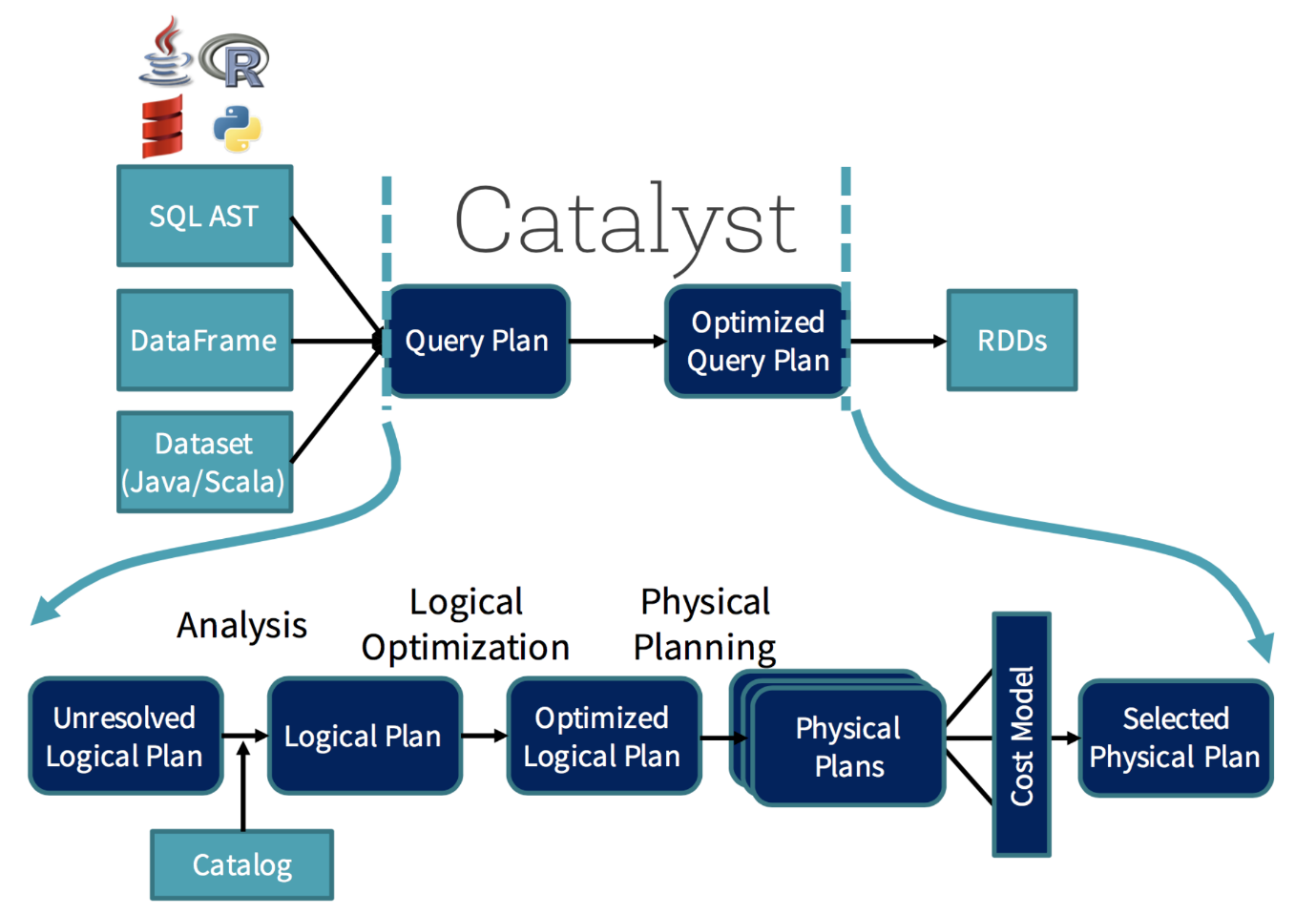
Your First Apache Spark ML Model

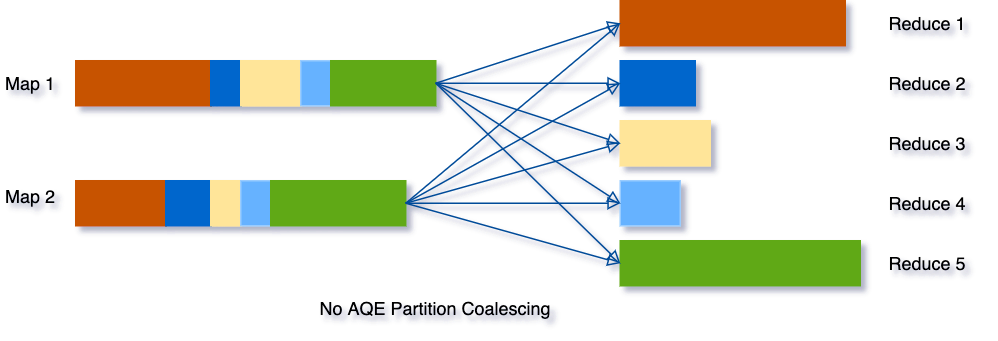


n Spark 3.0 we will get something called “Adaptive Query Execution” (AQE) that will reoptimize and adjust query plans based on runtime statistics collected in the process of query execution. This is going to be huge on performance, for [example](https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html) let’s say we are running the query

SELECT max(i) FROM table GROUP BY column

Without AQE, Spark will start five tasks to do the final aggregation:

Image for post



<https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html>

But with AQE, Spark will coalesce these three small partitions into one and, as a result, the final aggregation now only needs to perform three tasks rather than five:

Image for post



Spark’s library for machine learning is called MLlib (Machine Learning library). It’s heavily based on Scikit-learn’s ideas on pipelines. In this library to create an ML model the basics concepts are:

* DataFrame: This ML API uses DataFrame from Spark SQL as an ML dataset, which can hold a variety of data types. E.g., a DataFrame could have different columns storing text, feature vectors, true labels, and predictions.
* Transformer: A Transformer is an algorithm that can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer that transforms a DataFrame with features into a DataFrame with predictions.
* Estimator: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model
* Pipeline: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow
* Parameter: All Transformers and Estimators now share a common API for specifying parameters.

If you want to know more about the APIs and how they work check the [official documentation](https://spark.apache.org/docs/latest/ml-pipeline.html).

For this example, we will use a very basic dataset. The Titanic dataset, hopefully, you are all familiar with the case and the data. To start we have to download the data, for that we are using Kaggle:

**We will be predicting if a passenger survived or not depending on its features.**

**Before starting: Make sure to close and stop all other Spark notebooks. Java can complain sometimes when working with multiple instances of Spark.**

## Loading the data into Spark

To load the data we are using Spark DataFrames. Spark it’s a little bit more complicated than Pandas. You can’t just do “import -> read\_csv()”. You first need to start a Spark Session, to do that write:

+--------+------+---------------------------------------------------+------+---+-----+-----+----------------+-------+-----+--------+

|survived|pclass|name |sex |age|sibsp|parch|ticket |fare |cabin|embarked|

+--------+------+---------------------------------------------------+------+---+-----+-----+----------------+-------+-----+--------+

|0 |3 |Braund, Mr. Owen Harris |male |22 |1 |0 |A/5 21171 |7.25 |null |S |

|1 |1 |Cumings, Mrs. John Bradley (Florence Briggs Thayer)|female|38 |1 |0 |PC 17599 |71.2833|C85 |C |

|1 |3 |Heikkinen, Miss. Laina |female|26 |0 |0 |STON/O2. 3101282|7.925 |null |S |

|1 |1 |Futrelle, Mrs. Jacques Heath (Lily May Peel) |female|35 |1 |0 |113803 |53.1 |C123 |S |

|0 |3 |Allen, Mr. William Henry |male |35 |0 |0 |373450 |8.05 |null |S |

+--------+------+---------------------------------------------------+------+---+-----+-----+----------------+-------+-----+--------+

only showing top 5 rows

titanic\_pd = titanic\_df.toPandas()  
titanic\_pd.head(5)

<https://stackoverflow.com/questions/53807854/how-to-fix-importerror-pandas-0-19-2-must-be-installed-however-it-was-not>

<https://stackoverflow.com/questions/39905931/dataframe-head-not-shown-in-pycharm/39906582>

<https://stackoverflow.com/questions/11707586/how-do-i-expand-the-output-display-to-see-more-columns-of-a-pandas-dataframe>

import pandas as pd  
##pip install pandas  
pd.set\_option('display.expand\_frame\_repr', False)  
titanic\_pd = titanic\_df.toPandas()  
print(titanic\_pd.head(5))

survived pclass name sex age sibsp parch ticket fare cabin embarked

0 0 3 Braund, Mr. Owen Harris male 22 1 0 A/5 21171 7.25 None S

1 1 1 Cumings, Mrs. John Bradley (Florence Briggs Th... female 38 1 0 PC 17599 71.2833 C85 C

2 1 3 Heikkinen, Miss. Laina female 26 0 0 STON/O2. 3101282 7.925 None S

3 1 1 Futrelle, Mrs. Jacques Heath (Lily May Peel) female 35 1 0 113803 53.1 C123 S

4 0 3 Allen, Mr. William Henry male 35 0 0 373450 8.05 None S

print(titanic\_pd.count())

survived 891

pclass 891

name 891

sex 891

age 714

sibsp 891

parch 891

ticket 891

fare 891

cabin 204

embarked 889

dtype: int64

print(titanic\_pd.columns)  
print(titanic\_pd.dtypes)

Index(['survived', 'pclass', 'name', 'sex', 'age', 'sibsp', 'parch', 'ticket',

'fare', 'cabin', 'embarked'],

dtype='object')

survived object

pclass object

name object

sex object

age object

sibsp object

parch object

ticket object

fare object

cabin object

embarked object

dtype: object

print(titanic\_pd.describe())

survived pclass name sex age sibsp parch ticket fare cabin embarked

count 891 891 891 891 714 891 891 891 891 204 889

unique 2 3 891 2 88 7 7 681 248 147 3

top 0 3 Pinsky, Mrs. (Rosa) male 24 0 0 CA. 2343 8.05 B96 B98 S

freq 549 491 1 577 30 608 678 7 43 4 644

## Data preparation and feature engineering

One of the things we noticed from the data exploration from above was that all the columns were of String type. But that doesn’t seem right. Some of them should be numeric. So we are going to cast them. Also because of time I’m only selecting a few variables for modeling so we don’t have to deal with the whole dataset:

#titanic\_df.show(5, truncate=False)  
import pandas as pd  
##pip install pandas  
pd.set\_option('display.expand\_frame\_repr', False)  
titanic\_pd = titanic\_df.toPandas()

titanicDataSet = titanic\_df.select(F.col('survived').cast('float'),  
 F.col('pclass').cast('float'), F.col('sex'), F.col('age').cast('float'), F.col('fare').cast('float'), F.col('embarked'))

titanicDataSet.show(10)

+--------+------+------+----+-------+--------+

|survived|pclass| sex| age| fare|embarked|

+--------+------+------+----+-------+--------+

| 0.0| 3.0| male|22.0| 7.25| S|

| 1.0| 1.0|female|38.0|71.2833| C|

| 1.0| 3.0|female|26.0| 7.925| S|

| 1.0| 1.0|female|35.0| 53.1| S|

| 0.0| 3.0| male|35.0| 8.05| S|

| 0.0| 3.0| male|null| 8.4583| Q|

| 0.0| 1.0| male|54.0|51.8625| S|

| 0.0| 3.0| male| 2.0| 21.075| S|

| 1.0| 3.0|female|27.0|11.1333| S|

| 1.0| 2.0|female|14.0|30.0708| C|

+--------+------+------+----+-------+--------+

only showing top 10 rows

titanicDataSet.select([count(when(isnull(c), c)).alias(c) for c in titanicDataSet.columns]).show()

--------+------+---+---+----+--------+

|survived|pclass|sex|age|fare|embarked|

+--------+------+---+---+----+--------+

| 0| 0| 0|177| 0| 2|

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

**dropna**(*how='any'*, *thresh=None*, *subset=None*)[[source]](https://spark.apache.org/docs/latest/api/python/_modules/pyspark/sql/dataframe.html#DataFrame.dropna)

Returns a new **[DataFrame](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** omitting rows with null values. **[DataFrame.dropna()](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame.dropna" \o "pyspark.sql.DataFrame.dropna)** and **[DataFrameNaFunctions.drop()](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrameNaFunctions.drop" \o "pyspark.sql.DataFrameNaFunctions.drop)** are aliases of each other.

**Parameters**

* **how** – ‘any’ or ‘all’. If ‘any’, drop a row if it contains any nulls. If ‘all’, drop a row only if all its values are null.
* **thresh** – int, default None If specified, drop rows that have less than *thresh* non-null values. This overwrites the *how* parameter.
* **subset** – optional list of column names to consider.

**replace**(*to\_replace*, *value=<no value>*, *subset=None*)[[source]](https://spark.apache.org/docs/latest/api/python/_modules/pyspark/sql/dataframe.html#DataFrame.replace)

Returns a new **[DataFrame](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** replacing a value with another value. [**DataFrame.replace()**](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.replace) and [**DataFrameNaFunctions.replace()**](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameNaFunctions.replace) are aliases of each other. Values to\_replace and value must have the same type and can only be numerics, booleans, or strings. Value can have None. When replacing, the new value will be cast to the type of the existing column. For numeric replacements all values to be replaced should have unique floating point representation. In case of conflicts (for example with *{42: -1, 42.0: 1}*) and arbitrary replacement will be used.

**Parameters**

* **to\_replace** – bool, int, long, float, string, list or dict. Value to be replaced. If the value is a dict, then *value* is ignored or can be omitted, and *to\_replace* must be a mapping between a value and a replacement.
* **value** – bool, int, long, float, string, list or None. The replacement value must be a bool, int, long, float, string or None. If *value* is a list, *value* should be of the same length and type as *to\_replace*. If *value* is a scalar and *to\_replace* is a sequence, then *value* is used as a replacement for each item in *to\_replace*.
* **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored.

>>>

**>>>** df4.na.replace(10, 20).show()

+----+------+-----+

| age|height| name|

+----+------+-----+

| 20| 80|Alice|

| 5| null| Bob|

|null| null| Tom|

|null| null| null|

+----+------+-----+

We see that we also have null values in some columns, so we will just eliminate them:

titanicDataSet.replace('?', None).dropna(how='any')

titanicDataSet = titanicDataSet.replace('?', None).dropna(how='any')

+--------+------+---+---+----+--------+

|survived|pclass|sex|age|fare|embarked|

+--------+------+---+---+----+--------+

| 0| 0| 0| 0| 0| 0|

+--------+------+---+---+----+--------+

Now, the Spark ML library only works with numeric data. But we still want to use the Sex and the Embarked column. For that, we will need to encode them. To do it let’s use something called the [StringIndexer](https://spark.apache.org/docs/latest/ml-features#stringindexer):

from pyspark.ml.feature import StringIndexer  
titanicDataSet = StringIndexer(  
 inputCol='sex',  
 outputCol='Gender',  
 handleInvalid='keep').fit(titanicDataSet).transform(titanicDataSet)  
  
titanicDataSet = StringIndexer(  
 inputCol='embarked',  
 outputCol='Boarded',  
 handleInvalid='keep').fit(titanicDataSet).transform(titanicDataSet)  
  
titanicDataSet.show(10)

+--------+------+------+----+-------+--------+------+-------+

|survived|pclass| sex| age| fare|embarked|Gender|Boarded|

+--------+------+------+----+-------+--------+------+-------+

| 0.0| 3.0| male|22.0| 7.25| S| 0.0| 0.0|

| 1.0| 1.0|female|38.0|71.2833| C| 1.0| 1.0|

| 1.0| 3.0|female|26.0| 7.925| S| 1.0| 0.0|

| 1.0| 1.0|female|35.0| 53.1| S| 1.0| 0.0|

| 0.0| 3.0| male|35.0| 8.05| S| 0.0| 0.0|

| 0.0| 1.0| male|54.0|51.8625| S| 0.0| 0.0|

| 0.0| 3.0| male| 2.0| 21.075| S| 0.0| 0.0|

| 1.0| 3.0|female|27.0|11.1333| S| 1.0| 0.0|

| 1.0| 2.0|female|14.0|30.0708| C| 1.0| 1.0|

| 1.0| 3.0|female| 4.0| 16.7| S| 1.0| 0.0|

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

So all the columns we want are numeric. We now have to get rid of the old columns “Sex” and “Embarked” because we won’t be using them:

+--------+------+----+-------+------+-------+

|survived|pclass| age| fare|Gender|Boarded|

+--------+------+----+-------+------+-------+

| 0.0| 3.0|22.0| 7.25| 0.0| 0.0|

| 1.0| 1.0|38.0|71.2833| 1.0| 1.0|

| 1.0| 3.0|26.0| 7.925| 1.0| 0.0|

| 1.0| 1.0|35.0| 53.1| 1.0| 0.0|

| 0.0| 3.0|35.0| 8.05| 0.0| 0.0|

| 0.0| 1.0|54.0|51.8625| 0.0| 0.0|

| 0.0| 3.0| 2.0| 21.075| 0.0| 0.0|

| 1.0| 3.0|27.0|11.1333| 1.0| 0.0|

| 1.0| 2.0|14.0|30.0708| 1.0| 1.0|

| 1.0| 3.0| 4.0| 16.7| 1.0| 0.0|

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

And you want to predict “Survived”, you need to combine the information of the columns “Pclass”, “Age”, “Fare”, “Gender” and “Boarded” into one column. We normally call that column features and it should look like this:

+--------+------+----+-------+------+-------+----------------------+  
|Survived|Pclass| Age| Fare|Gender|Boarded| features |  
+--------+------+----+-------+------+-------+----------------------+  
| 0.0| 3.0|22.0| 7.25| 0.0| 0.0|[3.0, 22.0, 7.25, 0, 0] |  
+--------+------+----+-------+------+-------+----------------------+

ut one step left before going into the machine learning part. Spark actually works to predict with a column with all the features smashed together into a list-like structure. For example, if you have the features:

+--------+------+----+-------+------+-------+  
|Survived|Pclass| Age| Fare|Gender|Boarded|  
+--------+------+----+-------+------+-------+  
| 0.0| 3.0|22.0| 7.25| 0.0| 0.0|  
+--------+------+----+-------+------+-------+

And you want to predict “Survived”, you need to combine the information of the columns “Pclass”, “Age”, “Fare”, “Gender” and “Boarded” into one column. We normally call that column features and it should look like this:

+--------+------+----+-------+------+-------+----------------------+  
|Survived|Pclass| Age| Fare|Gender|Boarded| features |  
+--------+------+----+-------+------+-------+----------------------+  
| 0.0| 3.0|22.0| 7.25| 0.0| 0.0|[3.0, 22.0, 7.25, 0, 0] |  
+--------+------+----+-------+------+-------+----------------------+

As you can see the new column features contain the same information from all of our features but in a list-like object. To do that in Spark we use the [VectorAssembler](https://spark.apache.org/docs/latest/ml-features" \l "vectorassembler):

# Assemble all the features with VectorAssemblerrequired\_features = ['Pclass',  
 'Age',  
 'Fare',  
 'Gender',  
 'Boarded'  
 ]from pyspark.ml.feature import VectorAssemblerassembler = VectorAssembler(inputCols=required\_features, outputCol='features')transformed\_data = assembler.transform(dataset)

+--------+------+----+-------+------+-------+--------------------+

|survived|pclass| age| fare|Gender|Boarded| features|

+--------+------+----+-------+------+-------+--------------------+

| 0.0| 3.0|22.0| 7.25| 0.0| 0.0|[3.0,22.0,7.25,0....|

| 1.0| 1.0|38.0|71.2833| 1.0| 1.0|[1.0,38.0,71.2833...|

| 1.0| 3.0|26.0| 7.925| 1.0| 0.0|[3.0,26.0,7.92500...|

| 1.0| 1.0|35.0| 53.1| 1.0| 0.0|[1.0,35.0,53.0999...|

| 0.0| 3.0|35.0| 8.05| 0.0| 0.0|[3.0,35.0,8.05000...|

| 0.0| 1.0|54.0|51.8625| 0.0| 0.0|[1.0,54.0,51.8624...|

| 0.0| 3.0| 2.0| 21.075| 0.0| 0.0|[3.0,2.0,21.07500...|

| 1.0| 3.0|27.0|11.1333| 1.0| 0.0|[3.0,27.0,11.1332...|

| 1.0| 2.0|14.0|30.0708| 1.0| 1.0|[2.0,14.0,30.0708...|

| 1.0| 3.0| 4.0| 16

## Modeling

Now for the fun part right? NO! Haha. Modeling is important but without all the previous steps it would be impossible. So have fun in all the steps :)

Before modeling let’s do the usual splitting between training and testing:

(training\_data, test\_data) = transformed\_data.randomSplit([0.8,0.2])

Ok. Modeling. That means, in this case, build and fit an ML model to our dataset to predict the “Survived” columns with all the other ones. We will be using a [Random Forest Classifier](https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier). This is actually an [estimator](https://spark.apache.org/docs/latest/ml-pipeline.html#estimators) that we have to fit.

This is actually the easy part:

from pyspark.ml.classification import RandomForestClassifierrf = RandomForestClassifier(labelCol='Survived',   
 featuresCol='features',  
 maxDepth=5)

Now we fit the model:

model = rf.fit(training\_data)

This will give us something called a [transformer](https://spark.apache.org/docs/latest/ml-pipeline.html#transformers). And finally, we predict using the test dataset:

predictions = model.transform(test\_data)

And that’s it! You did it. Congratulations :). Your first Spark ML model. Now let’s see how well we did. For that, we will use a basic metric called the [accuracy](https://en.wikipedia.org/wiki/Accuracy_and_precision):

# Evaluate our model  
from pyspark.ml.evaluation import MulticlassClassificationEvaluatorevaluator = MulticlassClassificationEvaluator(  
 labelCol='Survived',   
 predictionCol='prediction',   
 metricName='accuracy')

And we to get the accuracy we do:

accuracy = evaluator.evaluate(predictions)  
print('Test Accuracy = ', accuracy) -> 0.843

Our basic model is giving us an accuracy of 0.843. Not bad at all :). Test Accuracy = 0.8496732026143791

Process finished with exit code 0