1 Apache Spark

Lab Objective: *Dealing with massive amounts of data often requires parallelization and cluster computing; Apache Spark is an industry standard for doing just that. In this lab we introduce the basics of PySpark, Spark's Python API, including data structures, syntax, and use cases. Finally, we conclude with a brief introduction to the Spark Machine Learning Package.*

Apache Spark

Apache Spark is an open-source, general-purpose distributed computing system used for big data analytics. Spark is able to complete jobs substantially faster than previous big data tools (i.e. Apache Hadoop) because of its in-memory caching, and optimized query execution. Spark provides development APIs in Python, Java, Scala, and R. On top of the main computing framework, Spark provides machine learning, SQL, graph analysis, and streaming libraries.

Spark's Python API can be accessed through the PySpark package. Installation for local execution or remote connection to an existing cluster can be done with `conda` or `pip` commands.[1]

```
# PySpark installation with conda
>>> conda install -c conda-forge pyspark

# PySpark installation with pip
>>> pip install pyspark
```

PySpark

One major benefit of using PySpark is the ability to run it in an interactive environment. One such option is the interactive Spark shell that comes prepackaged with PySpark. To use the shell, simply run `pyspark` in the terminal. In the Spark shell you can run code one line at a time without the need to have a fully written program. This is a great way to get a feel for Spark. To get help with a function use `help(function)`; to exit the shell simply run `quit()`.

In the interactive shell, the `SparkSession` object - the main entrypoint to all Spark functionality - is available by default as `spark`. When running Spark in a standard Python script (or in IPython)

[1] See https://runawayhorse001.github.io/LearningApacheSpark/setup.html for detailed installation instructions.

you need to define this object explicitly. The code box below outlines how to do this. It is standard practice to name your `SparkSession` object `spark`.

It is important to note that when you are finished with a `SparkSession` you should end it by calling `spark.stop()`.

> **Note**
>
> While the interactive shell is very robust, it may be easier to learn Spark in an environment that you are more familiar with (like IPython). To do so, just use the code given below. Help can be accessed in the usual way for your environment. Just remember to `stop()` the `SparkSession`!

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

# instantiate your SparkSession object
>>> spark = SparkSession\
...          .builder\
...          .appName("app_name")\
...          .getOrCreate()

# stop your SparkSession
>>> spark.stop()
```

> **Note**
>
> The syntax
>
> ```
> >>> spark = SparkSession\
> ...          .builder\
> ...          .appName("app_name")\
> ...          .getOrCreate()
> ```
>
> is somewhat unusual. While this code can be written on a single line, it is often more readable to break it up when dealing with many chained operations; this is standard styling for Spark. Note that you *cannot* write a comment after a line continuation character `'\'`.

Resilient Distributed Datasets

The most fundamental data structure used in Apache Spark is the Resilient Distributed Dataset (RDD). RDDs are immutable distributed collections of objects. They are *resilient* because performing an operation on one RDD produces a *new* RDD without altering the original; if something goes wrong, you can always go back to your original RDD and restart. They are *distributed* because the data resides in logical partitions across multiple machines. While RDDs can be difficult to work with, they offer the most granular control of all the Spark data structures.

There are two main ways of creating RDDs. The first is reading a file directly into Spark and the second is parallelizing an existing collection (list, numpy array, pandas dataframe, etc.). We will use the Titanic dataset[2] in most of the examples throughout this lab. Please note that most of this lab will be taught via example; as such, you are **strongly encouraged** to follow along on your own computer. The example below shows various ways to load the Titanic dataset as an RDD.

```
# SparkSession available as spark
# load the data directly into an RDD
>>> titanic = spark.sparkContext.textFile('titanic.csv')

# the file is of the format
# Survived | Class | Name | Sex | Age | Siblings/Spouses Aboard | Parents/↩
    Children Aboard | Fare

>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
 '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# note that each element is a single string - not particularly useful
# one option is to first load the data into a numpy array
>>> np_titanic = np.loadtxt('titanic.csv', delimiter=',', dtype=list)

# use sparkContext to parallelize the data into 4 partitions
>>> titanic_parallelize = spark.sparkContext.parallelize(np_titanic, 4)

>>> titanic_parallelize.take(2)
[array(['0', '3', ..., 'male', '22', '1', '0', '7.25'], dtype=object),
 array(['1', '1', ..., 'female', '38', '1', '0', '71.2833'], dtype=object)]
```

Achtung!

Because Apache Spark partitions and distributes data, calling for the first `n` objects using the same code (such as `take(n)`) may yield different results on different computers (or even each time you run it on one computer). This is not something you should worry about; it is the result of variation in partitioning and will not affect data analysis.

RDD Operations

Transformations

There are two types of operations you can perform on RDDs: *transformations* and *actions*. Transformations are functions that produce new RDDs from existing ones. Transformations are also lazy; they are not executed until an *action* is performed. This allows Spark to boost performance by optimizing *how* a sequence of transformations is executed at runtime.

The most commonly used transformation is probably `map(func)`, which creates a new RDD by applying `func` to each element of the current RDD. This function, `func`, can be any callable python

[2] https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/problem12.html

function, though it is often implemented as a `lambda` function. Similarly, `flatMap(func)` creates an RDD with the flattened results of `map(func)`.

```
# use map() to format the data
>>> titanic = spark.sparkContext.textFile('titanic.csv')
>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
 '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# apply split(',') to each element of the RDD with map()
>>> titanic.map(lambda row: row.split(','))\
...         .take(2)
[['0', '3', 'Mr. Owen Harris Braund', 'male', '22', '1', '0', '7.25'],
 ['1', '1', ..., 'female', '38', '1', '0', '71.283']]

# compare to flatMap(), which flattens the results of each row
>>> titanic.flatMap(lambda row: row.split(','))\
...         .take(2)
['0', '3']
```

The `filter(func)` transformation returns a new RDD containing only the elements that satisfy `func`. In this case, `func` should be a callable python function that returns a Boolean. The elements of the RDD that evaluate to `True` are included in the new RDD while those that evaluate to `False` are excluded.

```
# create a new RDD containing only the female passengers
>>> titanic_f = titanic.filter(lambda row: row[3] == 'female')
>>> titanic_f.take(3)
[['1', '1', ..., 'female', '38', '1', '0', '71.2833'],
 ['1', '3', ..., 'female', '26', '0', '0', '7.925'],
 ['1', '1', ..., 'female', '35', '1', '0', '53.1']]
```

Note

A great transformation to help validate or explore your dataset is `distinct()`. This will return a new RDD containing only the distinct elements of the original. In the case of the Titanic dataset, if you did not know how many classes there were, you could do the following:

```
>>> titanic.map(lambda row: row[1])\
...         .distinct()\
...         .collect()
['1', '3', '2']
```

Spark Command	Transformation
`map(f)`	Returns a new RDD by applying `f` to each element of this RDD
`flatmap(f)`	Same as `map(f)`, except the results are flattened
`filter(f)`	Returns a new RDD containing only the elements that satisfy `f`
`distinct()`	Returns a new RDD containing the distinct elements of the original
`reduceByKey(f)`	Takes an RDD of `(key, val)` pairs and merges the values for each `key` using an associative and commutative reduce function `f`
`sortBy(f)`	Sorts this RDD by the given function `f`
`sortByKey(f)`	Sorts an RDD assumed to consist of `(key, val)` pairs by the given function `f`
`groupBy(f)`	Returns a new RDD of groups of items based on `f`
`groupByKey()`	Takes an RDD of `(key, val)` pairs and returns a new RDD with `(key, (val1, val2, ...))` pairs

```
# the following counts the number of passengers in each class
# note that this isn't necessarily the best way to do this

# create a new RDD of (pclass, 1) elements to count occurances
>>> pclass = titanic.map(lambda row: (row[1], 1))
>>> pclass.take(5)
[('3', 1), ('1', 1), ('3', 1), ('1', 1), ('3', 1)]

# count the members of each class
>>> pclass = pclass.reduceByKey(lambda x, y: x + y)
>>> pclass.collect()
[('3', 487), ('1', 216), ('2', 184)]

# sort by number of passengers in each class, ascending order
>>> pclass.sortBy(lambda row: row[1]).collect()
[('2', 184), ('1', 216), ('3', 487)]
```

Problem 1. You are now ready for the "Hello World!" of big data: word count!

Write a function that accepts the name of a text file with default `filename=huck_finn.txt`.[a] Load the file as a PySpark RDD, and count the number of occurrences of each word. Sort the words by count, in descending order, and return a list of the `(word, count)` pairs for the 20 most used words.

[a] https://www.gutenberg.org/files/76/76-0.txt

Actions

Actions are operations that return non-RDD objects. Two of the most common actions, `take(n)` and `collect()`, have already been seen above. The key difference between the two is that `take(n)` returns the first `n` elements from one (or more) partition(s) while `collect()` returns the contents of

the entire RDD. When working with small datasets this may not be an issue, but for larger datasets running `collect()` can be very expensive.

Another important action is `reduce(func)`. Generally, `reduce()` combines (reduces) the data in each row of the RDD using `func` to produce some useful output. Note that `func` *must* be an associative and commutative binary operation; otherwise the results will vary depending on partitioning.

```
# create an RDD with the first million integers in 4 partitions
>>> ints = spark.sparkContext.parallelize(range(1, 1000001), 4)
# [1, 2, 3, 4, 5, ..., 1000000]
# sum the first one million integers
>>> ints.reduce(lambda x, y: x + y)
500000500000

# create a new RDD containing only survival data
>>> survived = titanic.map(lambda row: int(row[0]))
>>> survived.take(5)
[0, 1, 1, 1, 0]

# find total number of survivors
>>> survived.reduce(lambda x, y: x + y)
342
```

Spark Command	Action
`take(n)`	returns the first `n` elements of an RDD
`collect()`	returns the entire contents of an RDD
`reduce(f)`	merges the values of an RDD using an associative and commutative operator `f`
`count()`	returns the number of elements in the RDD
`min(); max(); mean()`	returns the minimum, maximum, or mean of the RDD, respectively
`sum()`	adds the elements in the RDD and returns the result
`saveAsTextFile(path)`	saves the RDD as a collection of text files (one for each partition) in the directory specified
`foreach(f)`	immediately applies `f` to each element of the RDD; not to be confused with `map()`, `foreach()` is useful for saving data somewhere not natively supported by PySpark

Problem 2. Since the area of a circle of radius r is $A = \pi r^2$, one way to estimate π is to estimate the area of the unit circle. A Monte Carlo approach to this problem is to uniformly sample points in the square $[-1, 1] \times [-1, 1]$ and then count the percentage of points that land within the unit circle. The percentage of points within the circle approximates the percentage of the area occupied by the circle. Multiplying this percentage by 4 (the area of the square $[-1, 1] \times [-1, 1]$) gives an estimate for the area of the circle. [a]

Write a function that uses Monte Carlo methods to estimate the value of π. Your function should accept two keyword arguments: `n=10**5` and `parts=6`. Use `n*parts` sample points and

partition your RDD with `parts` partitions. Return your estimate.

[a]See Example 7.1.1 in the Volume 2 textbook

DataFrames

While RDDs offer granular control, they can be slower than their Scala and Java counterparts when implemented in Python. The solution to this was the creation of a new data structure: Spark DataFrames. Just like RDDs, DataFrames are immutable distributed collections of objects; however, unlike RDDs, DataFrames are organized into named (and typed) columns. In this way they are conceptually similar to a relational database (or a pandas DataFrame).

The most important difference between a relational database and Spark DataFrames is in the execution of transformations and actions. When working with DataFrames, Spark's Catalyst Optimizer creates and optimizes a logical execution plan before sending any instructions to the drivers. After the logical plan has been formed, an optimal physical plan is created and executed. This provides significant performance boosts, especially when working with massive amounts of data. Since the Catalyst Optimizer functions the same across all language APIs, DataFrames bring performance parity to all of Spark's APIs.

Spark SQL and DataFrames

Creating a DataFrame from an existing text, csv, or JSON file is generally easier than creating an RDD. The DataFrame API also has arguments to deal with file headers or to automatically infer the schema.

```
# load the titanic dataset using default settings
>>> titanic = spark.read.csv('titanic.csv')
>>> titanic.show(2)
+---+---+--------------------+------+---+---+---+-------+
|_c0|_c1|                 _c2|   _c3|_c4|_c5|_c6|    _c7|
+---+---+--------------------+------+---+---+---+-------+
|  0|  3|Mr. Owen Harris B...|  male| 22|  1|  0|   7.25|
|  1|  1|Mrs. John Bradley...|female| 38|  1|  0|71.2833|
+---+---+--------------------+------+---+---+---+-------+
only showing top 2 rows

# spark.read.csv('titanic.csv', inferSchema=True) will try to infer
# data types for each column

# load the titanic dataset specifying the schema
>>> schema = ('survived INT, pclass INT, name STRING, sex STRING, '
...           'age FLOAT, sibsp INT, parch INT, fare FLOAT'
...           )
>>> titanic = spark.read.csv('titanic.csv', schema=schema)
>>> titanic.show(2)
+--------+------+--------------------+------+---+-----+-----+-------+
|survived|pclass|                name|   sex|age|sibsp|parch|   fare|
+--------+------+--------------------+------+---+-----+-----+-------+
```

```
|       0|     3|Mr. Owen Harris B...|  male| 22|    1|    0|   7.25|
|       1|     1|Mrs. John Bradley...|female| 38|    1|    0|71.2833|
+--------+------+--------------------+------+---+-----+-----+-------+
only showing top 2 rows

# for files with headers, the following is convenient
spark.read.csv('my_file.csv', header=True, inferSchema=True)
```

> **Note**
>
> To convert a DataFrame to an RDD use `my_df.rdd`; to convert an RDD to a DataFrame use `spark.createDataFrame(my_rdd)`. You can also use `spark.createDataFrame()` on numpy arrays and pandas DataFrames.

DataFrames can be easily updated, queried, and analyzed using SQL operations. Spark allows you to run queries directly on DataFrames similar to how you perform transformations on RDDs. Additionally, the `pyspark.sql.functions` module contains many additional functions to further analysis. Below are many examples of basic DataFrame operations; further examples involving the `pyspark.sql.functions` module can be found in the additional materials section. Full documentation can be found at https://spark.apache.org/docs/latest/api/python/pyspark.sql.html.

```
# select data from the survived column
>>> titanic.select(titanic.survived).show(3)  # or titanic.select("survived")
+--------+
|survived|
+--------+
|       0|
|       1|
|       1|
+--------+
only showing top 3 rows

# find all distinct ages of passengers (great for data exploration)
>>> titanic.select("age")\
...         .distinct()\
...         .show(3)
+----+
| age|
+----+
|18.0|
|64.0|
|0.42|
+----+
only showing top 3 rows

# filter the DataFrame for passengers between 20-30 years old (inclusive)
>>> titanic.filter(titanic.age.between(20, 30)).show(3)
```

```
+--------+------+--------------------+------+----+-----+-----+------+
|survived|pclass|                name|   sex| age|sibsp|parch|  fare|
+--------+------+--------------------+------+----+-----+-----+------+
|       0|     3|Mr. Owen Harris B...|  male|22.0|    1|    0|  7.25|
|       1|     3|Miss. Laina Heikk...|female|26.0|    0|    0| 7.925|
|       0|     3|     Mr. James Moran|  male|27.0|    0|    0|8.4583|
+--------+------+--------------------+------+----+-----+-----+------+
only showing top 3 rows

# find total fare by pclass (or use .avg('fare') for an average)
>>> titanic.groupBy('pclass')\
...         .sum('fare')\
...         .show()
+------+---------+
|pclass|sum(fare)|
+------+---------+
|     1| 18177.41|
|     3|  6675.65|
|     2|  3801.84|
+------+---------+

# group and count by age and survival; order age/survival descending
>>> titanic.groupBy("age", "survived").count()\
...         .sort("age", "survived", ascending=False)\
...         .show(2)
+---+--------+-----+
|age|survived|count|
+---+--------+-----+
| 80|       1|    1|
| 74|       0|    1|
+---+--------+-----+
only showing top 2 rows

# join two DataFrames on a specified column (or list of columns)
>>> titanic_cabins.show(3)
+--------------------+-------+
|                name|  cabin|
+--------------------+-------+
|Miss. Elisabeth W...|     B5|
|Master. Hudsen Tr...|C22 C26|
|Miss. Helen Lorai...|C22 C26|
+--------------------+-------+
only showing top 3 rows

>>> titanic.join(titanic_cabins, on='name').show(3)
+--------------------+--------+------+------+----+-----+-----+------+-------+
|                name|survived|pclass|   sex| age|sibsp|parch|  fare|  cabin|
+--------------------+--------+------+------+----+-----+-----+------+-------+
|Miss. Elisabeth W...|       0|     3|  male|22.0|    1|    0|  7.25|     B5|
```

```
|Master. Hudsen Tr...|       1|     3|female|26.0|    0|    0| 7.925|C22 C26|
|Miss. Helen Lorai...|       0|     3|  male|27.0|    0|    0|8.4583|C22 C26|
+--------------------+--------+------+------+----+-----+-----+------+-------+
only showing top 3 rows
```

Note

If you prefer to use traditional SQL syntax you can use `spark.sql("SQL QUERY")`. Note that this requires you to first create a temporary view of the DataFrame.

```
# create the temporary view so we can access the table through SQL
>>> titanic.createOrReplaceTempView("titanic")

# query using SQL syntax
>>> spark.sql("SELECT age, COUNT(*) AS count\
...             FROM titanic\
...             GROUP BY age\
...             ORDER BY age DESC").show(3)
+---+-----+
|age|count|
+---+-----+
| 80|    1|
| 74|    1|
| 71|    2|
+---+-----+
only showing top 3 rows
```

Spark SQL Command	SQLite Command
`select(*cols)`	`SELECT`
`groupBy(*cols)`	`GROUP BY`
`sort(*cols, **kwargs)`	`ORDER BY`
`filter(condition)`	`WHERE`
`when(condition, value)`	`WHEN`
`between(lowerBound, upperBound)`	`BETWEEN`
`drop(*cols)`	`DROP`
`join(other, on=None, how=None)`	`JOIN` (join type specified by `how`)
`count()`	`COUNT()`
`sum(*cols)`	`SUM()`
`avg(*cols)` or `mean(*cols)`	`AVG()`
`collect()`	`fetchall()`

Problem 3. In this problem, we will be exploring the Titanic dataset a bit further. Begin by spending 10-15 minutes exploring the data in an interactive environment (IPython, jupyter, etc.). If you run out of ideas, there are examples with further functions in the Additional Materials section.

Now that you've become familiar with DataFrames and the Titanic dataset, write a function with keyword argument `filename='titanic.csv'`. Load the file into a PySpark DataFrame and find (1) the number of women on-board, (2) the number of men on-board, (3) the survival rate of women, and (4) the survival rate of men. Return these four values in the order given.

Problem 4. In this problem, you will be using the `london_income_by_borough.csv` and the `london_crime_by_lsoa.csv` files to visualize the relationship between income and the frequency of crime.[a] The former contains estimated mean and median income data for each London borough, averaged over 2008-2016; the first line of the file is a header with columns `borough`, `mean-08-16`, and `median-08-16`. The latter contains over 13 million lines of crime data, organized by borough and LSOA (Lower Super Output Area) code, for London between 2008 and 2016; the first line of the file is a header, containing the following seven columns:

`lsoa_code`: LSOA code (think area code) where the crime was committed
`borough`: London borough were the crime was committed
`major_category`: major (read: general) category of the crime
`minor_category`: minor (read: specific) category of the crime
`value`: number of occurrences of this crime in the given `lsoa_code`, `month`, and `year`
`year`: year the crime was committed
`month`: month the crime was committed

Write a function that accepts three keyword arguments: `f1='london_crime_by_lsoa.csv'`, `f2='london_income_by_borough.csv'`, and `min_cat='Murder'`. Load the two files as PySpark DataFrames. Use them to create a new DataFrame containing, for each borough, the average crime rate for the given `min_cat` and the median income. Order the DataFrame by the average crime rate, descending. The final DataFrame should have three columns: `borough`, `average_crime_rate`, `median-08-16` (column names may be different).

Use the following function to generate a plot. Return `data`, formatted as expected by the function below.

```
def plot_crime(data, min_cat):
    """
    Helper function that plots the crime and income data from problem 4.

    Parameters:
        data ((n, 3) ndarray, dtype=str (or '<U22')); format should be:
            First Column:   borough names
            Second Column:  crime rate (i.e. avg crimes per month)
            Third Column:   median income
        min_cat (str): minor category of crime
```

```
    Returns: None
    """
    domain = np.arange(len(data))
    fig, ax = plt.subplots()

    # plot number of months with
    p1 = ax.plot(domain, data[:, 1].astype(float) , color='red',
                 label='Months w/ 1+ {}'.format(min_cat))

    # create and plot on second axis
    ax2 = ax.twinx()
    p2 = ax2.plot(domain, data[:, 2].astype(float), color='green',
                  label='Median Income')

    # create legend
    plots = p1 + p2
    labels = [line.get_label() for line in plots]
    ax.legend(plots, labels, loc=0)

    plt.title('Months w/ 1+ {} with Median Income'.format(min_cat))

    plt.show()
```

[a] data.london.gov.uk

Machine Learning with Apache Spark

Apache Spark includes a vast and expanding ecosystem to perform machine learning. PySpark's primary machine learning API, `pyspark.ml`, is DataFrame-based. There is an RDD-based machine learning API, `pyspark.mllib`, but it is much slower. For this reason, `pyspark.mllib` is in maintenance mode and will become deprecated as soon as `pyspark.ml` reaches feature parity. The RDD-based API is expected to be removed entirely in Spark 3.0, so use `pyspark.ml` whenever possible.

Here we give a start to finish example using Spark ML to tackle the classic Titanic classification problem.

```
# prepare data
# convert the 'sex' column to binary categorical variable
>>> from pyspark.ml.feature import StringIndexer, OneHotEncoderEstimator
>>> sex_binary = StringIndexer(inputCol='sex', outputCol='sex_binary')

# one-hot-encode pclass (Spark automatically drops a column)
>>> onehot = OneHotEncoderEstimator(inputCols=['pclass'],
...                                 outputCols=['pclass_onehot'])

# create single features column
```

```
from pyspark.ml.feature import VectorAssembler
features = ['sex_binary', 'pclass_onehot', 'age', 'sibsp', 'parch', 'fare']
features_col = VectorAssembler(inputCols=features, outputCol='features')

# now we create a transformation pipeline to apply the operations above
# this is very similar to the pipeline ecosystem in sklearn
>>> from pyspark.ml import Pipeline
>>> pipeline = Pipeline(stages=[sex_binary, onehot, features_col])
>>> titanic = pipeline.fit(titanic).transform(titanic)

# drop unnecessary columns for cleaner display (note the new columns)
>>> titanic = titanic.drop('pclass', 'name', 'sex')
>>> titanic.show(2)
+--------+----+-----+-----+----+----------+-------------+----------+
|survived| age|sibsp|parch|fare|sex_binary|pclass_onehot|  features|
+--------+----+-----+-----+----+----------+-------------+----------+
|       0|22.0|    1|    0|7.25|       0.0|    (3,[],[])|(8,[4,5...|
|       1|38.0|    1|    0|71.3|       1.0|   (3,[1],...|[0.0,1....|
+--------+----+-----+-----+----+----------+-------------+----------+

# split into train/test sets (75/25)
>>> train, test = feature_vector.randomSplit([0.75, 0.25], seed=11)

# initialize logistic regression
>>> from pyspark.ml.classification import LogisticRegression
>>> lr = LogisticRegression(labelCol='survived', featuresCol='features')

# run a train-validation-split to fit best elastic net param
# ParamGridBuilder constructs a grid of parameters to search over.
>>> from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator as MCE
>>> paramGrid = ParamGridBuilder()\
...                 .addGrid(lr.elasticNetParam, [0, 0.5, 1]).build()
# TrainValidationSplit will try all combinations and determine best model using
# the evaluator (see also CrossValidator)
>>> tvs = TrainValidationSplit(estimator=lr,
...                            estimatorParamMaps=paramGrid,
...                            evaluator=MCE(labelCol='survived'),
...                            trainRatio=0.75,
...                            seed=11)

# we train the classifier by fitting our tvs object to the training data
>>> clf = tvs.fit(train)

# use the best fit model to evaluate the test data
>>> results = clf.bestModel.evaluate(test)
>>> results.predictions.select(['survived', 'prediction']).show(5)
+--------+----------+
|survived|prediction|
```

```
+--------+----------+
|       0|       1.0|
|       0|       1.0|
|       0|       1.0|
|       0|       1.0|
|       0|       0.0|
+--------+----------+

# performance information is stored in various attributes of "results"
>>> results.accuracy
0.8044776119402985

>>> results.weightedRecall
0.8044776119402985

>>> results.weightedPrecision
0.8031447376345529

# many classifiers do not have this object-oriented interface (yet)
# it isn't much more effort to generate the same statistics for a ↩
    DecisionTreeClassifier, for example
>>> dt_clf = dt_tvs.fit(train) # same process, except for a different paramGrid

# generate predictions - this returns a new DataFrame
>>> preds = clf.bestModel.transform(test)
>>> preds.select('survived', 'probability', 'prediction').show(5)
+--------+-----------+----------+
|survived|probability|prediction|
+--------+-----------+----------+
|       0|  [1.0,0.0]|       0.0|
|       0|  [1.0,0.0]|       0.0|
|       0|  [1.0,0.0]|       0.0|
|       0|  [0.0,1.0]|       1.0|
+--------+-----------+----------+

# initialize evaluator object
>>> dt_eval = MCE(labelCol='survived')
>>> dt_eval.evaluate(preds, {dt_eval.metricName: 'accuracy'})
0.8433179723502304
```

Below is a broad overview of the `pyspark.ml` ecosystem. It should help give you a starting point when looking for a specific functionality.

PySpark ML Module	Module Purpose
`pyspark.ml.feature`	provides functions to transform data into feature vectors
`pyspark.ml.tuning`	grid search, cross validation, and train/validation split functions
`pyspark.ml.evaluation`	tools to compute prediction metrics (accuracy, f1, etc.)
`pyspark.ml.classification`	classification models (logistic regression, SVM, etc.)
`pyspark.ml.clustering`	clustering models (k-means, Gaussian mixture, etc.)
`pyspark.ml.regression`	regression models (linear regression, decision tree regressor, etc.)

ACHTUNG!

At the time of writing, Spark hosts both RDD- and DataFrame-based APIs for machine learning. The RDD API, MLlib, is in maintenance mode and will be deprecated as soon as the DataFrame API attains feature parity. MLlib is expected to be removed entirely in Spark 3.0.

Problem 5. Write a function with keyword argument `filename='titanic.csv'`. Load the file into a PySpark DataFrame, and use the `pyspark.ml` package to train a classifier. Your goal is to outperform the logistic regression each of the three metrics from the example above (`accuracy`, `weightedRecall`, `weightedPrecision`).

Some of Spark's available classifiers are listed below. For complete documentation, visit https://spark.apache.org/docs/latest/api/python/pyspark.ml.html.

```
# from pyspark.ml.classification import LinearSVC
#                              DecisionTreeClassifier
#                              GBTClassifier
#                              MultilayerPerceptronClassifier
#                              NaiveBayes
#                              RandomForestClassifier
```

Use `randomSplit([0.75, 0.25], seed=11)` to split your data into train and test sets before fitting the model. Return the `accuracy`, `weightedRecall`, and `weightedPrecision` for your model, in the given order.

Additional Material

Further DataFrame Operations

There are a few other functions built directly on top of DataFrames to further analysis. Additionally, the `pyspark.sql.functions` module expands the available functions significantly.[3]

```
# some immediately accessible functions
# covariance between pclass and fare
>>> titanic.cov('pclass', 'fare')
-22.86289824115662

# summary of statistics for selected columns
>>> titanic.select("pclass", "age", "fare")\
...         .summary().show()
+-------+------------------+------------------+------------------+
|summary|            pclass|               age|              fare|
+-------+------------------+------------------+------------------+
|  count|               887|               887|               887|
|   mean| 2.305524239007892|29.471443066501564|32.305420253026846|
| stddev|0.8366620036697728|14.121908405492908| 49.78204096767521|
|    min|                 1|              0.42|               0.0|
|    25%|                 2|              20.0|             7.925|
|    50%|                 3|              28.0|           14.4542|
|    75%|                 3|              38.0|            31.275|
|    max|                 3|              80.0|          512.3292|
+-------+------------------+------------------+------------------+

# additional functions from the functions module
>>> from pyspark.sql import functions as sqlf

# finding the mean of a column without grouping requires sqlf.avg()
# alias(new_name) allows us to rename the column on the fly
>>> titanic.select(sqlf.avg("age").alias("Average Age")).show()
+------------------+
|       Average Age|
+------------------+
|29.471443066516347|
+------------------+

# use .agg([dict]) on GroupedData to specify [multiple] aggregate
# functions, including those from pyspark.sql.functions
>>> titanic.groupBy('pclass')\
...         .agg({'fare': 'var_samp', 'age': 'stddev'})\
...         .show(3)
+------+------------------+------------------+
|pclass|    var_samp(fare)|       stddev(age)|
+------+------------------+------------------+
```

[3] https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

```
|     1| 6143.483042924841|14.183632587264817|
|     3|139.64879027298073|12.095083834183779|
|     2|180.02658999396826|13.756191206499766|
+------+------------------+------------------+

# perform multiple aggregate actions on the same column
>>> titanic.groupBy('pclass')\
...         .agg(sqlf.sum('fare'), sqlf.stddev('fare'))\
...         .show()
+------+------------------+------------------+
|pclass|         sum(fare)| stddev_samp(fare)|
+------+------------------+------------------+
|     1|18177.412506103516| 78.38037409278448|
|     3| 6675.653553009033| 11.81730892686574|
|     2|3801.8417053222656|13.417398778972332|
+------+------------------+------------------+
```

pyspark.sql.functions	**Operation**
`ceil(col)`	computes the ceiling of each element in `col`
`floor(col)`	computes the floor of each element in `col`
`min(col)`, `max(col)`	returns the minimum/maximum value of `col`
`mean(col)`	returns the average of the values of `col`
`stddev(col)`	returns the unbiased sample standard deviation of `col`
`var_samp(col)`	returns the unbiased variance of the values in `col`
`rand(seed=None)`	generates a random column with i.i.d. samples from $[0, 1]$
`randn(seed=None)`	generates a random column with i.i.d. samples from the standard normal distribution
`exp(col)`	computes the exponential of `col`
`log(arg1, arg2=None)`	returns `arg1`-based logarithm of `arg2`; if there is only one argument, then it returns the natural logarithm
`cos(col)`, `sin(col)`, etc.	computes the given trigonometric or inverse trigonometric (`asin(col)`, etc.) function of `col`