Mini project : Customer Churn with SparkML

Objective: To build a complete, end-to-end machine learning pipeline using PySpark. We will predict customer churn by performing data loading, exploratory data analysis (EDA), feature engineering, model training, and evaluation.

Dataset : We will use a public Telco Customer Churn dataset, which is available : https://www.kaggle.com/datasets/blastchar/telco-customer-churn?resource=download

Deliverables :

- You can use your local machine or Google Colab :
  - If you use your local machine, please use a git and a notebook if possible
    - Share the link of git and make it public or add my profile to the contributors (rosariAdr)
    - Send the code/notebook file
  - If you use Google Colab, please use a notebook
    - Share the link of the Google Colab and make it public
- For the questions, send a pdf with the answers
- Don't forget to put all your family names in the pdf and in the code (comment in .py or Markdown in .ipynb)
- Make presentations efforts (use of markdown, comments, etc.) ⇒ Your deliverables are the face of your work, so present it nicely for clients to appreciate and understand your work

---

Phase 1: Setup and Data Loading

Our first step is to initialize a SparkSession and load our data into a DataFrame. This is the entry point for any Spark application.

1.1 Initialize SparkSession

First, we need to import `SparkSession` and create an instance. This object is the main entry point for DataFrame and SQL functionality.

```
# Create a SparkSession
spark = SparkSession.builder \
   .appName("ChurnPredictionPipeline") \
   .getOrCreate()
```

1.2 Load the Dataset

We'll load the `WA_Fn-UseC_-Telco-Customer-Churn.csv` file with `spark.read.csv`.

1.3 Initial Data Inspection

Get a first look at the data with printSchema, show, count functions (part of spark functions)

Phase 2: Exploratory Data Analysis (EDA) & Data Cleaning

Before building a model, we must understand our data. EDA helps us find patterns, anomalies, and relationships between variables. We will also clean the data by handling missing values.

2.1 Data preparation

Prepare the data by doing the correct and necessary conversion of columns (hint : `from pyspark.sql.types import DoubleType` ; `df.cast(DoubleType())`)

2.2 Data Cleaning (Handling Missing Values)

Check and handle any missing values. Use methods like `df.na.drop(), df.count()`

```
# Check for null values in each column
print("Count of null values in each column:")
```

```
df.select([F.count(F.when(F.col(c).isNull(), c)).alias(c) for
c in df.columns]).show()
```

2.3 Univariate Analysis (Analyzing Single Variables)

Let's analyze individual columns to understand their distributions.

Numerical Features

We can get a quick statistical summary of all numerical columns using select, `describe, show.`

Categorical Features

Let's count the occurrences of each category for a few key columns.

2.4 Bivariate Analysis (Analyzing Relationships)

Now, let's explore how different features relate to our target variable, `Churn` , using groupBy, agg, count and orderby.

Phase 3 : Data Transformation & Feature Engineering

Machine learning models require numerical inputs. We need to convert our categorical string columns into numbers and combine all our features into a single vector.

3.1 Identify Feature Columns

First, let's separate our columns into categorical and numerical types. We'll exclude the customerID (as it's just an identifier) and the target variable `Churn`.

```
`# Identify categorical and numerical columns
categorical_cols = #TODO
numerical_cols = #TODO

# The target variable also needs to be converted to a number
df_clean = df_clean.withColumn() #TODO
```

3.2 Define Pipeline Stages

We will now define the `Transformer` and `Estimator` stages for our pipeline.

> 💡 Recap:
>
> - `StringIndexer` (Estimator): Converts a string column to a numerical index column.
> - `VectorAssembler` (Transformer): Merges multiple columns into a single vector column.
> - `StandardScaler` (Estimator): Scales features to have a mean of 0 and standard deviation of 1. This helps some algorithms converge faster.

Phase 4: Building the ML Pipeline

Now we assemble all our transformation stages along with a machine learning model into a single `Pipeline` object. This encapsulates our entire workflow.

4.1 Define the Model

We'll use `LogisticRegression` as our classification model. It's a simple, powerful, and interpretable model for binary classification.

4.2 Assemble the Pipeline

Let's chain all the stages together in the correct order.

```
from pyspark.ml.classification import LogisticRegression

# Define the logistic regression model
lr = LogisticRegression(#TODO)

# Create the pipeline by chaining all stages
pipeline = Pipeline(stages=#TODO)

print("Pipeline created successfully with the following stage
s:")
```

```
for i, stage in #TODO:
    print(f"  {i+1}. {stage}")
```

Phase 5 : Model Training and Evaluation

The final step is to train our pipeline on the data and evaluate its performance on unseen data.

5.1 Split Data

We'll split our data into a training set (80%) and a testing set (20%), using `randomSplit` function.

5.2 Train the Model

We call `.fit()` on our pipeline with the training data. This will execute all the stages in order: the `StringIndexer`s will learn the mappings, the `StandardScaler` will calculate the mean/std, and the `LogisticRegression` model will be trained.

5.3 Make Predictions and Evaluate

Now we use our trained `model` (which is a `PipelineModel`) to make predictions on the test data. Then, we use an `Evaluator` to measure its performance.

```
from pyspark.ml.evaluation import MulticlassClassificationEva
luator

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

# Show some predictions
print("Sample predictions:")
predictions.select(#TODO).show(#TODO)

# Evaluate the model
evaluator_acc = MulticlassClassificationEvaluator(#TODO for t
he accuracy)
```

```
evaluator_f1 = MulticlassClassificationEvaluator(#TODO for th
e F1 score)

accuracy = #TODO
f1_score = #TODO

print("-" * 40)
print(f"Test Accuracy = {accuracy:.4f}")
print(f"Test F1 Score = {f1_score:.4f}")
print("-" * 40)
```

> 💡 Next Steps: An accuracy of ~80% is a good start! To improve this, you could:
>
> - Try more complex models like `RandomForestClassifier` or `GBTClassifier`.
> - Perform hyperparameter tuning using `CrossValidator` to find the best model settings.
> - Engineer more complex features.*

Phase 6 : Data Visualization with Seaborn

Visualizations make the patterns we found in EDA much easier to understand. Since PySpark doesn't have a built-in plotting library, the standard workflow is to aggregate data in Spark, convert the small result to a Pandas DataFrame, and then plot it.

> ⚠️ Important: Only use .toPandas() on small, aggregated DataFrames. Calling it on a large, raw DataFrame will try to pull all the data to a single machine and will likely cause a memory error.

Create at least 5 visualisations ; here are three examples :

- `Churn Distribution (Bar Plot)`
- `Tenure vs. Churn (Box Plot)`
- `Contract Type vs. Churn (Count Plot)`

Phase 8 : More traditional way

Prepare the same cleaning, feature engineering and the rest with Pandas, Scikit-learn.

Phase 9 : Questions and optimisations

Questions :

- Compare both approaches regarding performances, results and easiness.
- Give me 3 optimisations that you could input for this project
    1. Tell me how to implement these changes
    2. Apply one optimisation of your choice