Classification with PySpark

Purpose

Introduction

In general, we use Spark for big data processing. We will utilize PySpark, which is the Python wrapper to connect the Python IDE to the Java engine, as Spark is written in Java. Spark SQL is a Spark module for structured data processing. It can execute SQL queries and contains a dataframe for scaling the data analysis for big data. The Spark ML library performs machine learning in Spark. It makes machine learning scalable for big data. For this project, we will use algorithms such as logistic regression (Ridge and Lasso Regressions) and Gradient Boosting. Generally, there are two types of operations in Spark, transformations and actions. Transformations creates a new dataframe from the previous one, and actions compute a result based on a dataframe and returns a value to the driver program.

Objective

The objective of this project is to utilize the PySpark library to perform data processing and machine learning on the Titanic dataset. The ultimate goal is to predict whether a passenger survived or not.

In [ ]:
```
# Load Library
import numpy as np
import pandas as pd
from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.sql.functions import mean, col, regexp_extract, when, isnan, count
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import LogisticRegression, GBTClassifier
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from itertools import chain
```

We will first start a Spark Session and create a spark instance. We read in csv files like we would with Pandas. When we enable inferSchema, it will find the right schema for each column.

In [ ]:
```
# Create Spark Session (like a container)
spark = SparkSession.builder.appName('PySpark with ML').getOrCreate()
train_df = spark.read.csv('train.csv', header=True, inferSchema=True)
test_df = spark.read.csv('test.csv', header=True, inferSchema=True)
```

In [ ]:
```
train_df.printSchema()
```

```
root
 |-- PassengerId: integer (nullable = true)
 |-- Survived: integer (nullable = true)
 |-- Pclass: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Age: double (nullable = true)
 |-- SibSp: integer (nullable = true)
 |-- Parch: integer (nullable = true)
 |-- Ticket: string (nullable = true)
 |-- Fare: double (nullable = true)
 |-- Cabin: string (nullable = true)
 |-- Embarked: string (nullable = true)
```

In [ ]:

```
train_df.show(5)
```

```
+-----------+--------+------+--------------------+------+----+-----+-----+----
------------+-------+-----+--------+
|PassengerId|Survived|Pclass|                Name|   Sex| Age|SibSp|Parch|
Ticket|   Fare|Cabin|Embarked|
+-----------+--------+------+--------------------+------+----+-----+-----+----
------------+-------+-----+--------+
|          1|       0|     3|Braund, Mr. Owen ...|  male|22.0|    1|    0|
A/5 21171|   7.25| null|       S|
|          2|       1|     1|Cumings, Mrs. Joh...|female|38.0|    1|    0|
PC 17599|71.2833|  C85|       C|
|          3|       1|     3|Heikkinen, Miss. ...|female|26.0|    0|    0|STO
N/O2. 3101282|  7.925| null|       S|
|          4|       1|     1|Futrelle, Mrs. Ja...|female|35.0|    1|    0|
113803|   53.1| C123|       S|
|          5|       0|     3|Allen, Mr. Willia...|  male|35.0|    0|    0|
373450|   8.05| null|       S|
+-----------+--------+------+--------------------+------+----+-----+-----+----
------------+-------+-----+--------+
only showing top 5 rows
```

We also have the option to convert the dataframe to a Pandas dataframe. Limit() is a transformation, whereas toPandas() is an action.

In [ ]:

```
train_df.limit(5).toPandas()
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Ca
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	N
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	N
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	N

In [ ]:

```
train_df.select('Survived', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare').summary(
```

```
+-------+-------------------+------------------+------------------+-----------
-------+-------------------+-----------------+
|summary|           Survived|            Pclass|               Age|
SibSp|              Parch|             Fare|
+-------+-------------------+------------------+------------------+-----------
-------+-------------------+-----------------+
|  count|                891|               891|               714|
891|                891|              891|
|   mean| 0.3838383838383838| 2.308641975308642| 29.69911764705882|0.523007856
3411896|0.38159371492704824| 32.2042079685746|
| stddev|0.48659245426485753|0.8360712409770491|14.526497332334035|1.102743432
2934315| 0.8060572211299488|49.69342859718089|
|    min|                  0|                 1|              0.42|
0|                  0|              0.0|
|    25%|                  0|                 2|              20.0|
0|                  0|           7.8958|
|    50%|                  0|                 3|              28.0|
0|                  0|          14.4542|
|    75%|                  1|                 3|              38.0|
1|                  0|             31.0|
|    max|                  1|                 3|              80.0|
8|                  6|         512.3292|
+-------+-------------------+------------------+------------------+-----------
-------+-------------------+-----------------+
```

As we observe the summary for the integer and double columns (except PassengerId):

- Survived seemed to have more passengers not survived the crash
- Pclass has more passengers in the 3rd class than the other 2

- Both the median and mean of Age are in the late 20s, and the youngest passenger was less than 1 year old, whereas the oldest passenger was 80 years old
- SibSp seems to mostly have 0 or 1 siblings or spouses with a max of 8
- Parch shows that most passengers came with no parents or children with the maximum of 6.
- Fare is skewed to the right as the mean is greater than the median, and the highest fare price was 512.

EDA

In [ ]:
```
train_df.groupBy('Survived').count().show()
```

```
+--------+-----+
|Survived|count|
+--------+-----+
|       1|  342|
|       0|  549|
+--------+-----+
```

As we can see from these counts, more passengers did not survive.

In [ ]:
```
train_df.groupBy('Survived').mean('Age', 'Fare').show()
```

```
+--------+------------------+------------------+
|Survived|          avg(Age)|         avg(Fare)|
+--------+------------------+------------------+
|       1|28.343689655172415| 48.39540760233917|
|       0| 30.62617924528302|22.117886885245877|
+--------+------------------+------------------+
```

When we breakdown the survivors by average Age and Fare, we can observe that although the average age between survived or not are about the same, it is apparent that those who paid more for their tickets were more likely to survive.

In [ ]:
```
train_df.groupBy('Survived').pivot('Pclass').count().show()
```

```
+--------+---+---+---+
|Survived|  1|  2|  3|
+--------+---+---+---+
|       1|136| 87|119|
|       0| 80| 97|372|
+--------+---+---+---+
```

First class passengers had the most likely chance to survive compared to the other two classes.

In [ ]:
```
train_df.groupBy('Survived').pivot('Sex').count().show()
```

```
+--------+------+----+
|Survived|female|male|
+--------+------+----+
|       1|   233| 109|
|       0|    81| 468|
+--------+------+----+
```

Females had a higher chance of surviving than males.

In [ ]:
```
train_df.groupBy('Survived').pivot('Parch').count().show()
```

```
+--------+---+---+---+---+----+---+----+
|Survived|  0|  1|  2|  3|   4|  5|   6|
+--------+---+---+---+---+----+---+----+
|       1|233| 65| 40|  3|null|  1|null|
|       0|445| 53| 40|  2|   4|  4|   1|
+--------+---+---+---+---+----+---+----+
```

In [ ]:
```
train_df.groupBy('Survived').pivot('SibSp').count().show()
```

```
+--------+---+---+---+---+---+----+----+
|Survived|  0|  1|  2|  3|  4|   5|   8|
+--------+---+---+---+---+---+----+----+
|       1|210|112| 13|  4|  3|null|null|
|       0|398| 97| 15| 12| 15|   5|   7|
+--------+---+---+---+---+---+----+----+
```

Those with a smaller family or without a partner were more likely to survive.

In [ ]:
```
train_df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in t
```

```
+-----------+--------+------+----+---+---+-----+-----+------+----+-----+------
--+
|PassengerId|Survived|Pclass|Name|Sex|Age|SibSp|Parch|Ticket|Fare|Cabin|Embark
ed|
+-----------+--------+------+----+---+---+-----+-----+------+----+-----+------
--+
|          0|       0|     0|   0|  0|177|    0|    0|     0|   0|  687|
2|
+-----------+--------+------+----+---+---+-----+-----+------+----+-----+------
--+
```

After checking for missing values, Age, Cabin, and Embarked had missing values, thus we must decide how to deal with these values.

In [ ]:
```
train_df.select(count(train_df.Cabin)).show()
```

```
+------------+
|count(Cabin)|
+------------+
|         204|
+------------+
```

Since Cabin has about 77% data missing, and Cabin and Pclass are similar, we can drop Cabin.

In [ ]:
```
train_df = train_df.drop('Cabin')
```

In [ ]:
```
train_df.groupBy('Embarked').count().show()
```

```
+--------+-----+
|Embarked|count|
+--------+-----+
|       Q|   77|
|    null|    2|
|       C|  168|
|       S|  644|
+--------+-----+
```

In [ ]:
```
train_df.select('Fare').summary('50%').show()
```

```
+-------+-------+
|summary|   Fare|
+-------+-------+
|    50%|14.4542|
+-------+-------+
```

Since S is the majority of the Embarked column, we will replace the two null values with S. There are missing values in test data for Fare so we will fill in those missing values with the median value.

In [ ]:
```
train_df = train_df.fillna({'Embarked':'S', 'Fare': 14.45})
```

We will group similar titles together and assign the average age to the missing values for that title. We will first extract the titles using regular expression, and see the count and average age for each title.

In [ ]:
```
train_df = train_df.withColumn('Title', regexp_extract(train_df['Name'], '([A-Z
train_df.groupBy('Title').agg(count('Age'), mean('Age')).sort(col('count(Age)')
```

```
+--------+---------+------------------+
|   Title|count(Age)|          avg(Age)|
+--------+---------+------------------+
|      Mr|      398|32.368090452261306|
|    Miss|      146|21.773972602739725|
|     Mrs|      108|35.898148148148145|
|  Master|       36| 4.574166666666667|
|     Rev|        6|43.166666666666664|
|      Dr|        6|              42.0|
|     Col|        2|              58.0|
|    Mlle|        2|              24.0|
|   Major|        2|              48.5|
|     Don|        1|              40.0|
|Countess|        1|              33.0|
|    Lady|        1|              48.0|
|Jonkheer|        1|              38.0|
|     Mme|        1|              24.0|
|    Capt|        1|              70.0|
|      Ms|        1|              28.0|
|     Sir|        1|              49.0|
+--------+---------+------------------+
```

The top 3 titles are Mr, Miss, and Mrs which account for most of the titles of passengers. Master is lower in count than the top three, however, the average age is much lower which could account for a different group of passengers. Thus, we will map the other titles to the top 4 titles.

In [ ]:

```
title_dictionary = {'Mr':'Mr', 'Miss':'Miss', 'Mrs':'Mrs','Master':'Master','Re
title_map = create_map([lit(x) for x in chain(*title_dictionary.items())])
train_df = train_df.withColumn('Title', title_map[train_df['Title']])
train_df.groupBy('Title').mean('Age').show()
```

```
+------+------------------+
| Title|          avg(Age)|
+------+------------------+
|  Miss|             21.86|
|Master| 4.574166666666667|
|    Mr| 33.02272727272727|
|   Mrs|35.981818181818184|
+------+------------------+
```

We will create a function called age_imputer() which will fill in the missing values of Age for each given title.

In [ ]:

```
def age_imputer(df, title, age):
    return df.withColumn('Age', when((df['Age'].isNull()) & (df['Title'] == tit
```

In [ ]:

```
train_df = age_imputer(train_df, 'Mr', 33.02)
train_df = age_imputer(train_df, 'Miss', 21.86)
train_df = age_imputer(train_df, 'Mrs', 35.98)
train_df = age_imputer(train_df, 'Master', 4.57)
```

Will create a new column named FamilySize to have a count of total members of a family using columns Parch and SibSpl.

```
In [ ]: train_df = train_df.withColumn('FamilySize', train_df['Parch'] + train_df['SibS
```

We are also dropping other columns we do not need for this analysis.

```
In [ ]: train_df = train_df.drop('PassengerID', 'Name', 'Ticket', 'Title')
```

```
In [ ]: train_df.show(5)
```

```
+--------+------+------+----+-------+--------+----------+
|Survived|Pclass|   Sex| Age|   Fare|Embarked|FamilySize|
+--------+------+------+----+-------+--------+----------+
|       0|     3|  male|22.0|   7.25|       S|         1|
|       1|     1|female|38.0|71.2833|       C|         1|
|       1|     3|female|26.0|  7.925|       S|         0|
|       1|     1|female|35.0|   53.1|       S|         1|
|       0|     3|  male|35.0|   8.05|       S|         0|
+--------+------+------+----+-------+--------+----------+
only showing top 5 rows
```

```
In [ ]: train_df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in t
```

```
+--------+------+---+---+----+--------+----------+
|Survived|Pclass|Sex|Age|Fare|Embarked|FamilySize|
+--------+------+---+---+----+--------+----------+
|       0|     0|  0|  0|   0|       0|         0|
+--------+------+---+---+----+--------+----------+
```

We now have no missing values and the columns that we need, so we will move on to modeling.

Modeling

We will first convert the 'Sex' and 'Embarked' columns from string to numeric index.

```
In [ ]: strInd = StringIndexer(inputCols=['Sex', 'Embarked'], outputCols=['SexNum', 'Em
        strInd_mod = strInd.fit(train_df)

        train_df_new = strInd_mod.transform(train_df).drop('Sex', 'Embarked')
        train_df_new.show(5)
```

```
+--------+------+----+-------+----------+------+-----------+
|Survived|Pclass| Age|   Fare|FamilySize|SexNum|EmbarkedNum|
+--------+------+----+-------+----------+------+-----------+
|       0|     3|22.0|   7.25|         1|   0.0|        0.0|
|       1|     1|38.0|71.2833|         1|   1.0|        1.0|
|       1|     3|26.0|  7.925|         0|   1.0|        0.0|
|       1|     1|35.0|   53.1|         1|   1.0|        0.0|
|       0|     3|35.0|   8.05|         0|   0.0|        0.0|
+--------+------+----+-------+----------+------+-----------+
only showing top 5 rows
```

We are going to use VectorAssembler because in scikit learn, it takes X and y in a separation matrix. Usually, y is a column vector and X is a matrix. But for Spark API, X and y has to be in a single matrix instead of two for the training data. It only accepts X in the prediction part. X should also be a vector in each row of the dataframe. We cannot directly feed the dataframe to the model.

In [ ]:
```
vec_assemble = VectorAssembler(inputCols=train_df_new.columns[1:], outputCol='f
train_df_new = vec_assemble.transform(train_df_new).select('features', 'Survive
train_df_new.show(5, truncate=False)
```

```
+------------------------------+--------+
|features                      |Survived|
+------------------------------+--------+
|[3.0,22.0,7.25,1.0,0.0,0.0]   |0       |
|[1.0,38.0,71.2833,1.0,1.0,1.0]|1       |
|[3.0,26.0,7.925,0.0,1.0,0.0]  |1       |
|[1.0,35.0,53.1,1.0,1.0,0.0]   |1       |
|[3.0,35.0,8.05,0.0,0.0,0.0]   |0       |
+------------------------------+--------+
only showing top 5 rows
```

In [ ]:
```
# Split data into training and validation first. Will use the test dataset late
train_df_sub, validation_df = train_df_new.randomSplit([0.8, 0.2], seed = 0)
```

In [ ]:
```
train_df_sub.show(5, truncate=False)
```

```
+---------------------+--------+
|features             |Survived|
+---------------------+--------+
|(6,[0,1],[1.0,33.02])|0       |
|(6,[0,1],[1.0,33.02])|0       |
|(6,[0,1],[1.0,38.0]) |0       |
|(6,[0,1],[1.0,39.0]) |0       |
|(6,[0,1],[1.0,40.0]) |0       |
+---------------------+--------+
only showing top 5 rows
```

Logistic Modeling

For logistic modeling, we will use both ridge regression and lasso regression to predict whether a passenger will survive or not. Ridge regression will keep all of the features when predicting and reduces the magnitude of coefficients towards zero, whereas lasso regression will shrink the less important feature's coefficient to zero, performing feature selection.

We will first use MulticlassClassificationEvaluator() and specify that we are looking to evaluate accuracy.

In [ ]:
```
evaluator = MulticlassClassificationEvaluator(labelCol = 'Survived', metricName
```

In [ ]:
```
rid_log = LogisticRegression(labelCol='Survived', maxIter=100, elasticNetParam=
ridge_model = rid_log.fit(train_df_sub)
ridge_pred = ridge_model.transform(validation_df)
evaluator.evaluate(ridge_pred)
```

Out[ ]: 0.8021390374331551

```
In [ ]: lasso_log = LogisticRegression(labelCol='Survived', maxIter=100, elasticNetPara
        lasso_model = lasso_log.fit(train_df_sub)
        lasso_pred = lasso_model.transform(validation_df)
        evaluator.evaluate(lasso_pred)
```

Out[ ]: 0.8074866310160428

Gradient Boosting

We will try to see if gradient boosting will give us better results than logistic regression. Gradient boosting is a prediction model in the form of an ensemble of weak prediction models, usually decision trees.

```
In [ ]: gb = GBTClassifier(labelCol='Survived', maxIter=75, maxDepth=2)
        gb_model = gb.fit(train_df_sub)
        gb_pred = gb_model.transform(validation_df)
        evaluator.evaluate(gb_pred)
```

Out[ ]: 0.8181818181818182

As shown, gradient boosting gave the best result, thus we will move forward with that and predict the test dataset.

Prediction

We will proceed with the same data preprocessing techniques we performed on the training dataset and run our gradient boosting model on this dataset.

```
In [ ]: test_df.show(5)
```

```
+-----------+------+--------------------+------+----+-----+-----+-------+-----
--+-----+--------+
|PassengerId|Pclass|                Name|   Sex| Age|SibSp|Parch| Ticket|   Fa
re|Cabin|Embarked|
+-----------+------+--------------------+------+----+-----+-----+-------+-----
--+-----+--------+
|        892|     3|    Kelly, Mr. James|  male|34.5|    0|    0| 330911| 7.82
92| null|       Q|
|        893|     3|Wilkes, Mrs. Jame...|female|47.0|    1|    0| 363272|
7.0| null|       S|
|        894|     2|Myles, Mr. Thomas...|  male|62.0|    0|    0| 240276| 9.68
75| null|       Q|
|        895|     3|    Wirz, Mr. Albert|  male|27.0|    0|    0| 315154| 8.66
25| null|       S|
|        896|     3|Hirvonen, Mrs. Al...|female|22.0|    1|    1|3101298|12.28
75| null|       S|
+-----------+------+--------------------+------+----+-----+-----+-------+-----
--+-----+--------+
only showing top 5 rows
```

```
In [ ]: test_df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in te
```

```
+-----------+------+----+---+---+-----+-----+------+----+-----+--------+
|PassengerId|Pclass|Name|Sex|Age|SibSp|Parch|Ticket|Fare|Cabin|Embarked|
+-----------+------+----+---+---+-----+-----+------+----+-----+--------+
|          0|     0|   0|  0| 86|    0|    0|     0|   1|  327|       0|
+-----------+------+----+---+---+-----+-----+------+----+-----+--------+
```

```
In [ ]: test_df = test_df.fillna({'Embarked':'S', 'Fare': 14.45})
```

```
In [ ]: test_df = test_df.withColumn('Title', regexp_extract(test_df['Name'], '([A-Za-z
        test_df = test_df.withColumn('Title', title_map[test_df['Title']])
```

```
In [ ]: test_df = age_imputer(test_df, 'Mr', 33.02)
        test_df = age_imputer(test_df, 'Miss', 21.86)
        test_df = age_imputer(test_df, 'Mrs', 35.98)
        test_df = age_imputer(test_df, 'Master', 4.57)
```

```
In [ ]: test_df = test_df.withColumn('FamilySize', test_df['Parch'] + test_df['SibSp'])
```

```
In [ ]: test_df = test_df.drop('Name', 'Ticket', 'Title', 'Cabin')
```

```
In [ ]: test_df.show(5)
```

```
+-----------+------+------+----+-------+--------+----------+
|PassengerId|Pclass|   Sex| Age|   Fare|Embarked|FamilySize|
+-----------+------+------+----+-------+--------+----------+
|        892|     3|  male|34.5| 7.8292|       Q|         0|
|        893|     3|female|47.0|    7.0|       S|         1|
|        894|     2|  male|62.0| 9.6875|       Q|         0|
|        895|     3|  male|27.0| 8.6625|       S|         0|
|        896|     3|female|22.0|12.2875|       S|         2|
+-----------+------+------+----+-------+--------+----------+
only showing top 5 rows
```

```
In [ ]: test_df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in te
```

```
+-----------+------+---+---+----+--------+----------+
|PassengerId|Pclass|Sex|Age|Fare|Embarked|FamilySize|
+-----------+------+---+---+----+--------+----------+
|          0|     0|  0|  0|   0|       0|         0|
+-----------+------+---+---+----+--------+----------+
```

We keep the PassengerId column as we need to map this to our results at the end.

Creating a Pipeline

We wil create a pipeline to have operations performed in a specific order, and in our case, we will have StringIndexer, VectorAssembler, and our gradient boosting model in one pipeline. We will also perform a cross-validated grid search over a parameter grid to find the best hyperparameters to create the best model.

In [ ]:
```
pipeline_gb = Pipeline(stages=[strInd, vec_assemble, gb])

paramGrid = (ParamGridBuilder()
             .addGrid(gb.maxIter,[5, 10, 15])
             .addGrid(gb.maxDepth,[2,3,5])
             .addGrid(gb.maxBins,[20, 30, 50])
             .build())

opt_model = CrossValidator(estimator=pipeline_gb, estimatorParamMaps=paramGrid,

final_model = opt_model.fit(train_df)
pred_train = final_model.transform(train_df)
evaluator.evaluate(pred_train)
# best parameters maxIter: 10, maxDepth: 3, maxBins: 50
```

Out[ ]:
```
0.8451178451178452
```

With the in-sample accuracy at about 84.5%, we will use this model with these hyperparameters on the test dataset and map the PassengerId with prediction.

In [ ]:
```
pred_test = final_model.transform(test_df)
preds = pred_test.select('PassengerId', 'prediction')
preds = preds.withColumn('Survived', preds['prediction'].cast('integer')).drop(
preds.show(5)
```

```
+-----------+--------+
|PassengerId|Survived|
+-----------+--------+
|        892|       0|
|        893|       0|
|        894|       0|
|        895|       0|
|        896|       1|
+-----------+--------+
only showing top 5 rows
```

We will output the results to Pandas format and then to csv format. The file will be named results.csv.

In [ ]:
```
preds.toPandas().to_csv('results.csv', index=False)
```

We can now read in the file using the read_csv function from Pandas.

In [ ]:
```
pd.read_csv('results.csv').head(5)
```

Out[ ]:

	PassengerId	Survived
0	892	0
1	893	0
2	894	0
3	895	0
4	896	1

Conclusion

PySpark is a great tool to perform data preprocessing and machine learning, especially when it comes to big data. In this case, we were able to create a reliable model to predict if a passenger will survive or not based on the features we have. When we work with big data in the future, it would be best to utilize the PySpark library to have the best performance with data processing and data modeling.