Decision Trees

- Import the necessary libraries.
- Create a Spark session, which is the entry point for all Spark applications. This allows users to perform operations over distributed datasets.

In [ ]:

```
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_
```

- Read the data using the read function of spark. This allows data to be read from various sources
- The option function allows users to set the read options:
  ```
  - the schema is inferred automatically from the input data
  - there is no header in the input data
  ```
- the csv function allows the csv file to be read and converted to a dataframe from the specified file path

The schema of the created dataframe is displayed

In [ ]:

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

- The list of column names for the dataframe is defined, and the .toDF() function returns a new dataframe with the specified column names.
- The withColumn() function adds a new column to the dataframe, which is the result of a computation on an existing column. In this case, the values in the CoverType column are cast to Double Type. The new column with double type values overwrites the original.

- The first row of the dataframe is displayed

In [ ]:

```
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()
```

The dataset is randomly split into two portions, of 90% and 10% samples of the original using the randomSplit function. These two new dataframes are cached to persist them in memory, so they can be reused for further actions, avoiding the need for repeated computatations

In [ ]:

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

- VectorAssembler is a feature transformer which merges multiple columns into a single vector column. This can be used to train ML models.
- Here the instance of the VectorAssembler merges all columns specified by the input_cols list into a single vector.
- The transform function applies the transformation to the train_data and creates a new dataframe with a column called 'featureVector' containing the combined feature vector of columns for each record.

In [ ]:

```
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)
```

An instance of the Decision Tree Classifier is made, which uses the column 'featureVector' to estimate the value of 'Cover_type' column and provide its predictions in the column 'prediction'. A decision tree is a machine learning algorithm which uses a set of recursive rules to classify given data into categories.

The .fit() method is used to train the model on the processed train data, returning the trained model.

The toDebugString attribute of the model displays the rules learnt by the model to make classifications based on different features

In [ ]:

```
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)
```

The featureImportance attribute of the model contains the relative importance of each feature the model learnt during training. This signifies the relevance of each feature in making a decision about the class.

This is converted to an array using the toArray() function and subsequently into a Pandas Dataframe. The values are stored in descending order of their importances to show the modt significant features of the classifier

In [ ]:

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

The transform function of the model is used to make predictions on new unseen data. It uses its learnt decision rules on the provided features in the data and returns a new dataframe with a column called 'prediction' containing the predictions made.

- The predictions are displayed from this dataframe, along with the true label as stored in the column 'Cover_tyoe'. The 'probability' column represents the probability of the given data sample belonging to each class

In [ ]:

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

The multiclass classification evaluator is a module used to compare predictions, as specified in the predictionCol 'prediction' with the ground truth classes, as specified in the labelCol 'Cover_Type'.

The instance of the evaluator is applied on the predictions table, using the evaluate() functiomm to calculate different metrics such as accuracy and f1 score.

- Accuracy is the fraction of correct classifications out of all classifications
- F1 score is the harmonic mean of precision and recall. Precision is the fraction of true positives out of all positive predictions and recall is the fraction of true positives out of all the tru samples in the data.

These metrics provide a good indication of the performance of the model

In [ ]:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type",
    predictionCol="prediction")
evaluator.setMetricName("accuracy").evaluate(predictions)
evaluator.setMetricName("f1").evaluate(predictions)
```

A confusion matrix is created for the predicitons. This consists of a table of predicted classes against true classes, with a count of the number of correct and incorrect predictions per class.

A pivot table is used, which groups values by two dimensions, here by the 'CoverType' and 'prediction' column, so that the distinct values of each column become the rows and columns of the output. For each group, an aggregation is performed to count the total number of samples falling under that group. (The null values are replaced with 0)

In [ ]:

```
confusion_matrix = predictions.groupBy("Cover_Type").\
    pivot("prediction", range(1,8)).count().\
    na.fill(0.0).\
    orderBy("Cover_Type")
confusion_matrix.show()
```

A function is defined to find the prior probabilities of each class in the given data. This is done by performing the aggregation count over each distinct value in the 'Cover_Type' column and dividing it by the total number of samples.

The prior probabilites of the classes are calculated for both the train and test data.

In [ ]:

```
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
```

A random classifier can be made to make predictions based on the prevelance of the classes in the training set. Each classification would be correct in proportion to the prevelance of the class in the test set. Thus summing the products of the corresponding train and test prior probabilities would indicate the accuracy of a random classifier.

In [ ]:

```
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_
```

HyperParameter Tuning

An instance of the VectorAssembler and DecisionTreeClassfier are made whcih are set as stages of a Pipeline object. A pipeline is a sequence of stages, with each stage being a transformer or an estimator. The stages are run in sequence on the input dataframe. The transform method is run on transformers, such as assemblers, to convert the features of the dataframe to the desired format, and the fit method is run on estimators to create transformers in the form of trained models.

In [ ]:

```
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])
```

A ParamGridBuilder is instantiated. This is a module for hyperparameter tuning. It creates a grid of all the combinations of parameters to perform grid search based model selection.

```
- The addGrid method sets the possible values for the specified
parameters -- impurity, maximum depth, maximum number of bins and
minimum information gain required to use as a decision feature
- The build method returns the combination of all parameters
specified by the param grid.
```

A MultiClassClassificationEvaluator is also instantiated to calculate the accuracy of predictions made

In [ ]:

```
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")
```

The TrainValidationSplit model is a validator module for hyperparameter tuning. It randomly splits the input into training and validation sets, trains the estimator provided, for each set of parameters specified in the estimatorParamMaps attribute and then evaluates the model on the validation set to find the best model.

In [ ]:

```
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)
```

The best model found during validation is stored in the bestModel attribute of the trained validator model. The second stage of this pipeline, that is the decision tree classifier is selected, and the extractParamMap() function is applied to display the learned parameters of this model.

In [ ]:

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

The evaluation metrics for each model trained from the set of parameters are stored in the validationMetrics attribute of the validator model. The getEstimatorParamMaps() function retrieves the parameters of each trained model. The corresponding metrics and parameters for each model is stored and sorted in descending order of metrics to store the most accurate models first.

In [ ]:

```
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_and_params
```

The metrics are sorted in descending order to store the highest accuracy acheived by the model first.

In [ ]:

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

The best model is used to make predictions on the test data, and its results are evaluated by the evaluator object to return the accuracy of classification

In [ ]:

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

A function is defined to unencode one-hot-encoded data:

- the wilderness_assembler converts all the one-hot-categorised wilderness_cols into a single feature vector
- a user defined function is registered which finds the index (location) of 1 in a given input by transforming it into an array and subsequently a list. This is the unencoded category of the input
- the assembler is applied to the input data, with all the individual one-hot-categorised columns dropped and the one-hot-encoded feature vector replaced with the unencoded category by applying the udf on the feature vector column

A similar process is carried out for all the one-hot-encoded soil columns to replace it with the unencoded category.

In [ ]:

```
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
```

The function defined above is applied on the train data to unencode the category of wilderness and soil for the records in the data

In [ ]:

```
unenc_train_data = unencode_one_hot(train_data)
unenc_train_data.printSchema()
```

The unencoded data is grouped by the wilderness column and an aggregation is performed to display the count of records for each distinct value in the column

In [ ]:

```
unenc_train_data.groupBy('wilderness').count().show()
```

A new pipeline is created, which consists of:

- A vector assembler which combines all the columns except the label column 'Cover_Type' into a single vector in the 'featureVector' column
- A vector indexer which processes the dataset of unknown vectors into a dataset with some continuous nad some categorical features. The number of categorical features is

set by the maxCategories attribute. For the categorical variables, which it automatically detects, it builds an index of all the unique values for that feature
- a decision tree classifier

In [ ]:

```
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])
```

Random Forest

A random forest classifier object is instantiated. This is an algorithm which trains a number of decision trees on randomly sampled train data and uses an ensemble of the results of each tree to provide more accurate predictions\

A new pipeline is created, consisting of:

- A vector assembler merging all the feature columns of the unencoded train data
- A vector indexer to process the combined feature vectors
- A random forest classifer

A parameter grid is built for the various combinations of hyperparameters of the decision trees

An evaluator is instantiated to calculate the accuracy of predictions

Finally a TrainValidationSplit module is instantiated and fit on the train data to select the best model by finding the combination of hypermeparamters in the parameter grid that return the highest accuracy when evaluated

The best model is extracted from the bestModel attribute

In [ ]:

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

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
```

The trained random forest model is extracted from the third stage of the fit pipeline of the best model.

The relative importance of each feature is obtained from the featureImportances attribute of the model, storing them in descending order of importance, so the most significant features appear first, and displaying them.

In [ ]:

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

Finally the test data is unencoded.

The transform function is called on the unencoded test data to pass it through the pipeline of the best model and make predictions. These predictions are stored in the 'prediction' column and displayed

In [ ]:

```
unenc_test_data = unencode_one_hot(test_data)
best_model.transform(unenc_test_data.drop("Cover_Type")).\
    select("prediction").show(1)
```