...
>>> # String indexing for categorical features
... indexer_product = StringIndexer(inputCol="product", outputCol="indexed_product")
...
>>> # Stage 1: Append all indexers to the stages list
... stages.append(indexer_product)
...
>>> # Create the VectorAssembler instance
... assembler = VectorAssembler(inputCols=["indexed_product", "frequency_company", "frequency_issue", "frequency_state", "year", "month", "day"], outputCol="assembledFeatures")
...
>>> # Stage 2: Assemble features
... stages.append(assembler)
...
>>> # Stage 3: String indexing for label
... label_indexer = StringIndexer(inputCol="timely", outputCol="label")
... stages.append(label_indexer)
...
>>> # Stage 4: Logistic Regression model
... lr = LogisticRegression(featuresCol="assembledFeatures", labelCol="label")
... lr = LogisticRegression(featuresCol="assembledFeatures", labelCol="label", weightCol="weight")
>>> stages.append(lr)
>>> stages.append(lr)

```

- **Balancing the Data by Oversampling Minority Class (timely = No):**

To address the class imbalance in the dataset where "timely = No" is the minority class, we perform oversampling. First, we drop additional columns that are not needed for the analysis, such as **company**,

issue, **state**, and **date_sent_to_company**. We then isolate the minority class (**timely = No**) into a separate DataFrame. To achieve a balanced ratio, we calculate the fraction needed for oversampling by dividing the number of "Yes" instances by the number of "No" instances. The minority class is oversampled with replacement using this calculated fraction. Finally, we combine the oversampled minority class with the original majority class data, resulting in a balanced dataset. This approach ensures that both classes are well-represented, improving the model's ability to learn from the data.

```
# Balancing the data by Oversampling minority class (timely = No)

# -----

# dropping additional columns

df_timely = df_timely.drop('company' , 'issue', 'state',
'date_sent_to_company')

df_timely.show(10)

# Oversample the minority class (assuming "No" is the minority)

negative_df = df_timely.filter(col("timely") == "No")

# Calculate the fraction to achieve a more balanced ratio

# For example, if you want a 1:1 ratio, set fraction = number of "Yes"
instances / number of "No" instances

balanced_ratio = df_timely.filter(col("timely") == "Yes").count() /
negative_df.count()

oversampled_negative_df = negative_df.sample(withReplacement=True,
fraction=balanced_ratio)

# Combine oversampled negatives with original data (assuming positive
is the majority)

df_timely = df_timely.filter(col("timely") ==
"Yes").union(oversampled_negative_df)

# def calculate_weights(data, label_column, weight_column,
weight_value):

    # total_count = data.count()

    # positive_count = data.filter(col(label_column) == "Yes").count()
```

```

# negative_count = data.filter(col(label_column) == "No").count()

# weight_positive = lit(total_count) / (2 * positive_count *
weight_value)

# weight_negative = lit(total_count) / (2 * negative_count * (1 -
weight_value))

# return data.withColumn(weight_column, when(col(label_column) ==
"Yes", weight_positive).otherwise(weight_negative))

# # Calculate weights for the minority class

# df_timely_balanced = calculate_weights(df_timely, "timely", "weight",
0.3)

```

```

>>> # balancing the data by oversampling minority class (timely = No)
... # .....
...
... # dropping additional columns
... df_timely = df_timely.drop('company', 'issue', 'state', 'date_sent_to_company')
>>> df_timely.show(10)

# Oversample the minority class (assuming "No" is the minority)
negative_df = df_timely.filter(col("timely") == "No")

# Calculate the fraction to achieve a more balanced ratio
# For example, if you want a 1:1 ratio, set fraction = number of "Yes" instances / number of "No" instances
balanced_ratio = df_timely.filter(col("timely") == "Yes").count() / negative_df.count()
oversampled_negative_df = negative_df.sample(withReplacement=True, fraction=balanced_ratio)

# Combine oversampled negatives with original data (assuming positive is the majority)
sampled_negative_df) / 38][Stage 25]:> (0 + 0) / 38][Stage 26]:> (0 + 0) / 38]

def calculate_weights(data, label_column, weight_column, weight_value):
    total_count = data.count()
    positive_count = data.filter(col(label_column) == "Yes").count() + 0 / 38]
    print(positive_count)
    negative_count = data.filter(col(label_column) == "No").count()
    print(negative_count)
    weight_positive = lit(total_count) / (2 * positive_count * weight_value)
    weight_negative = lit(total_count) / (2 * negative_count * (1 - weight_value))
    return data.withColumn(weight_column, when(col(label_column) == "Yes", weight_positive).otherwise(weight_negative))

# Calculate weights for the minority class
[Stage 25]:> (1 + 4) / 38][Stage 26]:> (0 + 0) / 38]
+-----+-----+-----+-----+-----+-----+-----+-----+
| product|timely|year|month|day|frequency_company|frequency_issue|frequency_state|
+-----+-----+-----+-----+-----+-----+-----+-----+
|Credit reporting,...| Yes|2022| 10| 5|      352148|      589337|      90457|
|Credit reporting,...| Yes|2022| 10| 24|      334034|      742733|      38022|
|Credit reporting,...| Yes|2023| 3| 20|      352148|      742733|      568992|
|Credit reporting,...| Yes|2024| 2| 16|      334034|     1421025|      568992|
|Credit reporting,...| Yes|2023| 4| 14|     1011932|      589337|      588992|
|Credit reporting,...| Yes|2023| 5| 24|     1011932|     1421025|     130214|
|Credit reporting,...| Yes|2022| 7| 20|     1011932|     1421025|      62626|
|Credit reporting,...| Yes|2023| 4| 19|      352148|      742733|     206026|
|Credit reporting,...| Yes|2023| 8| 27|     1011932|      742733|     234009|
|Credit reporting,...| Yes|2023| 4| 19|      352148|     1421025|     590022|
+-----+-----+-----+-----+-----+-----+-----+-----+

only showing top 10 rows

>>>
>>> # Oversample the minority class (assuming "No" is the minority)
... negative_df = df_timely.filter(col("timely") == "No")
>>>
>>> # Calculate the fraction to achieve a more balanced ratio
... # For example, if you want a 1:1 ratio, set fraction = number of "Yes" instances / number of "No" instances
... balanced_ratio = df_timely.filter(col("timely") == "Yes").count() / negative_df.count()
>>> oversampled_negative_df = negative_df.sample(withReplacement=True, fraction=balanced_ratio)
>>>
>>> # Combine oversampled negatives with original data (assuming positive is the majority)
... df_timely = df_timely.filter(col("timely") == "Yes").union(oversampled_negative_df)
>>>
>>>
>>> def calculate_weights(data, label_column, weight_column, weight_value):
...     total_count = data.count()
...     positive_count = data.filter(col(label_column) == "Yes").count()
...     print(positive_count)
...     negative_count = data.filter(col(label_column) == "No").count()
...     print(negative_count)
...     weight_positive = lit(total_count) / (2 * positive_count * weight_value)
...     weight_negative = lit(total_count) / (2 * negative_count * (1 - weight_value))
...     return data.withColumn(weight_column, when(col(label_column) == "Yes", weight_positive).otherwise(weight_negative))
>>>
>>> # Calculate weights for the minority class
... df_timely_balanced = calculate_weights(df_timely, "timely", "weight", 0.3)

```


- **Model Training:**

To train the model using cross-validation, we begin by persisting the balanced DataFrame in memory for efficient access. The data is split into training and testing sets with a 70-30 split. We log the number of rows in each set to verify the split. The pipeline, which includes all preprocessing and modeling stages, is then combined. We define a **BinaryClassificationEvaluator** to assess model performance using the area under the ROC curve (AUC). A parameter grid (**paramGrid**) is set up to tune hyperparameters for the Logistic Regression model, including **regParam** and **elasticNetParam**. We use a **CrossValidator** with three folds to perform cross-validation on the training set. The model fitting process is timed to measure the training duration. The total training time is logged, showing the model's efficiency in terms of time spent.

```
# Model Training - Cross Validation

# -----

from pyspark.storagelevel import StorageLevel

df_timely.persist(StorageLevel.MEMORY_ONLY)

# Split data into training and testing sets

train, test = df_timely.randomSplit([0.7, 0.3], seed=42)

# Print the number of rows in train and test DataFrames

logging.info("Number of rows in train DataFrame:
{}".format(train.count()))

logging.info("Number of rows in test DataFrame:
{}".format(test.count()))

# Combine stages into a pipeline

pipeline = Pipeline(stages=stages)

# Define evaluator

evaluator = BinaryClassificationEvaluator(labelCol="label",
rawPredictionCol="rawPrediction", metricName="areaUnderROC")

#Define paramGrid

paramGrid = ParamGridBuilder() \
```

```
.addGrid(lr.regParam, [0.01, 0.1, 1.0]) \  
  
.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \  
  
.build()  
  
# Create a CrossValidator  
  
cv = CrossValidator(estimator=pipeline, evaluator=evaluator,  
estimatorParamMaps=paramGrid, numFolds=3)  
  
import time  
  
# Start time  
  
start_time = time.time()  
  
# Fit the model with cross-validation on the training set  
  
model = cv.fit(train)  
  
# End time  
  
end_time = time.time()  
  
# Calculate training time  
  
training_time = end_time - start_time  
  
# Calculate minutes and seconds  
  
minutes = int(training_time // 60)  
  
seconds = int(training_time % 60)  
  
logging.info("Training time: %02d:%02d" % (minutes, seconds))
```

```

# Model Training - Cross Validation
... # -----
...
... from pyspark.storagelevel import StorageLevel
... df_timely_balanced.persist(StorageLevel.MEMORY_ONLY)

# Split data into training and testing sets
train, test = df_timely_balanced.randomSplit([0.7, 0.3], seed=42)

DataFrame[product: string, timely: string, year: int, month: int, day: int, frequency_company: bigint, frequency_issue: bigint, frequency_state: bigint, weight: double]
...
... # Split data into training and testing sets
... train, test = df_timely_balanced.randomSplit([0.7, 0.3], seed=42)
...
... # Print the number of rows in train and test DataFrames
... logging.info("Number of rows in train DataFrame: {}".format(train.count()))
logging.info("Number of rows in test DataFrame: {}".format(test.count()))

# Combine stages into a pipeline
pipeline = Pipeline(stages=stages)

# Define evaluator
evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction", metricName="areaUnderROC")

# Define paramGrid
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
    .build()

# Create a CrossValidator
cv = CrossValidator(estimator=pipeline, evaluator=evaluator, estimatorParamMaps=paramGrid, numFolds=3)
Number of rows in train DataFrame: 6853650
... logging.info("Number of rows in test DataFrame: {}".format(test.count()))
Number of rows in test DataFrame: 2936205
...
... # Combine stages into a pipeline
... pipeline = Pipeline(stages=stages)
...
... # Define evaluator
... evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
...
... # Define paramGrid
... paramGrid = ParamGridBuilder() \
...     .addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
...     .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
...     .build()
...
... # Create a CrossValidator
... cv = CrossValidator(estimator=pipeline, evaluator=evaluator, estimatorParamMaps=paramGrid, numFolds=3)
... import time
... # Start time
... start_time = time.time()
...
... # Fit the model with cross-validation on the training set
... model = cv.fit(train)

# End time
end_time = time.time()

```

```

>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>>
>>> logging.info("Training time: %02d:%02d" % (minutes, seconds))
Training time: 31:35

```

- **Model Evaluation – Cross Validation:**

After training the model using cross-validation, we evaluate its performance on the test set. Predictions are made on the test data using the trained model. We calculate key metrics such as true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The area under the ROC curve (AUC) is evaluated to assess the model's overall performance. Precision and recall are computed to measure the model's accuracy and sensitivity, respectively. These metrics are compiled into a DataFrame and rounded for clarity. Finally, the evaluation results are logged and displayed, providing a comprehensive view of the model's effectiveness.

```

# Model Evaluation - Cross Validation

# -----

# Make predictions on the test set (use the actual test set)

predictions = model.transform(test)

tp = float(predictions.filter("prediction == 1.0 AND label ==
1").count())

fp = float(predictions.filter("prediction == 1.0 AND label ==
0").count())

tn = float(predictions.filter("prediction == 0.0 AND label ==
0").count())

fn = float(predictions.filter("prediction == 0.0 AND label ==
1").count())

auc = evaluator.evaluate(predictions)

metrics = spark.createDataFrame([

    ("TP", tp),

    ("FP", fp),

    ("TN", tn),

    ("FN", fn),

    ("Precision", tp / (tp + fp)),

    ("Recall", tp / (tp + fn)),

    ("AUC", auc)

], ["metric", "value"])

metrics = metrics.withColumn("value", round(col("value"), 2))

logging.info("*****CrossValidator Results *****")

metrics.show()

```

```

>>> # Model Evaluation - Cross Validation
... # -----
...
>>> # Make predictions on the test set (use the actual test set)
... predictions = model.transform(test)

tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
fp = float(predictions.filter("p>>>
>>> tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
rediction == 1.0 AND label == 0").count())
tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())

auc = evaluator.evaluate(predictions)

metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),
    ("Precision", tp / (tp + fp)),
    ("Recall", tp / (tp + fn)),
    ("AUC", auc)
], ["metric", "value"])
[Stage 14494:====>                                (32 + 5) / 400]
metrics = metrics.withColumn("value", round(col("value"), 2))

logging.info("*****CrossValidator Results *****")
>>> fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
>>> tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
>>> fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
>>>
>>> auc = evaluator.evaluate(predictions)
>>>
>>> metrics = spark.createDataFrame([
...     ("TP", tp),
...     ("FP", fp),
...     ("TN", tn),
...     ("FN", fn),
...     ("Precision", tp / (tp + fp)),
...     ("Recall", tp / (tp + fn)),
...     ("AUC", auc)
... ], ["metric", "value"])
>>>
>>> metrics = metrics.withColumn("value", round(col("value"), 2))

```

- **Model Evaluation – Cross Validator Results:**

- **True Positives (TP):** 1,412,381
- **False Positives (FP):** 628,801
- **True Negatives (TN):** 839,276
- **False Negatives (FN):** 54,459
- **Precision:** 0.69
- **Recall:** 0.96
- **AUC (Area Under the Curve):** 0.87

```
>>> logging.info("*****CrossValidator Results *****")
*****CrossValidator Results *****
>>> metrics.show()
+-----+
| metric| value|
+-----+
| TP|1412381.0|
| FP| 628801.0|
| TN| 839276.0|
| FN| 54459.0|
| Precision| 0.69|
| Recall| 0.96|
| AUC| 0.87|
+-----+
```

- **Train Validation Split:**

To validate the model's performance, we use the Train-Validation Split method, which involves splitting the data into training and validation sets. We define a **TrainValidationSplit** with the pre-configured pipeline, parameter grid, and evaluator, setting the training ratio to 0.8 (80% training, 20% validation). We then train the model and measure the time taken for this process. The start and end times are recorded to calculate the total training duration, which is logged for reference. This method ensures that the model is well-tuned and capable of generalizing well to new, unseen data.

```
#Train Validation Split

#-----

from pyspark.ml.tuning import TrainValidationSplit
from pyspark.sql.functions import col

# Define TrainValidationSplit

trainval = TrainValidationSplit(estimator=pipeline,

                                estimatorParamMaps=paramGrid,

                                evaluator=evaluator,

                                trainRatio=0.8)

#Training the model and Calculating its time

import time

# Start time

start_time = time.time()

# Fit the cross validator to the training data
```

```
tvModel = trainval.fit(train)

# End time

end_time = time.time()

print("Model trained!")

# Calculate training time

training_time = end_time - start_time

# Calculate minutes and seconds

minutes = int(training_time // 60)

seconds = int(training_time % 60)

logging.info("Training time: %02d:%02d" % (minutes, seconds))
```

```
>>> #Train Validation Split
... #-----
...
>>> from pyspark.ml.tuning import TrainValidationSplit
>>> from pyspark.sql.functions import col
>>>
>>> # Define TrainValidationSplit
... trainval = TrainValidationSplit(estimator=pipeline,
...                                 estimatorParamMaps=paramGrid,
...                                 evaluator=evaluator,
...                                 trainRatio=0.8)
>>>
...
... #Training the model and Calculating its time
... import time
>>>
>>> # Start time
... start_time = time.time()
>>>
>>> # Fit the cross validator to the training data
... tvModel = trainval.fit(train)

# End time
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
24/05/02 08:37:44 WARN BlockManager: Asked to remove block broadcast_3408, which does not exist
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
>>>
>>> # End time
... end_time = time.time()
>>> print("Model trained!")
Model trained!
```

```

>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>>
>>> logging.info("Training time: %02d:%02d" % (minutes, seconds))
Training time: 08:34

```

- **Model Evaluation – Train Validation Split Results:**

After training the model using the Train-Validation Split method, we evaluate its performance on the test data. Predictions are made using the best model from the Train-Validation Split. We calculate the model's accuracy and extract key metrics such as true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Precision and recall are computed to gauge the model's accuracy and sensitivity, respectively. The area under the ROC curve (AUC) is also evaluated to provide a comprehensive measure of model performance. These metrics are compiled into a DataFrame, rounded for clarity, and logged to give a detailed overview of the model's effectiveness.

```

# Model Evaluation - Train Validation Split
# -----

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

#Model Evaluate
accuracy = evaluator.evaluate(predictions)

# Extract TP, FP, TN, FN
tp = float(predicted.filter("prediction == 1.0 AND label ==
1").count())
fp = float(predicted.filter("prediction == 1.0 AND label ==
0").count())
tn = float(predicted.filter("prediction == 0.0 AND label ==
0").count())
fn = float(predicted.filter("prediction == 0.0 AND label ==
1").count())

metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),

```



```

        ("Precision", tp / (tp + fp)),
        ("Recall", tp / (tp + fn)),
        ("AUC", auc)
    ], ["metric", "value"])

metrics = metrics.withColumn("value", round(col("value"), 2))

logging.info("*****TrainValidator Results *****")
metrics.show()

```

- **True Positives (TP):** 1,412,381
- **False Positives (FP):** 628,801
- **True Negatives (TN):** 839,276
- **False Negatives (FN):** 54,459
- **Precision:** 0.69
- **Recall:** 0.96
- **AUC (Area Under the Curve):** 0.87

```

>>> # Model Evaluation - Train Validation Split
... # -----
... # Make predictions on the test data using the best model
... predictions = tvModel.transform(test)

#Model Evaluate
accuracy = evaluator.evaluate(predictions)

# Extract TP, FP, TN, FN
tp = float(predicted.filter("prediction == 1.0").count())
>>> #Model Evaluate
... accuracy = evaluator.evaluate(predictions)
... prediction == 1.0 AND label == 1").count())
fp = float(predicted.filter("prediction == 1.0 AND label == 0").count())
tn = float(predicted.filter("prediction == 0.0 AND label == 0").count())
fn = float(predicted.filter("prediction == 0.0 AND label == 1").count())

metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),
    ("Precision", tp / (tp + fp)),
    ("Recall", tp / (tp + fn)),
    ("AUC", auc)
], ["metric", "value"])

metrics = metrics.withColumn("value", round(col("value"), 2))    (35 + 5) / 400]
tag: 17224:=====                                           (51 + 5) / 400]
logging.info("*****TrainValidator Results *****")
metrics.show()
>>>
>>> # Extract TP, FP, TN, FN
... tp = float(predicted.filter("prediction == 1.0 AND label == 1").count())
>>> fp = float(predicted.filter("prediction == 1.0 AND label == 0").count())
>>> tn = float(predicted.filter("prediction == 0.0 AND label == 0").count())
>>> fn = float(predicted.filter("prediction == 0.0 AND label == 1").count())
>>>
>>> metrics = spark.createDataFrame([
...     ("TP", tp),
...     ("FP", fp),
...     ("TN", tn),
...     ("FN", fn),
...     ("Precision", tp / (tp + fp)),
...     ("Recall", tp / (tp + fn)),
...     ("AUC", auc)
... ], ["metric", "value"])
>>>
>>> metrics = metrics.withColumn("value", round(col("value"), 2))

```

```

>>>
... logging.info("*****TrainValidator Results *****")
*****TrainValidator Results *****
>>> metrics.show()
+-----+
| metric|    value|
+-----+
|      TP|1412381.0|
|      FP| 628801.0|
|      TN| 839276.0|
|      FN| 54459.0|
| Precision|    0.69|
|  Recall|    0.96|
|      AUC|    0.87|
+-----+

```

E. Timely Response using Support Vector Machines:

To set up the environment for building an SVM model with PySpark, we first configure logging to display informational messages, aiding in tracking the execution flow and debugging. We then import essential PySpark modules including **SparkSession** for managing Spark applications, **VectorAssembler** and **StringIndexer** for feature transformation, and **LinearSVC** for the Support Vector Machine classifier. We also import various functions and types for data manipulation, such as extracting year, month, and day from dates, and **BinaryClassificationEvaluator** for model evaluation. **CrossValidator** and **ParamGridBuilder** are imported for hyperparameter tuning, while **Pipeline** is used to streamline the workflow. If running in a PySpark CLI environment, we initialize **SparkContext** and **SparkSession** to manage Spark operations effectively.

```

import logging

logging.basicConfig(level=logging.INFO, format='%(message)s')

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler, StringIndexer

from pyspark.ml.classification import LinearSVC

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.sql.types import StringType, TimestampType

```

```

from pyspark.sql.functions import year, month, dayofmonth, count, col,
when, lit

from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from pyspark.ml import Pipeline

from pyspark.context import SparkContext

from pyspark.sql.session import SparkSession

import pandas as pd

# PYSPARK_CLI = True

# if PYSPARK_CLI:

    # sc = SparkContext.getOrCreate()

    # spark = SparkSession(sc)

```

- **Data Preprocessing:**

In the data preprocessing step, we start by reading the JSON file '**complaints.json**' into a DataFrame named **raw_complaints**. We select the necessary columns—**company**, **product**, **timely**, **issue**, **state**, and **date_sent_to_company**—while filtering out any corrupt records. To ensure clean data, we use **zipWithIndex** to index the rows and then remove the first row, which might contain header information or other unsuitable data. Next, we filter out rows where the **timely** column is empty. We convert the **date_sent_to_company** column to **TimestampType** to facilitate date-based transformations. Finally, we extract the year, month, and day from the **date_sent_to_company** column to create additional features for the model.

```

# Data Preprocessing

# -----

# Select necessary columns and drop corrupt records

```

```

# Read the JSON file 'complaints.json' into a DataFrame named
'raw_complaints'

raw_complaints = spark.read.json('5560_Complaints_DS/complaints.json')

# Select necessary columns and drop corrupt records

complaint_df = raw_complaints.select('company', 'product', 'timely',
'issue', 'state',
'date_sent_to_company').filter(raw_complaints['_corrupt_record'].isNul
l())

# ZipWithIndex and remove first row

complaint_df = complaint_df.rdd.zipWithIndex().filter(lambda x: x[1] >
0).map(lambda x: x[0]).toDF()

# Drop rows with timely=None

df_timely_initial = complaint_df.filter(col("timely") != "")

# Cast date_sent_to_company to TimestampType

df_timely_initial =
df_timely_initial.withColumn("date_sent_to_company",
col("date_sent_to_company").cast(TimestampType()))

# Extract year, month, and day from date_sent_to_company

df_timely_initial = df_timely_initial.withColumn("year",
year("date_sent_to_company")) \

                                .withColumn("month",
month("date_sent_to_company")) \

                                .withColumn("day",
dayofmonth("date_sent_to_company"))

```

```

>> import logging
>> logging.basicConfig(level=logging.INFO, format='%n(message)s')
>>
>> from pyspark.sql import SparkSession
>> from pyspark.ml.feature import VectorAssembler, StringIndexer
>> from pyspark.ml.classification import LinearSVC
>> from pyspark.ml.evaluation import BinaryClassificationEvaluator
>> from pyspark.ml.classification import LinearSVC
>> from pyspark.ml.evaluation import BinaryClassificationEvaluator
>> from pyspark.sql.types import StringType, TimestampType
>> from pyspark.sql.functions import year, month, dayofmonth, count, col, when, lit
>> from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
>> from pyspark.ml import Pipeline
>> from pyspark.context import SparkContext
>> from pyspark.sql.session import SparkSession
>> import pandas as pd
>> # Data Preprocessing
.. # -----
..
>> # Select necessary columns and drop corrupt records
..
>> # Read the JSON file 'complaints.json' into a DataFrame named 'raw_complaints'
.. raw_complaints = spark.read.json('3180_Complaints_DS/complaints.json')
..
>> # Select necessary columns and drop corrupt records
complaint_df = raw_complaints.select('company', 'product', 'timely', 'issue', 'state', 'date_sent_to_company').filter(raw_complaints['_corrupt_record'].isNull())
..
>> # ZipWithIndex and remove first row
complaint_df = complaint_df.rdd.zipWithIndex().filter(lambda x: x[1] > 0).map(lambda x: x[0]).toDF()
..
>> # Drop rows with timely=None
df_timely_initial = complaint_df.filter(col("timely") != "")
..
>> # Cast date_sent_to_company to TimestampType
df_timely_initial = df_timely_initial.withColumn("date_sent_to_company", col("date_sent_to_company").cast(TimestampType()))
..
>> # Extract year, month, and day from date_sent_to_company
df_timely_initial = df_timely_initial.withColumn("year", year("date_sent_to_company")) \
    .withColumn("month", month("date_sent_to_company")) \
    .withColumn("day", dayofmonth("date_sent_to_company"))
..
>> # Select necessary columns and drop corrupt records
.. complaint_df = raw_complaints.select('company', 'product', 'timely', 'issue', 'state', 'date_sent_to_company').filter(raw_complaints['_corrupt_record'].isNull())
>>
>> # ZipWithIndex and remove first row
.. complaint_df = complaint_df.rdd.zipWithIndex().filter(lambda x: x[1] > 0).map(lambda x: x[0]).toDF()
>>
>> # Drop rows with timely=None
.. df_timely_initial = complaint_df.filter(col("timely") != "")
>>
>> # Cast date_sent_to_company to TimestampType
.. df_timely_initial = df_timely_initial.withColumn("date_sent_to_company", col("date_sent_to_company").cast(TimestampType()))
>>
>> # Extract year, month, and day from date_sent_to_company
.. df_timely_initial = df_timely_initial.withColumn("year", year("date_sent_to_company")) \
..     .withColumn("month", month("date_sent_to_company")) \
..     .withColumn("day", dayofmonth("date_sent_to_company"))
..

```

- **Feature Engineering:**

In the feature engineering step, we enhance the dataset by calculating the frequency of occurrences for specific features and integrating these frequencies as new columns. First, we calculate the frequency of each company by grouping the data by the **company** column and counting the number of records for each company. This frequency information is then joined with the original DataFrame on the **company** column. Next, we calculate the frequency of each issue by grouping the data by the **issue** column and counting the records. This frequency data is also joined with the DataFrame on the **issue** column. Similarly, we calculate the frequency of each state by grouping the data by the **state** column and counting the number of records. This frequency information is joined with the DataFrame on the **state** column. These additional frequency features provide the model with valuable context about the relative occurrences of companies, issues, and states, potentially improving its predictive performance.

```
# Feature Engineering
```

```
# -----
```

```
# Calculate the frequency of each company
```

```
company_frequency =
df_timely_initial.groupBy("company").agg(count("*").alias("frequency_c
ompany"))

# Join the frequency DataFrame with the original DataFrame on the
company column

df_timely = df_timely_initial.join(company_frequency, on="company",
how="left")

# Calculate the frequency of each issue

issue_frequency =
df_timely_initial.groupBy("issue").agg(count("*").alias("frequency_iss
ue"))

# Join the issue frequency DataFrame with the existing DataFrame on the
issue column

df_timely = df_timely.join(issue_frequency, on="issue", how="left")

# Calculate the frequency of each state

state_frequency =
df_timely_initial.groupBy("state").agg(count("*").alias("frequency_sta
te"))

# Join the state frequency DataFrame with the existing DataFrame on the
state column

df_timely = df_timely.join(state_frequency, on="state", how="left")
```

```

>>> # Feature Engineering
... # -----
...
>>> # Calculate the frequency of each company
... company_frequency = df_timely_initial.groupBy("company").agg(count("*").alias("frequency_company"))

>>>
>>> # Join the frequency DataFrame with the original DataFrame on the company column
... df_timely = df_timely_initial.join(company_frequency, on="company", how="left")

>>>
>>> # Calculate the frequency of each issue
... issue_frequency = df_timely_initial.groupBy("issue").agg(count("*").alias("frequency_issue"))

>>>
>>> # Join the issue frequency DataFrame with the existing DataFrame on the issue column
... df_timely = df_timely.join(issue_frequency, on="issue", how="left")

>>>
>>> # Calculate the frequency of each state
... state_frequency = df_timely_initial.groupBy("state").agg(count("*").alias("frequency_state"))

>>>
>>> # Join the state frequency DataFrame with the existing DataFrame on the state column
... df_timely = df_timely.join(state_frequency, on="state", how="left")

```

- **Prepare Pipeline:**

To prepare the machine learning pipeline, we start by defining the features for the model, which include **product**, **frequency_company**, **frequency_issue**, **frequency_state**, **year**, **month**, and **day**. We create a list of stages for the pipeline, beginning with string indexing for the **product** feature to convert it into a numerical format using **StringIndexer**. Next, we use **VectorAssembler** to combine the indexed **product**, frequency features, and date-related features into a single feature vector named **assembledFeatures**. We then apply string indexing to the **timely** label to convert it into a numerical format. Finally, we add a Linear Support Vector Machine (SVM) model to the pipeline. These stages are appended to the pipeline, ensuring a streamlined and consistent workflow for data preprocessing and model training.

```

# Prepare Pipeline

# -----

# Define features_for_model

features_for_model = ["product", "frequency_company",
"frequency_issue", "frequency_state"]

# Create a list of stages for the pipeline

stages = []

# Stage 1: String indexing for categorical features

```

```

indexer_product = StringIndexer(inputCol="product",
outputCol="indexed_product")

stages.append(indexer_product)

# Stage 2: Assemble features

assembler = VectorAssembler(inputCols=["indexed_product",
"frequency_company", "frequency_issue", "frequency_state", "year",
"month", "day"], outputCol="assembledFeatures")

stages.append(assembler)

# Stage 3: String indexing for label

label_indexer = StringIndexer(inputCol="timely", outputCol="label")

stages.append(label_indexer)

# Stage 4: Linear SVM model

svm = LinearSVC(featuresCol="assembledFeatures", labelCol="label")

stages.append(svm)

```

```

>>> # Prepare Pipeline
... # -----
...
>>> # Define features for model
... features_for_model = ["product", "frequency_company", "frequency_issue", "frequency_state"]
...
>>> # Create a list of stages for the pipeline
... stages = []
...
>>> # Stage 1: String indexing for categorical features
... indexer_product = StringIndexer(inputCol="product", outputCol="indexed_product")
... stages.append(indexer_product)
...
>>> # Stage 2: Assemble features
... assembler = VectorAssembler(inputCols=["indexed_product", "frequency_company", "frequency_issue", "frequency_state", "year", "month", "day"], outputCol="assembledFeatures")
... stages.append(assembler)
...
>>> # Stage 3: String indexing for label
... label_indexer = StringIndexer(inputCol="timely", outputCol="label")
... stages.append(label_indexer)
...
>>> # Stage 4: Linear SVM model
... svm = LinearSVC(featuresCol="assembledFeatures", labelCol="label")
... stages.append(svm)
>>> stages.append(svm)

```

- **Balancing the Data by Oversampling Minority Class (timely = No):**

To address the class imbalance in the dataset, where the minority class is **timely = No**, we perform oversampling. We start by dropping unnecessary columns such as **company**, **issue**, **state**, and **date_sent_to_company** from the DataFrame. We then isolate the minority class (**timely = No**) into a

separate DataFrame. To achieve a balanced dataset, we calculate the ratio of the majority class (**timely = Yes**) to the minority class. Using this ratio, we oversample the minority class by creating a sample with replacement, ensuring the number of **No** instances matches the desired balanced ratio. Finally, we combine the oversampled minority class with the majority class, resulting in a balanced dataset that helps the model learn effectively from both classes.

```
# Balancing the data by Oversampling minority class (timely = No)

# -----

# Dropping additional columns

df_timely = df_timely.drop('company', 'issue', 'state',
'date_sent_to_company')

# Oversample the minority class

negative_df = df_timely.filter(col("timely") == "No")

balanced_ratio = df_timely.filter(col("timely") == "Yes").count() /
negative_df.count()

oversampled_negative_df = negative_df.sample(withReplacement=True,
fraction=balanced_ratio)

df_timely = df_timely.filter(col("timely") ==
"Yes").union(oversampled_negative_df)
```

```
>>> # Balancing the data by Oversampling minority class (timely = No)
... # -----
...
>>> # Dropping additional columns
... df_timely = df_timely.drop('company', 'issue', 'state', 'date_sent_to_company')
>>>
>>> # Oversample the minority class
... negative_df = df_timely.filter(col("timely") == "No")
>>> balanced_ratio = df_timely.filter(col("timely") == "Yes").count() / negative_df.count()
oversampled_negative_df = negative_df.sample(withReplacement=True, fraction=balanced_ratio)
df_timely = df_timely.filter(col("timely") == "Yes").union(oversampled_negative_df)
>>> oversampled_negative_df = negative_df.sample(withReplacement=True, fraction=balanced_ratio)
>>> df_timely = df_timely.filter(col("timely") == "Yes").union(oversampled_negative_df)
```

- **Model Training:**

To train the model using cross-validation, we start by persisting the balanced DataFrame in memory to ensure efficient access during training. We split the data into training and testing sets with a 70-30 split and log the number of rows in each set to verify the split. The stages defined earlier are combined into a pipeline, and we define a **BinaryClassificationEvaluator** to assess model performance using the area under the ROC curve (AUC). We set up a parameter grid (**paramGrid**) to tune the hyperparameters of the Linear SVM model, including **regParam** and **maxIter**. A **CrossValidator** with three folds is created to perform cross-validation on the training set. The model fitting process is timed to measure the training duration, which is then logged to provide insight into the model's training efficiency.

```
# Model Training - Cross Validation

# -----

# Persist the DataFrame in memory

from pyspark.storagelevel import StorageLevel

df_timely.persist(StorageLevel.MEMORY_ONLY)

# Split data into training and testing sets

train, test = df_timely.randomSplit([0.7, 0.3], seed=42)

# Print the number of rows in train and test DataFrames

logging.info("Number of rows in train DataFrame:
{}".format(train.count()))

logging.info("Number of rows in test DataFrame:
{}".format(test.count()))

# Combine stages into a pipeline

pipeline = Pipeline(stages=stages)

# Define evaluator
```

```
evaluator = BinaryClassificationEvaluator(labelCol="label",
rawPredictionCol="rawPrediction", metricName="areaUnderROC")

# Define paramGrid

paramGrid = ParamGridBuilder() \

    .addGrid(svm.regParam, [0.01, 0.1, 1.0]) \

    .addGrid(svm.maxIter, [5, 10, 15]) \

    .build()

# Create a CrossValidator

cv = CrossValidator(estimator=pipeline, evaluator=evaluator,
estimatorParamMaps=paramGrid, numFolds=3)

# Measure training time

import time

start_time = time.time()

model = cv.fit(train)

end_time = time.time()

training_time = end_time - start_time

minutes = int(training_time // 60)

seconds = int(training_time % 60)

logging.info("Training time: %02d:%02d" % (minutes, seconds))
```

```

>>> # Model Training - Cross Validation
... #
... # Persist the DataFrame in memory
... from pyspark.storagelevel import StorageLevel
... df_timely.persist(StorageLevel.MEMORY_ONLY)
DataFrame[product: string, timely: string, year: int, month: int, day: int, frequency_company: bigint, frequency_issue: bigint, frequency_state: bigint]
...
>>> # Split data into training and testing sets
... train, test = df_timely.randomSplit([0.7, 0.3], seed=42)
...
>>> # Print the number of rows in train and test DataFrames
... logging.info("Number of rows in train DataFrame: {}".format(train.count()))
Number of rows in train DataFrame: 4449507
>>> logging.info("Number of rows in test DataFrame: {}".format(test.count()))
Number of rows in test DataFrame: 2934454
...
>>> # Combine stages into a pipeline
... pipeline = Pipeline(stages=stages)
...
>>> # Define evaluator
... evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
...
>>> # Define paramGrid
... paramGrid = ParamGridBuilder() \
...     .addGrid(svm.epparams, [0.01, 0.1, 1.0]) \
...     .addGrid(svm.maxiter, [1, 10, 15]) \
...     .build()
...
>>> # Create a CrossValidator
... cv = CrossValidator(estimator=pipeline, evaluator=evaluator, estimatorParamMaps=paramGrid, numFolds=1)
...
>>> # Measure training time
... import time
... start_time = time.time()
>>> model = cv.fit(train)
Closing down clientserver connection
Closing down clientserver connection

>>> end_time = time.time()
>>> training_time = end_time - start_time
>>> minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>> logging.info("Training time: %02d:%02d" % (minutes, seconds))
Training time: 47:21

```

- **Model Evaluation – Cross Validation:**

After training the model using cross-validation, we evaluate its performance on the test set. Predictions are made using the trained model, and key metrics such as true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) are calculated. The area under the ROC curve (AUC) is also evaluated to assess the model's overall performance. These metrics are compiled into a DataFrame, and precision and recall are rounded for clarity. Finally, the evaluation results are logged and displayed to provide a comprehensive view of the model's effectiveness.

```

# Model Evaluation - Cross Validation

# -----

# Make predictions on the test set

predictions = model.transform(test)

# Calculate metrics

```

```
tp = float(predictions.filter("prediction == 1.0 AND label ==
1").count())

fp = float(predictions.filter("prediction == 1.0 AND label ==
0").count())

tn = float(predictions.filter("prediction == 0.0 AND label ==
0").count())

fn = float(predictions.filter("prediction == 0.0 AND label ==
1").count())

auc = evaluator.evaluate(predictions)

# Create DataFrame with evaluation metrics

metrics = spark.createDataFrame([

    ("TP", tp),

    ("FP", fp),

    ("TN", tn),

    ("FN", fn),

    ("Precision", round(tp / (tp + fp), 2)),

    ("Recall", round(tp / (tp + fn), 2)),

    ("AUC", round(auc, 2))

], ["metric", "value"])

# Print evaluation metrics

# print("*****CrossValidator Results *****")

logging.info("*****CrossValidator Results *****")
```

```
metrics.show()
```

```
>>> # Model Evaluation - Cross Validation
... # -----
...
>>> # Make predictions on the test set
... predictions = model.transform(test)

# Calculate metrics
tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
>>>
>>> # Calculate metrics
... tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
... fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
auc = evaluator.evaluate(predictions)

# Create DataFrame with evaluation metrics
metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),
    ("Precision", round(tp / (tp + fp), 2)),
    ("Recall", round(tp / (tp + fn), 2)),
    ("AUC", round(auc, 2))
], ["metric", "value"])

# Print evaluation metrics
# print("*****CrossValidator Results *****")
logging.info("*****CrossValidator Results *****")
>>> fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
>>> tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
>>> fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
>>> auc = evaluator.evaluate(predictions)
>>>
>>> # Create DataFrame with evaluation metrics
... metrics = spark.createDataFrame([
...     ("TP", tp),
...     ("FP", fp),
...     ("TN", tn),
...     ("FN", fn),
...     ("Precision", round(tp / (tp + fp), 2)),
...     ("Recall", round(tp / (tp + fn), 2)),
...     ("AUC", round(auc, 2))
... ], ["metric", "value"])
>>>
>>> # Print evaluation metrics
... # print("*****CrossValidator Results *****")
... logging.info("*****CrossValidator Results *****")
```

- **Model Evaluation – Cross Validator Results:**

- **True Positives (TP):** 1,412,410
- **False Positives (FP):** 628,801
- **True Negatives (TN):** 839,276
- **False Negatives (FN):** 53,967

- **Precision:** 0.69
- **Recall:** 0.96
- **AUC (Area Under the Curve):** 0.87

```
>>> metrics.show()
+-----+
| metric|   value|
+-----+
|      TP|1412410.0|
|      FP| 628801.0|
|      TN| 839276.0|
|      FN| 53967.0|
| Precision|    0.69|
|  Recall|    0.96|
|      AUC|    0.87|
+-----+
```

- **Train Validation Split:**

To validate the model's performance using the Train-Validation Split method, we define a **TrainValidationSplit** with the previously set up pipeline, parameter grid, and evaluator, specifying a training ratio of 0.8 (80% training data, 20% validation data). The model is trained by fitting the **TrainValidationSplit** to the training data, and the training time is measured. We record the start and end times of the training process to calculate the total training duration, which is then logged for reference. This approach helps ensure that the model is robust and can generalize well to unseen data.

```
#Train Validation Split

#-----

from pyspark.ml.tuning import TrainValidationSplit

from pyspark.sql.functions import col

# Define TrainValidationSplit

trainval = TrainValidationSplit(estimator=pipeline,

                                estimatorParamMaps=paramGrid,

                                evaluator=evaluator,
```

```
trainRatio=0.8)

#Training the model and Calculating its time

import time

# Start time

start_time = time.time()

# Fit the cross validator to the training data

tvModel = trainval.fit(train)

# End time

end_time = time.time()

print("Model trained!")

# Calculate training time

training_time = end_time - start_time

# Calculate minutes and seconds

minutes = int(training_time // 60)

seconds = int(training_time % 60)

# Print training time

logging.info("Training time: %02d:%02d" % (minutes, seconds))
```



```

...     .build()
>>> #Train Validation Split
... #-----
...
>>> from pyspark.ml.tuning import TrainValidationSplit
>>> from pyspark.sql.functions import col
>>>
>>>
>>> # Define TrainValidationSplit
... trainval = TrainValidationSplit(estimator=pipeline,
...                                 estimatorParamMaps=paramGrid,
...                                 evaluator=evaluator,
...                                 trainRatio=0.8)
>>>
...
>>> #Training the model and Calculating its time
... import time
>>>
>>> # Start time
... start_time = time.time()
>>>
>>> # Fit the cross validator to the training data
... tvModel = trainval.fit(train)

# End time
[Stage 92:>                                     (0 + 5) / 400]
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
Closing down clientserver connection
>>>
>>> # End time
... end_time = time.time()
>>> print("Model trained!")
Model trained!
>>>
>>>
>>> # Calculate training time
... training_time = end_time - start_time
>>>
>>> # Calculate minutes and seconds
... minutes = int(training_time // 60)
>>> seconds = int(training_time % 60)
>>>

>>> # Print training time
... logging.info("Training time: %02d:%02d" % (minutes, seconds))
Training time: 18:19

```

- **Model Evaluation – Train Validation Split:**

After training the model using the Train-Validation Split method, we evaluate its performance on the test data. Predictions are made using the best model obtained from the Train-Validation Split. We evaluate the model's accuracy and extract key metrics such as true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Precision and recall are calculated to assess the model's accuracy and sensitivity, respectively. These metrics are compiled into a DataFrame, rounded for clarity, and logged to provide a detailed overview of the model's effectiveness.

```
# Model Evaluation - Train Validation Split

# -----

# Make predictions on the test data using the best model

predictions = tvModel.transform(test)

#Model Evaluate

accuracy = evaluator.evaluate(predictions)

# Extract TP, FP, TN, FN

tp = float(predictions.filter("prediction == 1.0 AND label ==
1").count())

fp = float(predictions.filter("prediction == 1.0 AND label ==
0").count())

tn = float(predictions.filter("prediction == 0.0 AND label ==
0").count())
```

```
fn = float(predictions.filter("prediction == 0.0 AND label ==
1").count())

# Calculate precision and recall

precision = round(tp / (tp + fp), 2)

recall = round(tp / (tp + fn), 2)

# Create DataFrame with evaluation metrics

metrics = spark.createDataFrame([

    ("TP", tp),

    ("FP", fp),

    ("TN", tn),

    ("FN", fn),

    ("Precision", precision),

    ("Recall", recall),

    ("AUC", round(accuracy, 2))

], ["metric", "value"])

logging.info("*****TrainValidator Results *****")

metrics.show()
```

```

>>> # Model Evaluation - Train Validation Split
... # -----
...
>>> # Make predictions on the test data using the best model
... predictions = tvModel.transform(test)
#Model Evaluate
accuracy = evaluator.evaluate(predictions)

# Extract TP, FP, TN, FN
tp = float(predictions.filter("p>>>
>>> #Model Evaluate
... accuracy = evaluator.evaluate(predictions)
prediction == 1.0 AND label == 1").count())
fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())

# Calculate precision and recall
precision = round(tp / (tp + fp), 2)
recall = round(tp / (tp + fn), 2)

# Create DataFrame with evaluation metrics
metrics = spark.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),=====> (74 + 5) / 400]
    ("Precision", precision),
    ("Recall", recall),
    ("AUC", round(accuracy, 2))
])

>>>
>>> # Extract TP, FP, TN, FN
... tp = float(predictions.filter("prediction == 1.0 AND label == 1").count())
>>> fp = float(predictions.filter("prediction == 1.0 AND label == 0").count())
>>> tn = float(predictions.filter("prediction == 0.0 AND label == 0").count())
>>> fn = float(predictions.filter("prediction == 0.0 AND label == 1").count())
>>>

>>> # Calculate precision and recall
... precision = round(tp / (tp + fp), 2)
>>> recall = round(tp / (tp + fn), 2)
>>>

>>> # Create DataFrame with evaluation metrics
... metrics = spark.createDataFrame([
...     ("TP", tp),
...     ("FP", fp),
...     ("TN", tn),
...     ("FN", fn),
...     ("Precision", precision),
...     ("Recall", recall),
...     ("AUC", round(accuracy, 2))
... ], ["metric", "value"])
>>>
... logging.info("*****TrainValidator Results *****")
*****TrainValidator Results *****
>>> metrics.show()

```

- **Model Evaluation – Train Validation Split Results:**

- **True Positives (TP):** 839,276
- **False Positives (FP):** 54,276
- **True Negatives (TN):** 1,413,588
- **False Negatives (FN):** 628,801
- **Precision:** 0.94
- **Recall:** 0.57
- **AUC (Area Under the Curve):** 0.87

```

>>> ... logging.info("*****TrainValidator Results *****")
*****TrainValidator Results *****
>>> metrics.show()
+-----+-----+
| metric | value |
+-----+-----+
| TP      | 839276.0 |
| FP      | 54276.0 |
| TN      | 1413588.0 |
| FN      | 628801.0 |
| Precision | 0.94 |
| Recall   | 0.57 |
| AUC      | 0.87 |
+-----+-----+

```

F. Complaints Narrative using Latent Dirichlet Allocation (LDA):

To set up the environment for building a machine learning model with PySpark, we start by configuring logging to display informational messages, aiding in tracking the execution flow and debugging. We import essential PySpark modules and functions, including **year**, **month**, **dayofmonth**, **count**, **col**, **when**, **lit**, **expr**, **explode**, **udf**, and **trim** for various data manipulation tasks. We also import data types such as **StringType**, **TimestampType**, **ArrayType**, and **DoubleType**. For feature engineering, we utilize **Tokenizer**, **StopWordsRemover**, **CountVectorizer**, **IDF**, and **RegexTokenizer**. Additionally, we include clustering algorithms like **LDA** and **BisectingKMeans**, and tools for working with vectors, such as **Vectors** and **SparseVector**. Finally, we initialize a **SparkSession** to manage Spark operations, ensuring a streamlined and efficient workflow for machine learning tasks.

```

# Import necessary functions and types from PySpark
import logging
logging.basicConfig(level=logging.INFO, format='%(message)s')
from pyspark.sql.functions import year, month, dayofmonth, count, col,
when, lit, expr, explode, udf, trim
from pyspark.sql.types import StringType, TimestampType, ArrayType,
DoubleType
from pyspark.ml.feature import Tokenizer, StopWordsRemover,
CountVectorizer, IDF, RegexTokenizer
from pyspark.ml.clustering import LDA, BisectingKMeans
from pyspark.ml.linalg import Vectors, SparseVector
from pyspark.sql.functions import monotonically_increasing_id
from pyspark.context import SparkContext
from pyspark.sql import SparkSession
import re

```

```
# Set up Spark session
PYSPARK_CLI = True
if PYSPARK_CLI:
    sc = SparkContext.getOrCreate()
    spark = SparkSession(sc)
```

- **Data Preprocessing:**

In the data preprocessing step, we begin by reading the JSON file '**complaints.json**' into a DataFrame named **raw_complaints**. We select the necessary columns, specifically '**complaint_what_happened**' and '**date_received**', while filtering out any corrupt records and rows with empty or null complaint text. We then cast the '**date_received**' column to **TimestampType** and extract the year, month, and day from this date, dropping the original date column afterwards. To prepare the text data for analysis, we define a function to clean the text, removing unwanted characters and stop words. This function is applied to the '**complaint_what_happened**' column using a user-defined function (UDF). The cleaned text is then processed to remove stop words using **StopWordsRemover**. Finally, we convert the cleaned text into numerical features using **CountVectorizer** and **IDF** to prepare the data for modeling.

```
# Data Pre-processing

# -----

# Read the JSON file 'complaints.json' into a DataFrame named
# 'raw_complaints'
raw_complaints = spark.read.json('project/complaints.json')

# Select necessary columns and drop corrupt records
df_nlp = raw_complaints.select('complaint_what_happened',
                                'date_received') \
    .filter(raw_complaints['_corrupt_record'].isNull()) \
    .filter(trim(col('complaint_what_happened')) != '') \
    .filter(col('complaint_what_happened').isNotNull())

# Cast date_received to TimestampType
df_nlp = df_nlp.withColumn("date_received",
                            col("date_received").cast(TimestampType()))

# Extract year, month, and day from date_received
df_nlp = df_nlp.withColumn("year", year("date_received")) \
    .withColumn("month", month("date_received")) \
```

```

.withColumn("day", dayofmonth("date_received"))
df_nlp = df_nlp.drop("date_received")

# Define a function to clean text and remove stop words
def clean_text(text):

# Define regular expression pattern to match "xx" followed by one or
more "x"s
xx_pattern = r'\bxxx+\b|\bXX+\b|\bxx+\b'

# Combine the xx_pattern with the existing pattern
pattern = fr'{xx_pattern}|[^a-zA-Z\s]|(?<![a-zA-Z])\b[a-zA-Z]\b'

# Replace matched patterns with a space
cleaned_text = re.sub(pattern, ' ', text.lower())

# Split text into words and filter out empty strings and words with
less than 2 characters
cleaned_words = [word for word in cleaned_text.split() if len(word) >
1]
return cleaned_words

# Apply text cleaning function to complaint_what_happened column
clean_text_udf = udf(clean_text, ArrayType(StringType()))
df_cleaned = df_nlp.withColumn("cleaned_complaint",
clean_text_udf("complaint_what_happened"))

# Define stop words
stop_words = StopWordsRemover().getStopWords()

# Define and apply stop words remover
stop_words_remover = StopWordsRemover(inputCol="cleaned_complaint",
outputCol="cleaned_complaint_without_stopwords", stopWords=stop_words)
df_cleaned = stop_words_remover.transform(df_cleaned)
df_cleaned = df_cleaned.drop("cleaned_complaint",
"complaint_what_happened")

# Convert words into numerical features using CountVectorizer and IDF

# Initialize CountVectorizer
cv = CountVectorizer(inputCol="cleaned_complaint_without_stopwords",
outputCol="raw_features")

```



```

>>> from pyspark.ml.feature import StopWordsRemover
>>> import re
>>> # Define a function to clean text and remove stop words
... def clean_text(text):
...     # Define regular expression pattern to match "xx" followed by one or more "x"s
...     xx_pattern = r'\bxxxx+\b|\bxx+\b|\bxxx+\b'
...     # Combine the xx_pattern with the existing pattern
...     pattern = fr'{xx_pattern}|[a-zA-Z\s]{{7,}}|[a-zA-Z]]\b[a-zA-Z]\b'
...     # replace matched patterns with a space
...     cleaned_text = re.sub(pattern, ' ', text.lower())
...     # split text into words and filter out empty strings and words with less than 2 characters
...     cleaned_words = [word for word in cleaned_text.split() if len(word) > 1]
...     return cleaned_words
...
>>> # Apply text cleaning function to complaint_what_happened column
... clean_text_udf = udf(clean_text, ArrayType(StringType()))
>>> df_cleaned = df_nlp.withColumn("cleaned_complaint", clean_text_udf("complaint_what_happened"))
>>> # Define stop words
... stop_words = StopWordsRemover().getStopWords()

# Define and apply stop words remover
>>>
>>> # Define and apply stop words remover
... stop_words_remover = StopWordsRemover(inputCol="cleaned_complaint", outputCol="cleaned_complaint_without_stopwords", stopWords=stop_words)
df_cleaned = stop_words_remover.>>> df_cleaned = stop_words_remover.transform(df_cleaned)
>>> df_cleaned = df_cleaned.drop("cleaned_complaint", "complaint_what_happened")

```

```

>>> # Convert words into numerical features using CountVectorizer and IDF
...
>>> cv = CountVectorizer(inputCol="cleaned_complaint_without_stopwords", outputCol="raw_features")
>>> cv_model = cv.fit(df_cleaned)
>>> df_featurized = cv_model.transform(df_cleaned)
>>> print("Number of rows: ", df_featurized.count())
print("Number of columns: ", len(df_featurized.columns))

idf = IDF(inputCol="raw_features", outputCol="features")
idf_model = idf.fit(df_featurized)
Number of rows: 1752176
>>> print("Number of columns: ", len(df_featurized.columns))
Number of columns: 5
>>>
>>> idf = IDF(inputCol="raw_features", outputCol="features")
>>> idf_model = idf.fit(df_featurized)
24/05/06 03:16:50 WARN DAGScheduler: Broadcasting large task binary with size 2.3 MiB
24/05/06 03:22:58 WARN DAGScheduler: Broadcasting large task binary with size 2.3 MiB
>>> df_features = idf_model.transform(df_featurized)

```

- **Model Training:**

To train the LDA model, we begin by setting the number of topics (**num_topics**) to 25, though this parameter can be adjusted based on the specific requirements of the analysis. We initialize the LDA model with the specified number of topics, a random seed for reproducibility, and the expectation-maximization (EM) optimizer, using the **features** column as input. We then measure the training time by recording the start and end times of the model fitting process. The total training time is calculated in minutes and seconds and logged for reference. This process ensures that the model is trained efficiently and ready for further evaluation and interpretation.

```

# Model Training

# -----

# Train LDA model
num_topics = 25 # adjust this parameter

```

```
lda = LDA(k=num_topics, seed=123, optimizer="em",
featuresCol="features")

# Measure training time
import time
start_time = time.time()
lda_model = lda.fit(df_features)
end_time = time.time()
training_time = end_time - start_time
minutes = int(training_time // 60)
seconds = int(training_time % 60)
logging.info("Training time: %02d:%02d" % (minutes, seconds))
```

```
>>> # Train LDA mode
...
>>> num_topics = 25 # adjust this parameter
>>> lda = LDA(k=num_topics, seed=123, optimizer="em", featuresCol="features")
>>>
>>> lda_model = lda.fit(df_features)
```

- **Evaluation:**

For the evaluation step, we start by describing the topics identified by the LDA model using **lda_model.describeTopics()** and display the top 25 topics. To make the topics interpretable, we extract the vocabulary from the **CountVectorizer** model and broadcast it for efficient mapping. We define a function **map_termID_to_Word** to map term IDs to their corresponding words in the vocabulary. This function is applied to the term indices in the LDA topics using a user-defined function (UDF). The resulting DataFrame **ldatopics_mapped** contains the topics with their descriptive words, which we display. Additionally, we assign topics to each complaint by transforming the feature DataFrame with the LDA model and display the resulting topic distributions along with the month and day. This comprehensive evaluation helps in understanding the distribution and meaning of the topics identified in the complaints.

```
# Evaluation

# -----

ldatopics = lda_model.describeTopics()
ldatopics.show(25)

# Extract vocabulary from CountVectorizer model
```

```

vocab = cv_model.vocabulary
vocab_broadcast = sc.broadcast(vocab)

# Define function to map term IDs to words
def map_termID_to_Word(termIndices):
    words = []
    for termID in termIndices:
        words.append(vocab_broadcast.value[termID])
    return words

# Apply UDF to map term IDs to words in LDA topics
udf_map_termID_to_Word = udf(map_termID_to_Word ,
    ArrayType(StringType()))
ldatopics_mapped = ldatopics.withColumn("topic_desc",
    udf_map_termID_to_Word(ldatopics.termIndices))
ldatopics_mapped.select(ldatopics_mapped.topic,
    ldatopics_mapped.topic_desc).show(50,False)
from pyspark.sql.functions import avg, expr

# Assign topics to each complaint
df_topics = lda_model.transform(df_features)
df_topics.select("month", "day", "topicDistribution").show(20)
logging.info("Training time: %02d:%02d" % (minutes, seconds))

```

```

ldatopics = lda_model.describeTopics()
ldatopics.show(25)24/05/06 03:25:26 WARN DAGScheduler: Broadcasting large task binary with size 5.3 MiB

```

```

>>> ldatopics.show(25)
+-----+-----+-----+
|topic|termIndices|termweights|
+-----+-----+-----+
0|[1, 2, 4, 5, 3, 9...|[0.00695449560521...|
1|[1, 3, 8, 2, 0, 8...|[0.00506828574600...|
2|[10, 14, 6, 18, 6...|[0.00486316084952...|
3|[5, 9, 4, 2, 23, ...|[0.02174000514453...|
4|[6, 10, 15, 14, 1...|[0.00416836162038...|
5|[6, 1, 14, 2, 28...|[0.00459844020042...|
6|[1, 6, 10, 14, 15...|[0.00390756770108...|
7|[1, 6, 14, 8, 10...|[0.00422843174623...|
8|[5, 9, 23, 30, 4...|[0.03404553820427...|
9|[14, 60, 6, 18, 1...|[0.00793727083659...|
10|[10, 18, 15, 14, ...|[0.00568307202037...|
11|[1, 15, 10, 488, ...|[0.00472667127037...|
12|[1, 8, 3, 0, 2, 7...|[0.00547924618688...|
13|[8, 1, 14, 6, 10...|[0.00380233525965...|
14|[137, 8, 889, 361...|[0.00757293982834...|
15|[5, 4, 55, 79, 69...|[0.00661834431153...|
16|[8, 1, 0, 2, 6, 3...|[0.00456339663599...|
17|[1, 0, 3, 2, 8, 7...|[0.00452616814727...|
18|[537, 16, 1, 970...|[0.00721815904260...|
19|[81, 10, 1, 25, 4...|[0.01673995539900...|
20|[1, 145, 963, 759...|[0.00473827186286...|
21|[768, 1, 81, 2, 4...|[0.00896105511250...|
22|[1, 2, 8, 3, 0, 1...|[0.00455196555231...|
23|[1, 2, 3, 0, 25, ...|[0.00457815087466...|
24|[1, 8, 16, 0, 2, ...|[0.00428403809196...|
+-----+-----+-----+

```

```

>>> vocab = cv_model.vocabulary
vocab_broadcast = sc.broadcast(vocab)
>>> vocab_broadcast = sc.broadcast(vocab)
>>> def map_termID_to_word(termIndices):
...     words = []
...     for termID in termIndices:
...         words.append(vocab_broadcast.value[termID])
...     return words
...
>>> udf_map_termID_to_word = udf(map_termID_to_word , ArrayType(StringType()))
>>> ldatopics_mapped = ldatopics.withColumn("topic_desc", udf_map_termID_to_word(ldatopics.termIndices))

```

- **Results:**

- **Number of Rows in DataFrame:** Verified the size of the dataset used for modeling. The DataFrame contains approximately 1.7 million rows.
- **Interpreting Topics:** Presenting the top words for each topic, providing insights into the dominant themes within the complaints narratives.

```

>>> ldatopics_mapped.select(ldatopics_mapped.topic, ldatopics_mapped.topic_desc).show(50,False)
+-----+-----+
|topic|topic_desc|
+-----+-----+
|0| |[account, information, reporting, consumer, report, section, balance, items, accounts, credit]|
|1| |[account, report, debt, information, credit, inquiry, date, cc, accounts, number]|
|2| |[bank, loan, payment, told, mortgage, card, called, back, call, said]|
|3| |[consumer, section, reporting, information, agency, states, usc, person, report, shall]|
|4| |[payment, bank, card, loan, told, account, called, payments, pay, back]|
|5| |[payment, account, loan, information, payments, credit, late, report, consumer, debt]|
|6| |[account, payment, bank, loan, card, debt, told, balance, payments, received]|
|7| |[account, payment, loan, debt, bank, payments, told, information, card, credit]|
|8| |[consumer, section, agency, states, reporting, furnish, information, privacy, violated, rights]|
|9| |[loan, mortgage, payment, told, bank, payments, modification, home, called, pay]|
|10| |[bank, told, card, loan, payment, called, call, said, back, check]|
|11| |[account, card, bank, paypal, payment, money, told, received, information, credit]|
|12| |[account, debt, report, credit, information, creditor, accounts, reporting, payment, obligor]|
|13| |[debt, account, loan, payment, bank, card, mortgage, company, received, told]|
|14| |[claim, debt, compliant, claims, compliance, alleged, collection, plaintiff, document, standards]|
|15| |[consumer, reporting, collection, creditor, requirements, debt, compliance, information, validation, act]|
|16| |[debt, account, credit, information, payment, report, loan, balance, date, company]|
|17| |[account, credit, report, information, debt, accounts, reporting, please, date, payment]|
|18| |[dept, balance, account, ed, payment, information, credit, edxxxx, accounts, report]|
|19| |[inquiry, bank, account, date, finance, loan, payment, told, card, money]|
|20| |[account, score, fico, low, transunion, credit, attempting, causing, payment, report]|
|21| |[inquired, account, inquiry, information, reporting, consumer, date, report, credit, accounts]|
|22| |[account, information, debt, report, credit, balance, accounts, payment, consumer, company]|
|23| |[account, information, report, credit, date, debt, view, reporting, accounts, please]|
|24| |[account, debt, balance, credit, information, payment, report, reporting, bank, loan]|
+-----+-----+

```