Data Intensive Applications (Spark) Assignment Report

Tugay Talha İçen

March, 2024

1 Introduction

This report presents the process and results of building a regression model with PySpark for predicting house prices using the California Housing Prices dataset. The assignment's purpose and scope were to analyze the data, select attributes, prepare data for machine learning, build a machine learning model, and measure its performance.

2 Data Set

The California Housing Prices dataset was used for this assignment. It includes various features such as longitude, latitude, housing median age, total rooms, total bedrooms, population, households, median income, and median house value.

3 Code Overview

The code.ipynb notebook contains the implementation of the assignment's tasks using PySpark. It includes the following main sections:

1. Setting up PySpark environment and importing necessary libraries.
2. Loading the dataset and analyzing its structure and content.
3. Preprocessing the data, including handling missing values and encoding categorical variables.
4. Feature selection using Recursive Feature Elimination (RFE).
5. Building regression models using various algorithms such as Linear Regression, Random Forest, Gradient-Boosted Tree, Decision Tree, Generalized Linear Regression, and Factorization Machines.

6. Evaluating model performance using metrics like Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared.

7. Selecting the best-performing model and training it on the entire dataset.

8. Saving the trained model for future use.

4 Code Snippets

Below are some key code snippets from the implementation:

4.1 Importing Libraries and Setting up PySpark

```
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler, StandardScaler, OneHotEncoder, S
from pyspark.ml.regression import LinearRegression, RandomForestRegressor, GBTR
from pyspark.ml.evaluation import RegressionEvaluator
import numpy as np
```

4.2 Loading and Analyzing the Data

This part of the code initializes a SparkSession, reads the dataset from a CSV file, and displays its schema and the first 5 rows. Then, it counts the total number of rows in the dataset and prints unique values in the "ocean_proximity" column. Finally, it calculates the rate of missing values for each column in the dataset.

```
# Create SparkSession
spark = SparkSession.builder.master('local').appName("HousePricePrediction").get

# Load data from CSV file
data = spark.read.csv("housing.csv", header=True, inferSchema=True)

# Analyze the data
data.printSchema()
data.show(5)
print("Total number of rows:", data.count())

for col in ["ocean_proximity"]:
    print(f"Unique values in {col}:")
    data.select(col).distinct().show()

for col in data.columns:
    print(f"NA rate in {col}: {data.filter(data[col].isNull()).count() / data.co
```

4.3 Data Preprocessing

This snippet performs data preprocessing tasks such as one-hot encoding categorical variables, handling missing values in the "total_bedrooms" column, and scaling features using StandardScaler.

```
# Do one-hot encoding for categorical columns
indexer = StringIndexer(inputCol="ocean_proximity", outputCol="ocean_proximity_i
data = indexer.fit(data).transform(data)
encoder = OneHotEncoder(inputCol="ocean_proximity_index", outputCol="ocean_proxi
encModel = encoder.fit(data)
data = encModel.transform(data)


data.show(5)
print("Total number of rows:", data.count())

# Handle missing values in total_bedrooms column
data = data.fillna(data.approxQuantile("total_bedrooms", [0.5], 0.001)[0], subse
```

4.4 Feature Selection and Data Preparation

This section defines a function rfe_feature_selection for performing Recursive Feature Elimination (RFE) using Random Forest regression to select the most important features. It then applies RFE to select the top 8 features and prepares the data by assembling features into a single vector and scaling them.

```
def rfe_feature_selection(data, features, target_col, num_features, estimator):
    # Initialize remaining features and selected features
    remaining_features = features
    selected_features = []

    while len(selected_features) < num_features:
        # Create VectorAssembler for remaining features
        assembler = VectorAssembler(inputCols=remaining_features, outputCol="fea
        data_assembled = assembler.transform(data)

        # Train Random Forest model
        rf = estimator(labelCol=target_col, featuresCol="features")
        model = rf.fit(data_assembled)

        # Get feature importances
        importances = model.featureImportances

        # Convert SparseVector to dense representation
        importances_dense = importances.toArray()
```

```
            # Find the least important feature
            least_important_feature_index = np.argmin(importances_dense)

            least_important_feature = remaining_features[least_important_feature_ind


            # Remove least important feature from remaining features
            remaining_features.remove(least_important_feature)

            # Add least important feature to selected features
            selected_features.append(least_important_feature)

    # Select final features
    return data.select(selected_features + [target_col])

# Select reamining features except target column
features = [col for col in data.columns if col != "ocean_proximity_encoded" and

# Eleminate not necassary features
num_features = 8  # Smaller Values giving very bad results
data_filtered = rfe_feature_selection(data, features, "median_house_value", num

print(data_filtered.columns)

# Select reamining features except target column
features = [col for col in data_filtered.columns if col != "median_house_value"]
if "ocean_proximity_index" in features:
    features.remove("ocean_proximity_index")
    features.append("ocean_proximity_encoded")

# Create a VectorAssembler to combine features into a single vector
assembler = VectorAssembler(inputCols=features, outputCol="features")
data = assembler.transform(data)

# Scale features
scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures")
data = scaler.fit(data).transform(data)

# Split data into training and testing sets
train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)
```

4.5 Building, Training, and Evaluating Regression Models

Here, a list of regression models along with their names and corresponding PySpark models are defined. The loop iterates over each model, fits it to the training data, makes predictions on the test data, and evaluates its performance using Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared metrics.

```
# Create a Regression models
models = [
    ("Linear Regression", LinearRegression(labelCol="median_house_value", featur
    ("Random Forest", RandomForestRegressor(labelCol="median_house_value", featu
    ("Gradient-Boosted Tree", GBTRegressor(labelCol="median_house_value", featur
    ("Decision Tree", DecisionTreeRegressor(labelCol="median_house_value", featu
    ("Generalized Linear Regression", GeneralizedLinearRegression(labelCol="medi
    ("Factorization Machines", FMRegressor(labelCol="median_house_value", featur
]
evaluator = RegressionEvaluator(labelCol="median_house_value", predictionCol="pr

for name, model in models:
    model = model.fit(train_data)
    predictions = model.transform(test_data)
    mse = evaluator.evaluate(predictions, {evaluator.metricName: "mse"})
    rmse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})
    r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})
    print(f"Model: {name}")
    print(f"    MSE: {mse}, RMSE: {rmse}, R-squared: {r2}")
```

4.6 Evaluating Final Model Performance

This snippet trains the best-performing model (Gradient-Boosted Tree) on the entire dataset, evaluates its performance using various metrics such as MSE, RMSE, MAE, R-squared, and explained variance, and prints the results.

```
# Train the best model
best_model = GBTRegressor(labelCol="median_house_value", featuresCol="scaledFeat
best_model = best_model.fit(train_data)

# Evaluate best model
evaluator = RegressionEvaluator(labelCol="median_house_value", predictionCol="pr
predictions = best_model.transform(test_data)
mse = evaluator.evaluate(predictions, {evaluator.metricName: "mse"})
rmse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})
mae = evaluator.evaluate(predictions, {evaluator.metricName: "mae"})
r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})
explained_variance = evaluator.evaluate(predictions, {evaluator.metricName: "var
```

```
# Print results
print(f"Best Model: Gradient-Boosted Tree")
print(f"    MSE: {mse}, RMSE: {rmse}, MAE: {mae}, R-squared: {r2}, Explained Var
```

4.7 Saving The Best Model

Finally, the best model (GBTRegressor) is trained on the entire dataset and saved for future use. The model is saved in a directory named "best_model" and then zipped for easier distribution. The SparkSession is stopped to release resources.

```
# Train the best model on the entire dataset
best_model = GBTRegressor(labelCol="median_house_value", featuresCol="scaledFeat
best_model = best_model.fit(data)

# Save the model
best_model.save("best_model")

# Zip the model
import shutil
shutil.make_archive("best_model", 'zip', "best_model")

# Stop SparkSession
spark.stop()
```

5 Results

The performance of different regression models was evaluated using test data. The Gradient-Boosted Tree model achieved the best results with the following metrics:

- MSE: 3,210,917,002.30
- RMSE: 56,664.95
- MAE: 39,837.33
- R-squared: 0.768
- Explained Variance: 10,334,394,537.06

6 Conclusion

In conclusion, a regression model was successfully built using PySpark for predicting house prices. The Gradient-Boosted Tree algorithm demonstrated the

best performance among the models evaluated. The trained model can be further utilized for predicting house prices in California.