PySpark MLlib Regression Code - Line by Line Explanation

1. Environment Setup and Installation

python

```python
pip install pyspark
```

Explanation: Installs PySpark, which is the Python API for Apache Spark. This allows Python programs to use Spark's distributed computing capabilities.

python

```python
import os
os.environ["JAVA_HOME"]="/lib/jvm/java-11-openjdk-amd64"
```

Explanation: Sets the JAVA_HOME environment variable because Spark runs on the Java Virtual Machine (JVM). This tells Python where to find the Java installation. **Example**: Without this, you might get errors like "JAVA_HOME is not set" when trying to start Spark.

2. Spark Context and SQL Context Setup

python

```python
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext
```

Explanation: Imports necessary Spark components:

- `SparkConf`: Configuration for Spark applications
- `SparkContext`: Entry point for Spark functionality
- `SQLContext`: Interface for working with structured data

python

```python
conf = SparkConf().set('spark.ui.port', '4050').setAppName("films").setMaster("local[2]")
sc = SparkContext.getOrCreate(conf=conf)
sqlContext = SQLContext(sc)
```

Explanation:

- Creates Spark configuration with UI port 4050, app name "films", and master as "local[2]" (2 local threads)
- `getOrCreate()` either gets existing SparkContext or creates new one
- Creates SQLContext for DataFrame operations **Example**: "local[2]" means run Spark locally using 2 CPU cores

3. Data Loading and Exploration

```python
data = sqlContext.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load('Boston.csv')
data.show()
```

Explanation:

- Loads CSV file using Databricks CSV format
- `header='true'`: First row contains column names
- `inferschema='true'`: Automatically detects data types
- `show()`: Displays first 20 rows **Example**: This loads the Boston Housing dataset with columns like crime rate, property tax, etc.

```python
data.cache()
data.printSchema()
```

Explanation:

- `cache()`: Stores DataFrame in memory for faster repeated access
- `printSchema()`: Shows column names and data types **Example Output**:

```
root
 |-- crim: double (nullable = true)
 |-- zn: double (nullable = true)
 |-- medv: double (nullable = true)
```

4. Data Analysis and Visualization

python

```python
data.toPandas()
```

Explanation: Converts Spark DataFrame to Pandas DataFrame for easier manipulation and analysis. **Note**: Only use this with small datasets that fit in memory.

python

```python
import pandas as pd
from matplotlib import cm
from pandas.plotting import scatter_matrix
numeric_features = [t[0] for t in data.dtypes if t[1] == 'int' or t[1] == 'double']
sampled_data = data.select(numeric_features).sample(False, 0.8).toPandas()
axs = scatter_matrix(sampled_data, figsize=(10, 10))
```

Explanation:

- Extracts numeric column names from DataFrame schema
- `sample(False, 0.8)`: Takes 80% random sample without replacement
- Creates scatter plot matrix showing relationships between all numeric features **Example**: If you have columns [crim, zn, age, medv], this creates a 4x4 grid of scatter plots

python

```python
n = len(sampled_data.columns)
for i in range(n):
    v = axs[i, 0]
    v.yaxis.label.set_rotation(0)
    v.yaxis.label.set_ha('right')
    v.set_yticks(())
    h = axs[n-1, i]
    h.xaxis.label.set_rotation(90)
    h.set_xticks(())
```

Explanation: Formatting the scatter matrix plot by rotating labels and removing tick marks for better readability.

5. Feature Engineering

python

```python
from pyspark.ml.feature import VectorAssembler
vectorAssembler = VectorAssembler(inputCols = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax', 'ptratio', 'bla
v_data = vectorAssembler.transform(data)
v_data = v_data.select(['features', 'medv'])
```

Explanation:

- `VectorAssembler`: Combines multiple columns into a single vector column
- MLlib algorithms expect features in a single vector column
- Selects only the features vector and target variable 'medv' (median home value) **Example**: Transforms columns [crim=0.1, zn=0.2, indus=0.3] into features=[0.1, 0.2, 0.3]

python

```python
v_data.show(3)
```

Explanation: Shows first 3 rows of the transformed data with features vector and target.

6. Correlation Analysis

python

```python
import six
for i in data.columns:
  if not( isinstance(data.select(i).take(1)[0][0], six.string_types)):
    print( "Correlation to medv for ", i, data.stat.corr('medv',i))
```

Explanation:

- Calculates correlation between each numeric column and target variable 'medv'
- `six.string_types`: Checks if column contains strings (to skip non-numeric columns)
- `data.stat.corr()`: Computes Pearson correlation coefficient **Example Output**: "Correlation to medv for rm 0.695" (rooms per dwelling positively correlated with home value)

7. Train-Test Split

python

```python
splits = v_data.randomSplit([0.7, 0.3])
train_df = splits[0]
test_df = splits[1]
```

Explanation:

- Randomly splits data into 70% training and 30% testing sets
- `randomSplit([0.7, 0.3])`: Creates list with two DataFrames **Example**: If you have 1000 rows, train_df gets ~700 rows, test_df gets ~300 rows

8. Linear Regression Model

python

```python
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol = 'features', labelCol='medv', maxIter=10)
lr_model = lr.fit(train_df)
```

Explanation:

- Creates Linear Regression model instance
- `featuresCol='features'`: Column containing feature vectors
- `labelCol='medv'`: Target variable column
- `maxIter=10`: Maximum 10 iterations for optimization
- `fit()`: Trains the model on training data

python

```python
trainingSummary = lr_model.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)
```

Explanation:

- Gets training summary with model metrics
- RMSE (Root Mean Square Error): Lower is better
- R^2 (R-squared): Higher is better (0-1 scale, 1 = perfect fit) **Example**: "RMSE: 4.68, r2: 0.74" means model explains 74% of variance with average error of 4.68

9. Model Evaluation

```python
lr_predictions = lr_model.transform(test_df)
lr_predictions.select("prediction","medv","features").show(5)
```

Explanation:

- `transform()`: Applies trained model to test data
- Shows actual vs predicted values for first 5 test samples **Example Output**:

```
+----------+----+--------------------+
|prediction|medv|            features|
+----------+----+--------------------+
|     22.35|24.0|[0.09178,0.0,4.05...|
```

```python
from pyspark.ml.evaluation import RegressionEvaluator
lr_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="medv",metricName="r2")
print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(lr_predictions))
```

Explanation:

- Creates evaluator for regression metrics
- Calculates R^2 score on test data (unseen data)
- Tests model's generalization ability **Example**: "R Squared (R2) on test data = 0.72" means model explains 72% of variance in test data

10. Decision Tree Regression

```python
from pyspark.ml.regression import DecisionTreeRegressor
dt = DecisionTreeRegressor(featuresCol ='features', labelCol = 'medv')
dt_model = dt.fit(train_df)
dt_predictions = dt_model.transform(test_df)
```

Explanation:

- Decision Tree creates a tree-like model of decisions

- Each internal node represents a feature test
- Each leaf represents a prediction value
- Can capture non-linear relationships better than linear regression

python

```python
dt_evaluator = RegressionEvaluator(labelCol="medv", predictionCol="prediction", metricName="rmse")
rmse = dt_evaluator.evaluate(dt_predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

Explanation: Evaluates Decision Tree using RMSE metric.

python

```python
dt_model.featureImportances
```

Explanation:

- Shows which features are most important for predictions
- Values sum to 1.0, higher values indicate more important features **Example**: SparseVector showing [rm: 0.45, lstat: 0.23, ...] means 'rm' is most important feature

11. Gradient Boosted Tree Regression

python

```python
from pyspark.ml.regression import GBTRegressor
gbt = GBTRegressor(featuresCol = 'features', labelCol = 'medv', maxIter=10)
gbt_model = gbt.fit(train_df)
gbt_predictions = gbt_model.transform(test_df)
```

Explanation:

- Gradient Boosted Trees build multiple weak decision trees
- Each tree corrects errors from previous trees
- Usually provides better accuracy than single decision tree
- `maxIter=10`: Uses 10 trees in the ensemble

python

```python
gbt_evaluator = RegressionEvaluator(labelCol="medv", predictionCol="prediction", metricName="rmse")
rmse = gbt_evaluator.evaluate(gbt_predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

Explanation: Evaluates Gradient Boosted Tree model performance.

Key Concepts Summary

1. **Data Pipeline**: Load → Transform → Split → Train → Evaluate
2. **Feature Engineering**: VectorAssembler combines features into single vector
3. **Model Comparison**: Linear Regression vs Decision Tree vs Gradient Boosting
4. **Evaluation Metrics**: RMSE (error), R^2 (explained variance)
5. **Distributed Computing**: Spark handles large datasets across multiple machines

Boston Housing Dataset Features

- **crim**: Crime rate per capita
- **zn**: Proportion of residential land zoned for lots over 25,000 sq ft
- **indus**: Proportion of non-retail business acres
- **chas**: Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- **nox**: Nitric oxides concentration
- **rm**: Average number of rooms per dwelling
- **age**: Proportion of owner-occupied units built prior to 1940
- **dis**: Weighted distances to employment centers
- **rad**: Index of accessibility to radial highways
- **tax**: Property tax rate per $10,000
- **ptratio**: Pupil-teacher ratio by town
- **black**: Proportion of blacks by town
- **lstat**: % lower status of the population
- **medv**: Median value of owner-occupied homes (target variable)