Faculty of Mathematical, Physical and Natural Sciences of Tunis

Faculté des Sciences Mathématiques, Physiques et Naturelles de Tunis

كلية العلوم للرياضيات والفيزياء والطبيعيات بتونس

Academic Project Report

Machine Learning Pipeline in PySpark

Realized By : **Aziz Ayadi**

Louai Azzouni

Rayen Ben Fathallah

Supervised By : **Manel Zekri**

Academic Year : 2024 – 2025

Objective

The primary objective of this project is to build a machine learning model in PySpark to predict whether an individual's income exceeds $50,000 per year based on census data. This guide uses the "Census Income" dataset, incorporating data preprocessing, model building, evaluation, and hyperparameter tuning.

Dataset Overview

The dataset, referred to as "adult.csv," includes features such as age, work class, education level, occupation, marital status, and capital gain. The target variable, income, is a binary classification label that indicates whether a person's income is above or below $50,000.

age	workclass	fnlwgt	education	education_num	marital	occupation	relationship	race	sex	capital_gain	capital_loss	hours_week	native_country	label
25.0	Private	226802.0	11th	7.0	Never-married	Machine-op-inspct	Own-child	Black	Male	0.0	0.0	40.0	United-States	<=50K
38.0	Private	89814.0	HS-grad	9.0	Married-civ-spouse	Farming-fishing	Husband	White	Male	0.0	0.0	50.0	United-States	<=50K
28.0	Local-gov	336951.0	Assoc-acdm	12.0	Married-civ-spouse	Protective-serv	Husband	White	Male	0.0	0.0	40.0	United-States	>50K
44.0	Private	160323.0	Some-college	10.0	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688.0	0.0	40.0	United-States	>50K
18.0	?	103497.0	Some-college	10.0	Never-married	?	Own-child	White	Female	0.0	0.0	30.0	United-States	<=50K

only showing top 5 rows

Steps

1. Initialize SparkContext and SQLContext

- **SparkContext (sc)**: Provides the core functionality for Spark jobs. In PySpark, SparkContext is required to establish the initial connection with Spark.
- **SQLContext (sqlContext)**: Enables Spark to interface with SQL-based data sources. It's initialized to read the dataset and perform SQL-like transformations.

```python
# initialize spark context
import pyspark
from pyspark import SparkContext
sc =SparkContext()
```

```python
# initialize sql context
from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)
```

2. Load and Inspect Dataset

- **Load Dataset**: The CSV file is read into a DataFrame using sqlContext.read.csv(), with header=True and inferSchema=True to automatically infer column types.
- **Schema Inspection**: The .printSchema() function provides an overview of data types and nullability, showing features such as age, workclass, fnlwgt, education, marital-status, and income.

```python
#Todo: using SQLContext to read csv and assign to dataframe
df = sqlContext.read.csv("adult.csv", header=True, inferSchema= True)
```

```python
#Todo:printSchema
df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- workclass: string (nullable = true)
 |-- fnlwgt: integer (nullable = true)
 |-- education: string (nullable = true)
 |-- educational-num: integer (nullable = true)
 |-- marital-status: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- relationship: string (nullable = true)
 |-- race: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- capital-gain: integer (nullable = true)
 |-- capital-loss: integer (nullable = true)
 |-- hours-per-week: integer (nullable = true)
 |-- native-country: string (nullable = true)
 |-- income: string (nullable = true)
```

3. **Data Preprocessing**

- **Column Renaming**: Renaming columns for simplicity and consistency in naming conventions.

```
# Run the cell to rename the columns properly:
cols = ['age','workclass','fnlwgt','education','education_num','marital','occupation','relationship','race','sex','capi

#note income -renamed-> as label
df=df.toDF(*cols)
```

- **Type Conversion**: Conversion of continuous features to FloatType using a custom function. Important continuous features include age, fnlwgt, capital_gain, education_num, capital_loss, and hours_week.

```
# Import all from `sql.types`
from pyspark.sql.types import *

# Write a custom function to convert the data type of DataFrame columns
def convertColumn(df, names, newType):
    for name in names:
        df = df.withColumn(name, df[name].cast(newType))
    return df
# List of continuous features
CONTI_FEATURES  = ['age', 'fnlwgt','capital_gain', 'education_num', 'capital_loss', 'hours_week']
# Convert the type
df = convertColumn(df, CONTI_FEATURES, FloatType())
# Check the dataset
df.printSchema()
```

```
root
 |-- age: float (nullable = true)
 |-- workclass: string (nullable = true)
 |-- fnlwgt: float (nullable = true)
 |-- education: string (nullable = true)
 |-- education_num: float (nullable = true)
 |-- marital: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- relationship: string (nullable = true)
 |-- race: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- capital_gain: float (nullable = true)
 |-- capital_loss: float (nullable = true)
 |-- hours_week: float (nullable = true)
 |-- native_country: string (nullable = true)
 |-- label: string (nullable = true)
```

- **Descriptive Statistics**: Use .describe() to generate basic statistical summaries such as mean, standard deviation, min, and max for each column.

summary	age	workclass	fnlwgt	education	education_num	marital	occupation	relationship	race	sex	capital_gain	capital_loss	hours_week	native_country	label
count	48842	48842	48842	48842	48842	48842	48842	48842	48842	48842	48842	48842	48842	48842	48842
mean	38.64358543876172	null	189664.13459727284	null	10.078088530363212	null	null	null	null	null	1079.0676262233324	87.50231358257237	40.422382375824085	null	null
stddev	13.710509934443518	null	105604.02542315758	null	2.570972755592256	null	null	null	null	null	7452.019057655418	403.00455212435924	12.391444024252296	null	null
min	17.0	?	12285.0	10th	1.0	Divorced	?	Husband	Amer-Indian-Eskimo	Female	0.0	0.0	1.0	?	<=50K
max	90.0	Without-pay	1490400.0	Some-college	16.0	Widowed	Transport-moving	Wife	White	Male	99999.0	4356.0	99.0	Yugoslavia	>50K

- **Grouping and Aggregation**: For example, computing the average capital_gain by marital status provides insights into capital gains across different marital groups.

```python
#example
df.groupby('marital').agg({'capital_gain': 'mean'}).show()
```

marital	avg(capital_gain)
Separated	581.8424836601307
Never-married	384.382639449029
Married-spouse-ab...	629.0047770700637
Divorced	793.6755615860094
Widowed	603.6442687747035
Married-AF-spouse	2971.6216216216217
Married-civ-spouse	1739.7006121810625

- **Crosstab Analysis**: Creates a cross-tabulation between age and income (<=50K or >50K), providing a count of individuals in each age group by income category.

```
#todo crosstab computation
df.crosstab('age', 'label').sort("age_label").show()
```

```
+---------+-----+----+
|age_label|<=50K|>50K|
+---------+-----+----+
|     17.0|  595|   0|
|     18.0|  862|   0|
|     19.0| 1050|   3|
|     20.0| 1112|   1|
|     21.0| 1090|   6|
|     22.0| 1161|  17|
|     23.0| 1307|  22|
|     24.0| 1162|  44|
|     25.0| 1119|  76|
|     26.0| 1068|  85|
|     27.0| 1117| 115|
|     28.0| 1101| 179|
|     29.0| 1025| 198|
|     30.0| 1031| 247|
|     31.0| 1050| 275|
|     32.0|  957| 296|
|     33.0| 1045| 290|
|     34.0|  949| 354|
|     35.0|  997| 340|
|     36.0|  948| 400|
+---------+-----+----+
only showing top 20 rows
```

- **Null Handling and Row Exclusion**: Removes rows with missing values using .dropna(). For example, records from native_country with only one entry (like "Holand-Netherlands") are excluded to avoid issues in cross-validation.

```
#Drop null vals
df  = df.dropna()
```

```
df_remove = df.filter(df.native_country!='Holand-Netherlands')
```

```
#TODO : follow the above instruction
df.filter(df.native_country == 'Holand-Netherlands').count()
df.groupby('native_country').agg({'native_country': 'count'}).sort(asc("count(native_country)")).show()
```

```
+--------------------+---------------------+
|      native_country|count(native_country)|
+--------------------+---------------------+
|  Holand-Netherlands|                    1|
|             Hungary|                   19|
|            Honduras|                   20|
|            Scotland|                   21|
|Outlying-US(Guam-...|                   23|
|          Yugoslavia|                   23|
|                Laos|                   23|
|      Trinadad&Tobago|                   27|
|            Cambodia|                   28|
|                Hong|                   30|
|            Thailand|                   30|
|             Ireland|                   37|
|              France|                   38|
|             Ecuador|                   45|
|                Peru|                   46|
|              Greece|                   49|
|           Nicaragua|                   49|
|                Iran|                   59|
|              Taiwan|                   65|
|            Portugal|                   67|
+--------------------+---------------------+
only showing top 20 rows
```


4. Feature Engineering

- **Age Transformation**: Create a new feature, age_square, to capture potential non-linear relationships between age and income.

```python
#TODO:  # Add age square

from pyspark.sql.functions import *

# 1 Select the column
age_square = df.select(col("age")**2)

# 2 Apply the transformation and add it to the DataFrame
df = df.withColumn("age_square", col("age")**2)

df.printSchema()
```

```
root
 |-- age: float (nullable = true)
 |-- workclass: string (nullable = true)
 |-- fnlwgt: float (nullable = true)
 |-- education: string (nullable = true)
 |-- education_num: float (nullable = true)
 |-- marital: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- relationship: string (nullable = true)
 |-- race: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- capital_gain: float (nullable = true)
 |-- capital_loss: float (nullable = true)
 |-- hours_week: float (nullable = true)
 |-- native_country: string (nullable = true)
 |-- label: string (nullable = true)
 |-- age_square: double (nullable = true)
```

5. Building the Data Processing Pipeline

- **Categorical Feature Encoding**: Use StringIndexer and OneHotEncoderEstimator to encode categorical features, including workclass, education, marital, occupation, relationship, race, sex, and native_country.

```python
#import libraries for pipeline
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
```

```python
# 1. Encode the categorical data
CATE_FEATURES = ['workclass', 'education', 'marital', 'occupation', 'relationship', 'race', 'sex', 'native_country']

# stages in our Pipeline
stages = []


for categoricalCol in CATE_FEATURES:
    stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol + "Index")

    encoder = OneHotEncoderEstimator(inputCols=[stringIndexer.getOutputCol()],
                                     outputCols=[categoricalCol + "classVec"])

    stages += [stringIndexer, encoder]
```

- **Label Encoding**: The target variable income is converted into a binary label (label) using StringIndexer.

```
# 2. Index the label feature
# Convert label into label indices using the StringIndexer
label_stringIdx =  StringIndexer(inputCol="label", outputCol="newlabel")
stages += [label_stringIdx]
```

- **Feature Vectorization**: Use VectorAssembler to combine all transformed categorical and continuous features into a single vector, features.

```
# 3. Add continuous variable
assemblerInputs = [c + "classVec" for c in CATE_FEATURES] + CONTI_FEATURES
```

```
# 4. Assemble the steps
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
stages += [assembler]
```

- **Pipeline Creation**: Instantiate a Pipeline that will execute all the stages sequentially.

```
# Create a Pipeline.
pipeline = Pipeline(stages=stages)
pipelineModel = pipeline.fit(df_remove)
model = pipelineModel.transform(df_remove)
```

```
model
```

```
DataFrame[age: float, workclass: string, fnlwgt: float, education: string, education_num: float, marital: string, occup
ation: string, relationship: string, race: string, sex: string, capital_gain: float, capital_loss: float, hours_week: f
loat, native_country: string, label: string, age_square: double, workclassIndex: double, workclassclassVec: vector, edu
cationIndex: double, educationclassVec: vector, maritalIndex: double, maritalclassVec: vector, occupationIndex: double,
occupationclassVec: vector, relationshipIndex: double, relationshipclassVec: vector, raceIndex: double, raceclassVec: v
ector, sexIndex: double, sexclassVec: vector, native_countryIndex: double, native_countryclassVec: vector, newlabel: do
uble, features: vector]
```

6. Model Building - Logistic Regression

- **Dataset Preparation**: Split the dataset into an 80/20 train-test split for model validation.

```
# Split the data into train and test sets
train_data, test_data = df_train.randomSplit([.8,.2],seed=1234)
```

- **Logistic Regression Initialization**: Initialize Logistic Regression with labelCol="label" and featuresCol="features".

```python
#You initialize lr by indicating the label column and feature columns.
# Import `LogisticRegression`
from pyspark.ml.classification import LogisticRegression

# Initialize `lr`
lr = LogisticRegression(labelCol="label",
                        featuresCol="features",
                        maxIter=10,
                        regParam=0.3)

# Fit the data to the model
linearModel = lr.fit(train_data)
```

```python
# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(linearModel.coefficients))
print("Intercept: " + str(linearModel.intercept))
```

```
Coefficients: [-0.06629458791643376,-0.15218373640701383,-0.053913615606758065,-0.16967312430449774,-0.12115314684082322,0.13
25974961176333,0.1943887659566995,-0.6386553259560794,-0.20168892525823268,-0.06643613691435478,0.22587144074752571,0.3784635
752251114,-0.0044245321889490345,-0.2958940967195082,-0.011315453541718349,-0.3286032800269526,-0.4220383458346703,0.57488818
94870274,-0.40562097016057325,-0.2321967047902892,0.6030218476150906,-0.3529240017664873б,-0.42080161545502637,0.325045773350
55407,-0.3496608365399488,-0.20341828378529217,-0.21097882986337838,-0.16094910982880872,-0.10229984961820654,0.1935414055955
1283,-0.05741072218845116,0.27840022983220586,-0.10767152632715087,0.04564764837286497,-0.29067926822772944,-0.22290332501866
694,-0.1707732587338623,-0.1265004208587058,-0.3046016757809549,-0.3240136096656416З,0.11297915494496764,0.12425063820676888,
-0.2716362961533152,0.26838077428027984,-0.19803625220102 91,-0.29122621595643805,-0.24230441670165628,0.4124628047635877,-0.0
5433645026419642,-0.1908446548558885,-0.06449555808439705,-0.269348178043779,0.1680087916932683,-0.1253754753434063,-0.386858
1630578742,-0.18369433515375264,0.02912182143548542,-0.08956223920758943,-0.2708959392074213,-0.03487727693207883,-0.32413919
15336244,-0.04758250791733353,-0.1173475422607489,0.08015333921255102,-0.36180606787408326,-0.33618210864537906,-0.1663353505
9376112,-0.006091174986221248,-0.45442144073339164,-0.05071249102702611,-0.3521378742759532,-0.2426920862206138,-0.2172135389
7728612,-0.5821780298068088,-0.17111959024994547,-0.00021772139996622131,-0.15792241012962724,-0.13336347352772165,-0.1244484
9686729739,-0.4299324718213068,-0.4802401343425621,-0.3509031990149178,0.15554349625772512,0.0444256440286982,-0.28977028525
75532,-0.19176517240952143,0.35513884661883627,-0.47121791074628866,0.16608265034986552,-0.6627801214856749,-0.40786931479756
95,-0.22434028566679146,-0.21468806117681136,0.007013088722353848,1.1642146321728235e-07,2.1439613693010857e-05,0.02852355930
4261892,0.00022176143684426227,0.008705658401723285]
Intercept: -2.0325289129477486
```

7. Model Evaluation

- **Accuracy Calculation**: Compute accuracy by comparing predicted labels to true labels in the test set.
- **ROC and Area Under ROC**: Evaluate the model's performance using the ROC curve, where an area under the ROC of 0.89 indicates strong model performance.

```python
#We need to look at the accuracy metric to see how well (or bad) the model performs.

def accuracy_m(model):
    predictions = model.transform(test_data)
    cm = predictions.select("label", "prediction")
    acc = cm.filter(cm.label == cm.prediction).count() / cm.count()
    print("Model accuracy: %.3f%%" % (acc * 100))

accuracy_m(model = linearModel)
```

```
Model accuracy: 81.887%
```

```python
### Use ROC
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Evaluate model
evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")
print(evaluator.evaluate(predictions))
print(evaluator.getMetricName())
```

```
0.894260182552082
areaUnderROC
```

8. Hyperparameter Tuning with Cross-Validation

- **Parameter Grid**: Construct a ParamGridBuilder to test two values for the regParam parameter.

```
#you can tune the hyperparameters.
#Similar to scikit learn you create a parameter grid, and you add the parameters you want to tune.
#To reduce the time of the computation, you only tune the regularization parameter with only two values.
#use
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Create ParamGrid for Cross Validation
paramGrid = (ParamGridBuilder()
             .addGrid(lr.regParam, [0.01, 0.5])
             .build())
```

- **CrossValidator**: Set up a 5-fold cross-validation to identify the optimal regularization parameter and evaluate model accuracy.

```
from time import *
start_time = time()

# Create 5-fold CrossValidator
cv = CrossValidator(estimator=lr,
                    estimatorParamMaps=paramGrid,
                    evaluator=evaluator, numFolds=5)

# Run cross validations
cvModel = cv.fit(train_data)
# likely take a fair amount of time
end_time = time()
elapsed_time = end_time - start_time
print("Time to train model: %.3f seconds" % elapsed_time)
```

```
Time to train model: 1079.451 seconds
```

- **Best Model Accuracy**: The cross-validated model achieves an accuracy of 84.82%, demonstrating the effectiveness of tuning.

```
#accuracy of cv selected model
accuracy_m(model = cvModel)
```

```
Model accuracy: 84.820%
```

Conclusion

The completed PySpark ML pipeline demonstrates efficient handling of categorical data, preprocessing of features, and model evaluation in Spark. With a final accuracy of around

84.82% on the test set, this model effectively predicts income levels based on census data.

This pipeline can be expanded with additional feature engineering, model evaluation metrics, and alternative algorithms if needed. The entire process is scalable and easily adaptable for similar classification tasks in large datasets using Spark.