FIT5202 Data processing for big data¶

Activity: Machine Learning with Spark (Transformer, Estimator and Pipeline API)¶

This week we are going to learn about machine learning with Apache Spark. **`MLlib`** is Apache Spark's scalable machine learning library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

This week we are going to learn about transformers, estimators and machine learning pipeline in this tutorial activity.

Table of Contents¶

Initialize Spark Session ¶

TODO: In the cell block below,

- Create a SparkConfig object with application name set as "Spark ML-Transformer, Estimator and Pipeline"
- specify 2 cores for processing
- Use the configuration object to create a spark session named as **spark**.

Important: You cannot proceed to other steps without completing this.

In [ ]:

```
# TODO: Import libraries needed from pyspark

# TODO: Create Spark Configuration Object

# TODO: Create SparkSession
```

Problem Statement ¶

Before we jumpstart coding, it is important to understand the problem and its context. The dataset we are using today is the popular **Adult Income Dataset**.[Source] The dataset provides different parameters of an individual which might influence his/her income.

Objective: We want to explore and see if different personal attributes of a person influence his/her income and whether we can use these attributes to predict their income levels.

Machine Learning Flow¶

The figure below depicts the flow of the Machine Learning approach we want to take. The **ML Algorithm** we are going to use is **Logistic Regression**. Here, we are not going to get into details of the algorithm. We will look at these in details in coming tutorials.

Data Loading and Pre-processing and Exploration ¶

In this step, we shall load the given `adult.csv` file, examine the data and some basic data cleaning operations like checking for null values and finally select a set of relevant columns for our analysis.

In [ ]:

```
#Load the CSV File
df_adult = spark.read.format('csv')\
            .option('header',True).option('escape','"')\
            .load('adult.csv')
```

In [ ]:

```
#Print the Schema
df_adult.printSchema()
```

IMPORTANT: The `income` is our **target variable** also called **label** and we are going to use other **independent** variables to predict the target variable.

In [ ]:

```
#Check the shape of the dataframe
print((df_adult.count(), len(df_adult.columns)))
```

In [ ]:

```
#Summary Statistics
df_adult.describe().show(5)
```

TODO: For better readability of dataframes, you can also convert the dataframe to a **Pandas** DataFrame. Please note that, we can use this only if we have few rows, since the data is loaded into the driver node. Try using `df_adult.describe().toPandas().head()`

IMPORTANT: To use `.toPandas`, Pandas has to be installed. If not, please use `!pip install pandas` to install **Pandas** in Jupyter Notebook.

Exploratory Analysis: Data Exploration and visualization will be covered in the coming tutorials. Here we will focus on the Featurization part.

For simplicity, we are only considering a set of features from the dataset for our analysis. The columns we want to use are `'workclass','education','marital-status','occupation','relationship','race','gender','income'`. We are going to create a dataframe with these columns only and use this DataFrame for the rest of our analysis.

In [ ]:
```
cols=['workclass','education','marital-status','occupation','relationship','race','gender','income']
df = df_adult[cols]
df.show(5)
```
1. Lab Task: Examine the different unique values the `income` column has. Display the `distinct` values of the target variable i.e. the `income` column.

Checking Missing/Null values¶

Check for missing data, drop the rows for missing data.[Read More]

In [ ]:
```
from pyspark.sql.functions import isnan, when, count, col
df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]).show()
```

Estimators, Transformers and Pipelines¶

Spark's machine learning library has three main abstractions.

1. **Transformer:** Takes dataframe as input and returns a new DataFrame with one or more columns appended to it. Implements a `.transform()` method.
2. **Estimator:** Takes dataframe as input and returns a model. Estimator learns from the data. It implements a `.fit()` method.
3. **Pipeline: Combines** together `transformers` and `estimators`. Pipelines implement a `.fit` method.

NOTE: Spark appends columns to pre-existing **immutable** DataFrames rather than performing operations **in-place**.

StopWordsRemover ¶

`StopWordsRemover` takes input as a sequence of strings and drops all stop words. Stop Words are the words that should be excluded from the input, because they appear frequently and don't carry much meaning.[Ref]

In [ ]:
```
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([
    (0, ["I", "saw", "the", "red", "balloon"]),
    (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
remover.transform(sentenceData).show(truncate=False)
```

StringIndexer ¶

`StringIndexer` encodes string columns as indices. It assigns a unique value to each category. We need to define the input column/columns name and the output column/columns name in which we want the results.[Read More]

In [ ]:
```
from pyspark.ml.feature import StringIndexer

df_ref = spark.createDataFrame(
    [(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")],
    ["id", "category"])

indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
indexed_transformer = indexer.fit(df_ref)
indexed = indexed_transformer.transform(df_ref)
indexed.show()
```

Understanding Python List Comprehensions¶

The examples below use List Comprehensions in Python. You may frequently see this being used. You can read more about it here.

In [ ]:
```
# Getting column names from the dataframe
inputCols=[x for x in df.columns]
print(inputCols)

# Note that we have used Python List Comprehension in the above example
#This is equivalent to doing
inputCols=[]
for x in df.columns:
    inputCols.append(x)
print(inputCols)
```


2. Lab Task: Use `StringIndexer` to encode all the columns from the DataFrame `df` we created in the previous step. *Since all the columns we have are categorical in nature, we want to use `StringIndexer` to transform them into numerical values.*

NOTE: You can pass multiple columns as input and output in `StringIndexer` using `StringIndexer(inputCols=["col1","col2"], outputCols=["col1Index","col2Index"])`

In [ ]:
```
#Define the input columns
inputCols=[x for x in df.columns]
#Define the output columns
outputCols=[f'{x}_index' for x in df.columns]
# TODO: Initialize StringIndexer (use inputCols and outputCols)
indexer =

#TODO call the fit and transform() method to get the encoded results
df_indexed =

#TODO Display the output, only the output columns
```

One Hot Encoder (OHE) ¶

One hot encoding is representation of categorical variables as binary vectors. It works in 2 steps:

1. The categorical variables are mapped as integer values
2. Each integer value is represented as binary vector

[Spark References] [Read More]

NOTE: `OneHotEncoder` in Spark does not directly encode the categorical variable. We have converted the categorical variable to numerical using `StringIndexer` in the above step. Now we can implement the `OHE` to the *numerical columns* obtained from the step above.

In [ ]:

```
#Example of OHE from Spark Documentation
from pyspark.ml.feature import OneHotEncoder

df_ref = spark.createDataFrame([
    (0.0, 1.0),
    (1.0, 0.0),
    (2.0, 1.0),
    (0.0, 2.0),
    (0.0, 1.0),
    (2.0, 0.0)
], ["categoryIndex1", "categoryIndex2"])

encoder = OneHotEncoder(inputCols=["categoryIndex1", "categoryIndex2"],
                        outputCols=["categoryVec1", "categoryVec2"])
model = encoder.fit(df_ref)
encoded = model.transform(df_ref)
encoded.show()
```

3. Lab Task: Apply `OneHotEncoder` transformation to all numerical columns in the dataframe. We shouldn't be including the **target** column i.e. `income` anymore here. We just want to include the features.

In [ ]:

```
#WRITE THE CODE WHERE NECESSARY
#the outputcols of previous step act as input cols for this step
inputCols_OHE = #all output columns from StringIndexer except the Income
outputCols_OHE = [f'{x}_vec' for x in inputCols if x!='income']

#Define OneHotEncoder with the appropriate columns
encoder =
# Call fit and transform to get the encoded results
df_encoded =
#Display the output columns
```


Rename the target column to label¶

`label` is popularly used as the name for the target variable. In supervised learning, we have a **labelled** dataset which is why the column name **label** makes sense.

In [ ]:

```
df_encoded=df_encoded.withColumnRenamed('income_index', 'label')
```

VectorAssembler ¶

Finally, once we have transformed the data, we want to combine all the features into a single feature column to train the machine learning model. `VectorAssembler` combines the given list of columns to a *single vector* column. [Spark Ref]

In [ ]:

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

dataset = spark.createDataFrame(
    [(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.5]), 1.0)],
    ["id", "hour", "mobile", "userFeatures", "clicked"])

assembler = VectorAssembler(
    inputCols=["hour", "mobile", "userFeatures"],
```

```
    outputCol="features")

output = assembler.transform(dataset)
print("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column
'features'")
output.select("features", "clicked").show(truncate=False)
```

4. Lab Task: Referring to the example above, use `VectorAssembler` to combine the feature columns from **Task 2** to a single column named **features**.

In [ ]:

```
#WRITE THE CODE WHERE NECESSARY
inputCols=#the output columns from Task 3 i.e. OHE

#Define the assembler with appropriate input and output columns
assembler =
#use the asseembler transform() to get encoded results
df_final =
#Display the output
```

ML Algorithm and Prediction ¶

Here we are using Logistic Regression for the classification. We will explore the details about this algorithm in the next tutorials. Please refer to the [Spark Docs] for further reference.

In [ ]:

```
#Splitting the data into testing and training set 90% into training and 10% for
testing
train, test = df_final.randomSplit([0.9, 0.1])
```

In [ ]:

```
#Implementing the Logistic Regression
from pyspark.ml.classification import LogisticRegression

# Create a LogisticRegression instance. This instance is an Estimator.
lr = LogisticRegression(featuresCol='features',labelCol='label')
model = lr.fit(train)
```


In [ ]:

```
#Here we use the model trained with the training data to give predictions for our
test data
predicted_data = model.transform(test)
predicted_data.select('features','label','prediction').filter(predicted_data.label==1
).show()
```

In [ ]:

```
#This gives the accuracy of the model we have built,
trainingSummary = model.summary
trainingSummary.accuracy
```

NOTE: What is your interpretation about the accuracy of the model?

Pipeline API ¶

In machine learning, it is common to run a sequence of algorithms to process and learn from data. We have seen in the example above, there is a sequence of steps to be done to prepare the data for training. Such sequence of steps in Spark can be reqpresented by a Pipeline, which consists of a sequence of PipelineStages (Transformers and Estimators) to be run in a specific order. We will try to convert the above sequence of transformers and estimators into a Pipeline.

In the figure above, we can the steps `StringIndexer, OneHotEncoder, VectorAssembler and MLAlgorithm` are plugged into the pipeline.

Pipeline API Example ¶

An example demonstrating the use of `Pipeline` taken from [Spark Docs] is given below. Go through this example to implement your own `Pipeline` for **Task 5**.

In [ ]:

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "win million dollar", 1.0),
    (1, "Office meeting", 0.0),
    (2, "Do not miss this opportunity", 1.0),
    (3, "update your password", 1.0),
    (4, "Assignment submission", 0.0)
], ["id", "text", "label"])

# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (5, "get bonus of 200 dollars"),
    (6, "change your bank password"),
    (7, "Next meeting is at 5pm"),
    (8, "Late submission"),
    (9, "Daily newsletter")
], ["id", "text"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and 
lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
```

```
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))
```

Congratulations on finishing this activity. See you next week.¶