decision-trees

April 7, 2024

```
import pyspark
import os
import sys
from pyspark import SparkContext
```

Imports necessary libraries and modules for working with PySpark, including the SparkContext class.

```
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
```

Sets environment variables for PySpark to use the current Python executable for both the worker and driver processes.

```
from pyspark.sql import SparkSession
```

Imports the SparkSession class from the pyspark.sql module, which is used to create a Spark session and interact with Spark SQL.

[ ]:
```
import pyspark
import os
import sys
from pyspark import SparkContext
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.config("spark.driver.memory", "16g").appName('chapter_4').getOrCre
```

This line of code creates a new SparkSession object, which is the entry point for working with Apache Spark. Here's a breakdown of what's happening:

- `SparkSession.builder`: This creates a SparkSession builder object, which allows you to configure various settings for the Spark session.
- `.config("spark.driver.memory", "16g")`: This sets the configuration property `spark.driver.memory` to `"16g"`, which allocates 16GB of memory for the Spark driver process.
- `.appName('chapter_4')`: This sets the name of the Spark application to `"chapter_4"`. This name will appear in the Spark UI and logs.

- `.getOrCreate()`: This method either gets an existing SparkSession or creates a new one if none exists.

The resulting `spark` object is the entry point for working with Spark, allowing you to create DataFrames, perform transformations, and execute Spark jobs.

```
[ ]: spark = SparkSession.builder.config("spark.driver.memory", "16g").
      ↪appName('chapter_4').getOrCreate()
```

```
data_without_header = spark.read.option("inferSchema", True).option("header", False).csv("data/
data_without_header.printSchema()
```

This code reads a CSV file named "covtype.data" from the "data" directory using Apache Spark's `read` method. The `option("inferSchema", True)` tells Spark to infer the data types of the columns automatically. `option("header", False)` indicates that the CSV file does not have a header row.

The resulting DataFrame is stored in the variable `data_without_header`. Finally, `data_without_header.printSchema()` prints the schema (column names and data types) of the DataFrame to the console.

```
[ ]: data_without_header = spark.read.option("inferSchema", True).option("header",␣
      ↪False).csv("data/covtype.data")
     data_without_header.printSchema()
```

This code is written in Python and uses the PySpark library for working with Apache Spark, a distributed computing framework for big data processing.

```
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import col
```

These lines import the `DoubleType` class from `pyspark.sql.types` and the `col` function from `pyspark.sql.functions`. `DoubleType` is used to represent double-precision floating-point numbers, and `col` is used to reference a column in a DataFrame.

```
colnames = [
    "Elevation",
    "Aspect",
    "Slope",
    "Horizontal_Distance_To_Hydrology",
    "Vertical_Distance_To_Hydrology",
    "Horizontal_Distance_To_Roadways",
    "Hillshade_9am",
    "Hillshade_Noon",
    "Hillshade_3pm",
    "Horizontal_Distance_To_Fire_Points"
] + [f"Wilderness_Area_{i}" for i in range(4)] + [f"Soil_Type_{i}" for i in range(40)] + ["Cove
```

This line creates a list `colnames` containing column names for a DataFrame. It includes various feature names related to elevation, aspect, slope, distances, hillshade, wilderness areas, soil types, and a target variable "Cover_Type".

```
data = data_without_header.toDF(*colnames).withColumn("Cover_Type", col("Cover_Type").cast(Dou
```

This line creates a new DataFrame `data` from an existing DataFrame `data_without_header`. The `toDF` method is used to create a new DataFrame with column names specified in `colnames`. The `withColumn` method is then used to cast the "Cover_Type" column to `DoubleType`.

```
data.head()
```

This line displays the first few rows of the `data` DataFrame.

```
[ ]: from pyspark.sql.types import DoubleType
     from pyspark.sql.functions import col
     colnames = [
         "Elevation",
         "Aspect",
         "Slope",
         "Horizontal_Distance_To_Hydrology",
         "Vertical_Distance_To_Hydrology",
         "Horizontal_Distance_To_Roadways",
         "Hillshade_9am",
         "Hillshade_Noon",
         "Hillshade_3pm",
         "Horizontal_Distance_To_Fire_Points"
     ] + [f"Wilderness_Area_{i}" for i in range(4)] + [f"Soil_Type_{i}" for i in␣
      ↪range(40)] + ["Cover_Type"]

     data = data_without_header.toDF(*colnames).withColumn("Cover_Type",␣
      ↪col("Cover_Type").cast(DoubleType()))
     data.head()
```

```
(train_data, test_data) = data.randomSplit([0.9, 0.1])
```

This line splits the `data` DataFrame into two parts: `train_data` (90% of the data) and `test_data` (10% of the data). The `randomSplit` method is used to randomly split the data into the specified fractions.

```
train_data.cache()
test_data.cache()
```

These lines cache the `train_data` and `test_data` DataFrames in memory. Caching is useful when you plan to reuse the data multiple times, as it avoids recomputing the data each time it is accessed, improving performance.

```
[ ]: (train_data, test_data) = data.randomSplit([0.9, 0.1])
     train_data.cache()
     test_data.cache()
```

`from pyspark.ml.feature import VectorAssembler` imports the `VectorAssembler` class from the `pyspark.ml.feature` module. This class is used to combine multiple input columns into a single vector column.

```
input_cols = colnames[:-1]
```

This line creates a list `input_cols` containing all column names from the `colnames` list except the last one. These columns will be used as input features for the `VectorAssembler`.

```
vector_assembler = VectorAssembler(inputCols=input_cols, outputCol="featureVector")
```

This line creates an instance of the `VectorAssembler` class with `input_cols` as the input columns and `"featureVector"` as the name of the output column containing the assembled vectors.

```
assembled_train_data = vector_assembler.transform(train_data)
```

This line applies the `vector_assembler` transformation to the `train_data` DataFrame, creating a new DataFrame `assembled_train_data` with the `"featureVector"` column added.

```
assembled_train_data.select("featureVector").show(truncate = False)
```

This line selects the `"featureVector"` column from the `assembled_train_data` DataFrame and prints its contents without truncation.

[ ]:
```
from pyspark.ml.feature import VectorAssembler
input_cols = colnames[:-1]
vector_assembler = VectorAssembler(inputCols=input_cols,␣
 ↪outputCol="featureVector")
assembled_train_data = vector_assembler.transform(train_data)
assembled_train_data.select("featureVector").show(truncate = False)
```

```
from pyspark.ml.classification import DecisionTreeClassifier
```

Imports the DecisionTreeClassifier class from the pyspark.ml.classification module. This class is used to create a decision tree model for classification tasks.

```
classifier = DecisionTreeClassifier(seed = 1234, labelCol="Cover_Type",
                                    featuresCol="featureVector",
                                    predictionCol="prediction")
```

Creates an instance of the DecisionTreeClassifier with the following parameters:
- `seed`: Sets the random seed for reproducibility.
- `labelCol`: Specifies the name of the column containing the labels/target variable.
- `featuresCol`: Specifies the name of the column containing the feature vectors.
- `predictionCol`: Specifies the name of the column where the predicted labels will be stored.

```
model = classifier.fit(assembled_train_data)
```

Trains the decision tree model on the `assembled_train_data` DataFrame.

```
print(model.toDebugString)
```

Prints the debug string representation of the trained decision tree model, which can be useful for understanding the structure and parameters of the model.

[ ]:
```
from pyspark.ml.classification import DecisionTreeClassifier
classifier = DecisionTreeClassifier(seed = 1234, labelCol="Cover_Type",
featuresCol="featureVector",
predictionCol="prediction")
```

```
model = classifier.fit(assembled_train_data)
print(model.toDebugString)
```

```
import pandas as pd

pd.DataFrame(
    model.featureImportances.toArray(),
    index=input_cols,
    columns=['importance']
).sort_values(by="importance", ascending=False)
```

This code creates a pandas DataFrame from the feature importances of a machine learning model. Here's a breakdown:

- `import pandas as pd`: Imports the pandas library, which provides data manipulation and analysis tools.
- `model.featureImportances.toArray()`: Retrieves the feature importances from the `model` object and converts them to an array.
- `index=input_cols`: Sets the row labels (index) of the DataFrame to the `input_cols` variable, which likely contains the names of the input features.
- `columns=['importance']`: Sets the column name of the DataFrame to 'importance'.
- `.sort_values(by="importance", ascending=False)`: Sorts the DataFrame in descending order based on the 'importance' column.

The resulting DataFrame will have the feature names as row labels and their corresponding importance values in the 'importance' column, sorted from highest to lowest importance.

[ ]:
```
import pandas as pd
pd.DataFrame(
    model.featureImportances.toArray(),
    index=input_cols, columns=['importance']).sort_values(by="importance",␣
 ↪ascending=False
)
```

```
predictions = model.transform(assembled_train_data)
```

This line applies the trained model to the assembled training data to generate predictions.

```
predictions.select("Cover_Type", "prediction", "probability").show(10, truncate = False)
```

This line selects the columns "Cover_Type" (the target variable), "prediction" (the predicted class), and "probability" (the probability of the predicted class) from the `predictions` DataFrame, and displays the first 10 rows without truncating the output.

[ ]:
```
predictions = model.transform(assembled_train_data)
```

```
predictions.select("Cover_Type", "prediction", "probability").show(10, truncate␣
 ↪= False)
```

`from pyspark.ml.evaluation import MulticlassClassificationEvaluator`

Imports the `MulticlassClassificationEvaluator` class from the `pyspark.ml.evaluation` module. This class is used to evaluate the performance of multiclass classification models.

`evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type", predictionCol="prediction'`

Creates an instance of the `MulticlassClassificationEvaluator` class, specifying the column names for the true labels (`"Cover_Type"`) and predicted labels (`"prediction"`).

`evaluator.setMetricName("accuracy").evaluate(predictions)`

Sets the evaluation metric to "accuracy" and evaluates the predictions on the given `predictions` DataFrame. This calculates the overall accuracy of the multiclass classification model.

`evaluator.setMetricName("f1").evaluate(predictions)`

Sets the evaluation metric to "f1" (F1 score) and evaluates the predictions on the same `predictions` DataFrame. This calculates the weighted average of the F1 scores for each class in the multiclass classification problem.

[ ]:
```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type",␣
 ↪predictionCol="prediction")
evaluator.setMetricName("accuracy").evaluate(predictions)
evaluator.setMetricName("f1").evaluate(predictions)
```

```
confusion_matrix = predictions.groupBy("Cover_Type").pivot("prediction", range(1,8)).count().na
confusion_matrix.show()
```

This code calculates the confusion matrix for a multi-class classification problem and displays it.

`predictions.groupBy("Cover_Type")`: Groups the predictions DataFrame by the "Cover_Type" column, which likely represents the true labels.

`.pivot("prediction", range(1,8))`: Pivots the DataFrame to create a column for each possible prediction value (1 to 7). This is done to create a contingency table.

`.count()`: Counts the number of occurrences for each combination of "Cover_Type" and "prediction" values.

`.na.fill(0.0)`: Fills any missing (NaN) values with 0.0, assuming that missing values represent zero occurrences.

`.orderBy("Cover_Type")`: Sorts the resulting DataFrame by the "Cover_Type" column.

`confusion_matrix.show()`: Displays the resulting confusion matrix DataFrame.

The confusion matrix shows the number of instances that were correctly and incorrectly classified for each class. The diagonal elements represent the correctly classified instances, while the off-diagonal elements represent the misclassified instances.

```
[ ]: confusion_matrix = predictions.groupBy("Cover_Type").pivot("prediction",␣
      ↪range(1,8)).count().na.fill(0.0).orderBy("Cover_Type")
     confusion_matrix.show()
```

This code defines a function `class_probabilities` that takes a PySpark DataFrame `data` as input and calculates the proportion of each class in the data. Here's a breakdown of the code:

```
def class_probabilities(data):
    total = data.count() # Get the total number of rows in the DataFrame
    return data.groupBy("Cover_Type").count().orderBy("Cover_Type").select(col("count").cast(Do
```

The function is then called twice with different DataFrames `train_data` and `test_data` to calculate the class probabilities for each dataset:

```
train_prior_probabilities = class_probabilities(train_data)
test_prior_probabilities = class_probabilities(test_data)
```

The resulting lists of Row objects are converted to lists of floats:

```
train_prior_probabilities = [p[0] for p in train_prior_probabilities]
test_prior_probabilities = [p[0] for p in test_prior_probabilities]
```

Finally, the sum of the products of corresponding elements from `train_prior_probabilities` and `test_prior_probabilities` is calculated using a list comprehension and the `zip` function:

```
sum([train_p * cv_p for train_p, cv_p in zip(train_prior_probabilities, test_prior_probabiliti
```

This code calculates the class probabilities for two datasets and then computes a metric based on the product of corresponding class probabilities from the two datasets.

```
[ ]: from pyspark.sql import DataFrame
     def class_probabilities(data):
         total = data.count()
         return data.groupBy("Cover_Type").count().orderBy("Cover_Type").
      ↪select(col("count").cast(DoubleType())).withColumn("count_proportion",␣
      ↪col("count")/total).select("count_proportion").collect()

     train_prior_probabilities = class_probabilities(train_data)
     test_prior_probabilities = class_probabilities(test_data)

     train_prior_probabilities = [p[0] for p in train_prior_probabilities]
     test_prior_probabilities = [p[0] for p in test_prior_probabilities]
     sum([train_p * cv_p for train_p, cv_p in zip(train_prior_probabilities,␣
      ↪test_prior_probabilities)])
```

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

Imports the `Pipeline` class from the `pyspark.ml` module, which is used to chain multiple Transformer and Estimator objects in a sequence.

```
assembler = VectorAssembler(inputCols=input_cols, outputCol="featureVector")
```

Creates a `VectorAssembler` object that combines multiple input columns into a single vector column named "featureVector". The `input_cols` parameter specifies the list of input column names to be combined.

```
classifier = DecisionTreeClassifier(seed=1234, labelCol="Cover_Type", featuresCol="featureVect
```

Creates a `DecisionTreeClassifier` object with the specified parameters:
- `seed=1234`: Sets the random seed for reproducibility.
- `labelCol="Cover_Type"`: Specifies the name of the label column.
- `featuresCol="featureVector"`: Specifies the name of the feature vector column created by the `VectorAssembler`.
- `predictionCol="prediction"`: Specifies the name of the output column for predicted labels.

```
pipeline = Pipeline(stages=[assembler, classifier])
```

Creates a `Pipeline` object that chains the `assembler` and `classifier` stages together. The `Pipeline` will first apply the `VectorAssembler` to combine input columns, and then apply the `DecisionTreeClassifier` on the resulting feature vector.

```
[ ]: from pyspark.ml import Pipeline
     assembler = VectorAssembler(inputCols=input_cols, outputCol="featureVector")
     classifier = DecisionTreeClassifier(seed=1234, labelCol="Cover_Type",␣
      ↪featuresCol="featureVector", predictionCol="prediction")
     pipeline = Pipeline(stages=[assembler, classifier])
```

This markdown documentation explains the provided Python code:

The code imports the `ParamGridBuilder` class from the `pyspark.ml.tuning` module. This class is used to create a grid of hyperparameters for model tuning.

```
from pyspark.ml.tuning import ParamGridBuilder
```

The `ParamGridBuilder` is then used to create a `paramGrid` object, which specifies the hyperparameters and their values to be explored during the tuning process.

```
paramGrid = ParamGridBuilder().addGrid(classifier.impurity, ["gini", "entropy"]).addGrid(class
```

The `addGrid` method is called multiple times to add different hyperparameters and their respective values to the grid. In this case, the hyperparameters being tuned are:

- `impurity`: Either "gini" or "entropy" for the impurity criterion.
- `maxDepth`: Maximum depth of the decision tree, with values 1 and 20.
- `maxBins`: Maximum number of bins used for splitting features, with values 40 and 300.
- `minInfoGain`: Minimum information gain for a split to be considered, with values 0.0 and 0.05.

Finally, the `build` method is called to create the `paramGrid` object.

The code also creates a `MulticlassClassificationEvaluator` object, which is used to evaluate the performance of a multiclass classification model.

```
multiclassEval = MulticlassClassificationEvaluator().setLabelCol("Cover_Type").setPredictionCol
```

The `setLabelCol` method specifies the column containing the true labels, which is “Cover_Type” in this case. The `setPredictionCol` method specifies the column containing the predicted labels, which is “prediction”. The `setMetricName` method sets the evaluation metric to be used,

```
[ ]: from pyspark.ml.tuning import ParamGridBuilder
paramGrid = ParamGridBuilder().addGrid(classifier.impurity, ["gini",␣
 ↪"entropy"]).addGrid(classifier.maxDepth, [1, 20]).addGrid(classifier.
 ↪maxBins, [40, 300]).addGrid(classifier.minInfoGain, [0.0, 0.05]).build()
multiclassEval = MulticlassClassificationEvaluator().setLabelCol("Cover_Type").
 ↪setPredictionCol("prediction").setMetricName("accuracy")
```

```
from pyspark.ml.tuning import TrainValidationSplit
validator = TrainValidationSplit(
    seed=1234,
    estimator=pipeline,
    evaluator=multiclassEval,
    estimatorParamMaps=paramGrid,
    trainRatio=0.9
)
validator_model = validator.fit(train_data)
```

This code sets up a `TrainValidationSplit` object for model tuning in PySpark. It splits the training data into separate training and validation sets, allowing for hyperparameter tuning and model evaluation.

- `from pyspark.ml.tuning import TrainValidationSplit` imports the `TrainValidationSplit` class from PySpark’s tuning module.
- `TrainValidationSplit` is initialized with the following parameters:
  - `seed=1234` sets a random seed for reproducibility.
  - `estimator=pipeline` specifies the machine learning pipeline to be tuned.
  - `evaluator=multiclassEval` sets the evaluation metric for the tuning process (e.g., accuracy, F1-score).
  - `estimatorParamMaps=paramGrid` provides a grid of hyperparameter values to be evaluated.
  - `trainRatio=0.9` specifies that 90% of the data should be used for training, and the remaining 10% for validation.
- `validator_model = validator.fit(train_data)` trains the `TrainValidationSplit` object on the `train_data` dataset, producing a tuned model.

The resulting `validator_model` can be used for making predictions or further analysis, having been tuned on the validation set for optimal performance.

```
[ ]: from pyspark.ml.tuning import TrainValidationSplit
     validator = TrainValidationSplit(
         seed=1234,
         estimator=pipeline,
         evaluator=multiclassEval,
         estimatorParamMaps=paramGrid,
         trainRatio=0.9
     )
     validator_model = validator.fit(train_data)
```

```
from pprint import pprint
best_model = validator_model.bestModel
```

Imports the `pprint` module for pretty printing and assigns the best model from the `validator_model` object to the `best_model` variable.

```
pprint(best_model.stages[1].extractParamMap())
```

Pretty prints the parameter map of the second stage in the `best_model` pipeline.

```
multiclassEval.evaluate(best_model.transform(test_data))
```

Evaluates the `best_model` on the `test_data` using the `multiclassEval` evaluator, which likely calculates metrics for a multiclass classification problem.

```
[ ]: from pprint import pprint
     best_model = validator_model.bestModel
     pprint(best_model.stages[1].extractParamMap())

     multiclassEval.evaluate(best_model.transform(test_data))
```

```
validator_model = validator.fit(train_data)
```

Fits a validator model on the training data `train_data`.

```
metrics = validator_model.validationMetrics
params = validator_model.getEstimatorParamMaps()
```

Retrieves the validation metrics and estimator parameter maps from the fitted validator model.

```
metrics_and_params = list(zip(metrics, params))
metrics_and_params.sort(key=lambda x: x[0], reverse=True)
```

Combines the metrics and parameter maps into a list of tuples, and sorts the list in descending order based on the metric values.

```
metrics.sort(reverse=True)
print(metrics[0])
```

Sorts the metrics list in descending order, and prints the highest metric value.

```
[ ]: validator_model = validator.fit(train_data)
     metrics = validator_model.validationMetrics
     params = validator_model.getEstimatorParamMaps()
```

```
metrics_and_params = list(zip(metrics, params))
metrics_and_params.sort(key=lambda x: x[0], reverse=True)

metrics.sort(reverse=True)
print(metrics[0])
```

`from pyspark.sql.functions import udf`
Imports the `udf` function from the `pyspark.sql.functions` module, which is used to create user-defined functions in PySpark.

`from pyspark.sql.types import IntegerType`
Imports the `IntegerType` class from the `pyspark.sql.types` module, which is used to specify the return type of the user-defined function.

```
def unencode_one_hot(data):
    wilderness_cols = ['Wilderness_Area_' + str(i) for i in range(4)]
    wilderness_assembler = VectorAssembler().setInputCols(wilderness_cols).setOutputCol("wilde
    unhot_udf = udf(lambda v: v.toArray().tolist().index(1))
    with_wilderness = wilderness_assembler.transform(data).drop(*wilderness_cols).withColumn("w
```

This part of the code deals with the "Wilderness_Area" columns. It creates a list of column names `wilderness_cols`, then uses `VectorAssembler` to combine these columns into a single vector column named "wilderness". It then creates a user-defined function `unhot_udf` that takes a vector and returns the index of the element with value 1 (assuming it's a one-hot encoded vector). This function is applied to the "wilderness" column, and the result is cast to `IntegerType`.

```
    soil_cols = ['Soil_Type_' + str(i) for i in range(40)]
    soil_assembler = VectorAssembler().setInputCols(soil_cols).setOutputCol("soil")
    with_soil = soil_assembler.transform(with_wilderness).drop(*soil_cols).withColumn("soil", u
```

This part of the code does the same thing as the previous part, but for the "Soil_Type" columns. It creates a list of column names `soil_cols`, combines them into a vector

```
[ ]: from pyspark.sql.functions import udf
     from pyspark.sql.types import IntegerType
     def unencode_one_hot(data):
         wilderness_cols = ['Wilderness_Area_' + str(i) for i in range(4)]
         wilderness_assembler = VectorAssembler().setInputCols(wilderness_cols).
       ↪setOutputCol("wilderness")
         unhot_udf = udf(lambda v: v.toArray().tolist().index(1))
         with_wilderness = wilderness_assembler.transform(data).
       ↪drop(*wilderness_cols).withColumn("wilderness", unhot_udf(col("wilderness")).
       ↪cast(IntegerType()))
         soil_cols = ['Soil_Type_' + str(i) for i in range(40)]
         soil_assembler = VectorAssembler().setInputCols(soil_cols).
       ↪setOutputCol("soil")
         with_soil = soil_assembler.transform(with_wilderness).drop(*soil_cols).
       ↪withColumn("soil", unhot_udf(col("soil")).cast(IntegerType()))
         return with_soil
```

This code performs the following tasks:

1. `unenc_train_data = unencode_one_hot(train_data)` converts the one-hot encoded `train_data` into a more compact format.

2. `unenc_train_data.printSchema()` prints the schema of the `unenc_train_data` DataFrame.

3. `unenc_train_data.groupBy('wilderness').count().show()` groups the data by the 'wilderness' column and counts the number of rows for each group.

```
from pyspark.ml.feature import VectorIndexer
cols = unenc_train_data.columns
input_cols = [c for c in cols if c!='Cover_Type']
assembler = VectorAssembler().setInputCols(input_cols).setOutputCol("featureVector")
```

This code creates a `VectorAssembler` object that combines all columns except 'Cover_Type' into a single vector column named 'featureVector'.

```
indexer = VectorIndexer().setMaxCategories(40).setInputCol("featureVector").setOutputCol("inde
classifier = DecisionTreeClassifier().setLabelCol("Cover_Type").setFeaturesCol("indexedVector"
pipeline = Pipeline().setStages([assembler, indexer, classifier])
```

This code creates a `VectorIndexer` object that converts the 'featureVector' column into a numerical vector with a maximum of 40 categories. It then creates a `DecisionTreeClassifier` object with 'Cover_Type' as the label column, 'indexedVector' as the features column, and 'prediction' as the prediction column. Finally, it creates a `Pipeline` object that combines the `assembler`, `indexer`, and `classifier` stages.

[ ]:
```
unenc_train_data = unencode_one_hot(train_data)
unenc_train_data.printSchema()
unenc_train_data.groupBy('wilderness').count().show()

from pyspark.ml.feature import VectorIndexer
cols = unenc_train_data.columns
input_cols = [c for c in cols if c!='Cover_Type']
assembler = VectorAssembler().setInputCols(input_cols).
  ↪setOutputCol("featureVector")
indexer = VectorIndexer().setMaxCategories(40).setInputCol("featureVector").
  ↪setOutputCol("indexedVector")
classifier = DecisionTreeClassifier().setLabelCol("Cover_Type").
  ↪setFeaturesCol("indexedVector").setPredictionCol("prediction")
pipeline = Pipeline().setStages([assembler, indexer, classifier])
```

```
from pyspark.ml.classification import RandomForestClassifier

# Create a RandomForestClassifier instance
classifier = RandomForestClassifier(seed=1234, # Set the seed for reproducibility
                                    labelCol="Cover_Type", # Name of the label column
                                    featuresCol="indexedVector", # Name of the features column
                                    predictionCol="prediction") # Name of the prediction colum
```

This code imports the RandomForestClassifier class from the pyspark.ml.classification module. It then creates an instance of the RandomForestClassifier with the following parameters:

- seed=1234: Sets the seed for the random number generator to ensure reproducibility.
- labelCol="Cover_Type": Specifies the name of the column containing the labels/target variable.
- featuresCol="indexedVector": Specifies the name of the column containing the feature vectors.
- predictionCol="prediction": Specifies the name of the column where the predicted labels will be stored.

The RandomForestClassifier is a supervised learning algorithm used for classification tasks. It constructs multiple decision trees and combines their predictions to improve accuracy and reduce overfitting.

```
[ ]: from pyspark.ml.from pyspark.ml.classification import RandomForestClassifier
     classifier = RandomForestClassifier(seed=1234, labelCol="Cover_Type",␣
      ↪featuresCol="indexedVector", predictionCol="prediction")
```

This code is written in Python and uses the PySpark library for machine learning tasks. Here's a breakdown of what the code does:

```
cols = unenc_train_data.columns
input_cols = [c for c in cols if c!='Cover_Type']
```

This extracts the column names from the unenc_train_data DataFrame and creates a list input_cols containing all column names except 'Cover_Type'.

```
assembler = VectorAssembler().setInputCols(input_cols).setOutputCol("featureVector")
indexer = VectorIndexer.setMaxCategories(40).setInputCol("featureVector").setOutputCol("indexe
```

These lines create two PySpark Transformer objects: assembler and indexer. assembler combines the input columns into a single vector column named "featureVector". indexer encodes the vector column into a numerical format with a maximum of 40 categories.

```
pipeline = Pipeline().setStages([assembler, indexer, classifier])
```

This creates a Pipeline object that chains the assembler, indexer, and classifier (not shown) together.

```
paramGrid = ParamGridBuilder().addGrid(classifier.impurity, ["gini", "entropy"]).addGrid(class
```

This creates a ParamGrid object that defines a grid of hyperparameters to be tested for the classifier model.

```
multiclassEval = MulticlassClassificationEvaluator().setLabelCol("Cover_Type").setPredictionCo
```

This creates a MulticlassClassificationEvaluator object that will evaluate the model's performance using the accuracy metric.

“‘python
validator = TrainValidationSplit(
seed=1234,
estimator=pipeline,
evaluator=multiclassEval,

```
[ ]: cols = unenc_train_data.columns
input_cols = [c for c in cols if c!='Cover_Type']
assembler = VectorAssembler().setInputCols(input_cols).
 ↪setOutputCol("featureVector")
indexer = VectorIndexer.setMaxCategories(40).setInputCol("featureVector").
 ↪setOutputCol("indexedVector")
pipeline = Pipeline().setStages([assembler, indexer, classifier])
paramGrid = ParamGridBuilder().addGrid(classifier.impurity, ["gini",␣
 ↪"entropy"]).addGrid(classifier.maxDepth, [1, 20]).addGrid(classifier.
 ↪maxBins, [40, 300]).addGrid(classifier.minInfoGain, [0.0, 0.05]).build()
multiclassEval = MulticlassClassificationEvaluator().setLabelCol("Cover_Type").
 ↪setPredictionCol("prediction").setMetricName("accuracy")
validator = TrainValidationSplit(
    seed=1234,
    estimator=pipeline,
    evaluator=multiclassEval,
    estimatorParamMaps=paramGrid,
    trainRatio=0.9
)
validator_model = validator.fit(unenc_train_data)
best_model = validator_model.bestModel

forest_model = best_model.stages[2]
feature_importance_list = list(zip(input_cols,
forest_model.featureImportances.toArray()))
feature_importance_list.sort(key=lambda x: x[1], reverse=True)
pprint(feature_importance_list)
```