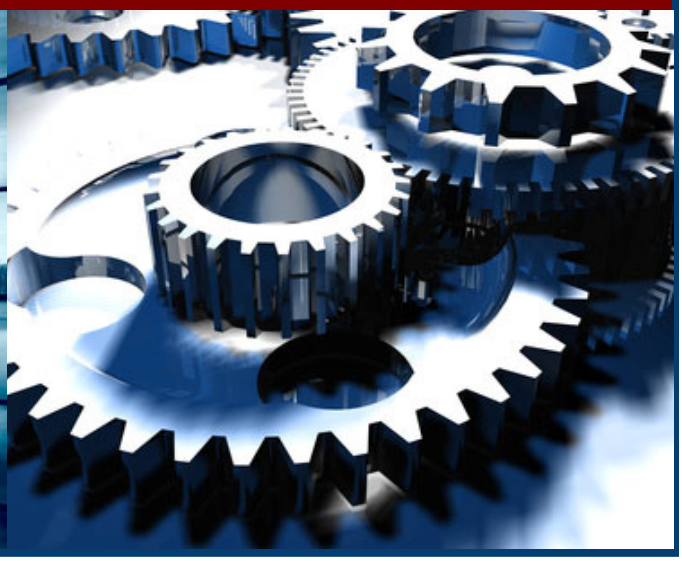
Industry Leader in Customized IT & Management Training



Course Notes

Course 455DE:
Spark for Data Engineers

ROI Training
Maximize Your Training Investment
www.roitraining.com
US: 1-800-ROI-9877 | Outside US: +44 (0)20 3287 6485

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

INTRODUCTION

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

The following course materials are copyright protected materials. They may not be reproduced or distributed and may only be used by students attending the *Spark for Data Engineers* course.

Welcome!

- ROI leads the industry in designing and delivering customized technology and management training solutions
- Meet your instructor
  - Name
  - Background
  - Contact info
- Let's get started!

Course Objectives

In this course, we will:

- Explore the Spark platform for using the power of clusters to solve complex queries
- Start with RDDs and see how they evolved into DataFrames
- Use DataFrames and SparkSQL to run queries on virtually any kind of data
- Take Datasets and see how they can be transformed for running machine learning algorithms
- Discover the basic machine learning models of Cluster, Regression, and Classification

Course Contents

Chapter 0	Introduction
Chapter 1	Introduction to Spark
Chapter 2	DataFrames
Chapter 3	Spark SQL
Chapter 4	Cluster Analysis
Chapter 5	Regression Analysis
Chapter 6	Classification Models
Chapter 7	Course Summary

Class Schedule

- Start of class _____
- Morning breaks approximately on the hour
- Lunch _____
- Afternoon breaks approximately on the hour
- Class end _____

ROI's Training Curricula

Agile Development	.NET and Visual Studio
Amazon Web Services (AWS)	Networking and IPv6
Azure	Oracle and SQL Server Databases
Big Data and Data Analytics	OpenStack and Docker
Business Analysis	Project Management
Cloud Computing and Virtualization	Python and Perl Programming
Excel and VBA	Security
Google Cloud Platform (GCP)	SharePoint
ITIL® and IT Service Management	Software Analysis and Design
Java	Software Engineering
Leadership and Management Skills	UNIX and Linux
Machine Learning and Neural Networks	Web and Mobile Apps
Microsoft Exchange	Windows and Windows Server

Please visit our website at www.ROItraining.com for a complete list of offerings

Student Introductions

Please introduce yourself stating:

- Name
- Position or role
- Expectations or a question you'd like answered during this class

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

CHAPTER 1: INTRODUCTION TO SPARK

Chapter Objectives

In this chapter, we will :

- Review the basics of Hadoop
- Review the history of Apache Spark
- Look at the architecture and components of Apache Spark
- Load files into RDD
- Process RDD using actions and transformation

Start Jupyter

- To start Jupyter with the latest lesson on the VM:
  - Open a terminal window and type the following commands:

```
sudo bash
start-notebook
```

  - This will launch the browser so you can navigate to which lesson folder you want to work on

About HDFS—I

- The Hadoop Distributed File System (HDFS) is the main storage used by Hadoop MapReduce applications
  - Distributed, POSIX-like file system
    - Designed to run on commodity hardware
    - Scales to clusters composed of thousands of nodes
  - Highly fault tolerant
    - Automatically detects hardware faults
    - Supports quick recovery
  - Implemented in Java
- Can be used as a standalone general purpose file system, but relaxes certain POSIX file system requirements
  - Designed for storing and reading very large files (>TB)
    - Supports high throughput read and writes
    - Write once, read many
    - Aimed at batch processing
    - Default block size is 128MB
  - Does not support random insertion or modification of data
  - Appending/truncating data is possible

About HDFS—II

- HDFS is used either directly or indirectly by many Big Data and NoSQL applications including:
  - Hadoop
  - Spark
  - HBase
  - Pig
  - Hive
  - Others

Core HDFS Services

- HDFS is implemented as several services which are usually deployed on a cluster of machines
  - Referred to as an HDFS cluster
  - Arranged in a controller/worker architecture
- Core HDFS services include:
  - **NameNode** (controller) stores file system metadata
  - **DataNode** (worker) stores file data (data blocks)
- The NameNode is the master server
  - Implements a POSIX-like hierarchical file system with '/' as the root directory
  - Enforces read/write permissions on files and directories
  - Tracks the location of the data blocks for each file
- The DataNode is the slave server
  - Handles read and write requests from HDFS clients
  - Performs block creation, deletion, and replication as instructed by the NameNode

Start Hadoop

- For this class, we need symlink to point to the class folder
  - `ln -s /home/student/ROI/SparkforDataEngineers ~/ROI/Spark`
- To start Hadoop on the VM:
  - Open a terminal window and type the following commands:

```
sudo bash
start-hadoop
jps
exit
```

- From a command line, enter the following commands:

```
hdfs
hdfs dfs
hdfs dfs -ls /
hdfs dfs -put ~/ROI/Spark/datasets/northwind/CSV/categories /
hdfs dfs -ls /
```

Command Line Examples

- As we have seen, HDFS provides a command line interface
- From a command line, enter the following commands:

```
hdfs
hdfs dfs
hdfs dfs -ls /
hdfs dfs -put ~/ROI/datasets/northwind/CSV/categories /
hdfs dfs -ls /
```

Introduction to Apache Spark

- Apache Spark is a computing engine that can be used for large-scale data processing
  - Spark 2 can perform between 100X and 1000X faster than Hadoop's default computing engine (MapReduce)
  - Created by Matei Zaharia at UC Berkley in 2009
  - Donated to Apache Software Foundation in 2013
  - Apache Spark has seen immense growth over the past several years
- Spark functionality includes the ability to:
  - Perform iterative processing
  - Work with structured data via SQL
  - Support Hive Query Language (HQL)
  - Interact with it via a command-line shell
  - Support near real-time processing using in-memory data structure
- See https://spark.apache.org/

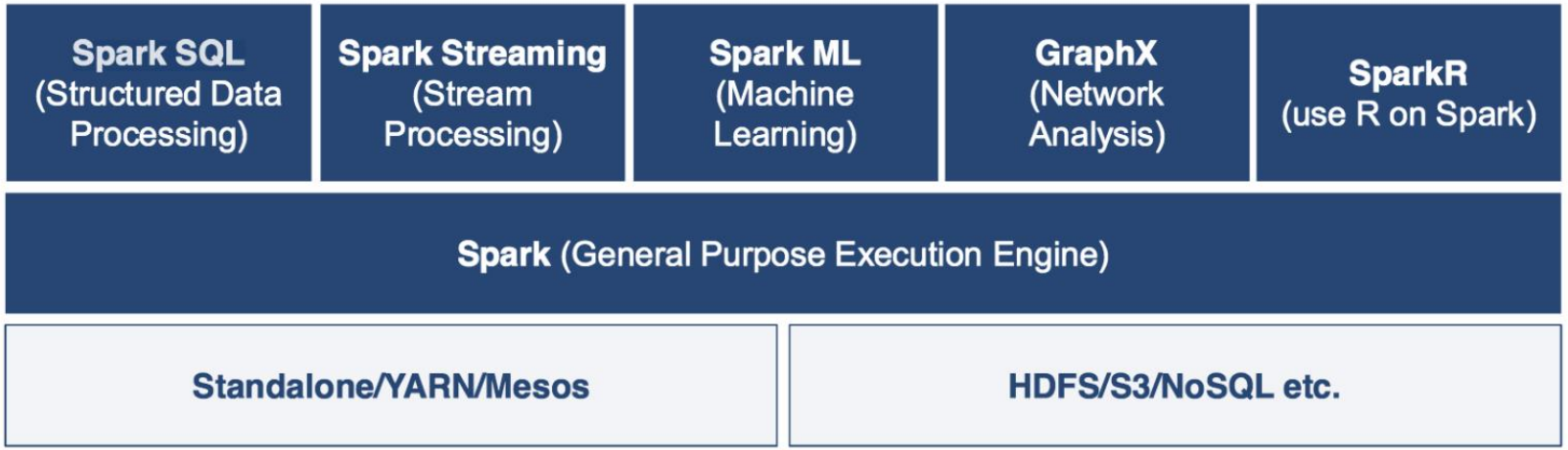
Introduction to Apache Spark (continued)

- Apache Spark provides high-level APIs in Java, Scala, Python, and R and has an optimized engine that supports general execution graphs
- Two important use cases for Apache Spark are data processing and AI
- Spark unifies data processing and AI by providing a powerful in-memory execution engine
- It also offers popular AI frameworks and libraries such as TensorFlow, R, and SciKit-Learn

Speed

- Run computations in memory
- Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing
- 100 times faster in-memory and 10 times faster even when running on a disk than MapReduce

Spark Components



Spark Core

- Spark Core is the underlying general execution engine for the Spark platform, all other functionality is built on top of it
- Provides distributed task dispatching, scheduling, and basic IO functionalities exposed through an application programming interface centered on the RDD, which is Spark's primary programming abstraction

Spark SQL

- Spark package designed for working with structured data which is built on top of Spark Core
- Provides a SQL-like interface for working with structured data
- More and more Spark workflow is moving towards Spark SQL

Spark Streaming

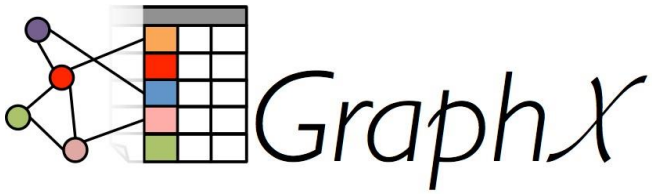
- Running on top of Spark, Spark Streaming provides an API for manipulating data streams that closely match the Spark Core's RDD API
- Enables powerful interactive and analytical applications across both streaming and historical data while inheriting Spark's ease of use and fault tolerance characteristics

Spark MLlib

- Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms and blazing speed
- Usable in Java, Scala, and Python as part of Spark applications
- Consists of common learning algorithms and utilities including classification, regression, clustering, collaborative filtering, and dimensionality reduction, etc.

GraphX

- A graph computation engine built on top of Spark that enables users to interactively create, transform, and reason about graph-structured data at scale
- Extends the Spark RDD by introducing a new graph abstraction
  - A directed multigraph with properties attached to each vertex and edge



Spark Application Top-Down View

- Spark executes *applications*
  - Either in local, stand-alone mode, or as a cluster
- Spark applications have one *driver* and one or more *executors*
  - Drivers and executors run as Java processes
  - Drivers assign *tasks* to the executors
  - Executors run tasks on *Resilient Distributed Datasets* (RDDs)
  - Executors send results to the driver
- Drivers include:
  - Spark Shell
    - PySpark—the Python shell
    - Spark Shell—the Scala shell
  - Custom program
    - Written in Python, Java, or Scala

Resilient Distributed Datasets (RDDs)

- RDDs represent the core data construct of Spark
  - RDDs are immutable
  - RDDs are fault tolerant (resilient)
    - When nodes or tasks fail, RDDs are reconstructed on other nodes
  - RDDs are split into *partitions* and can be distributed to any executor
  - RDDs can contain any kind of data
    - Prefer data that can be partitioned
- RDDs are objects that support two categories of operations
  - Transformations
    - Create new RDDs from existing RDDs
    - Always return RDDs
    - Lazily evaluated
  - Actions
    - Start computations
    - Return results to the driver
    - Save results to disk
    - Never return RDDs

Spark Application Flow

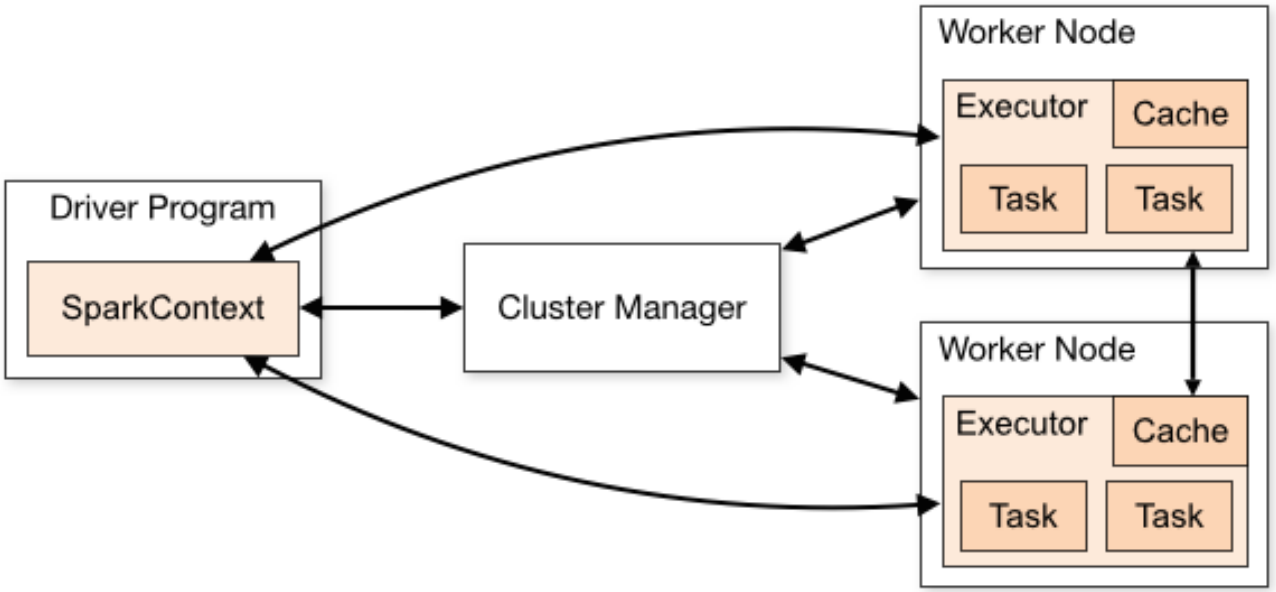
- Spark applications tend to have a similar flow
  1. Create a Spark context
     a. Automatically provided in the shells via the variable `sc`
  2. Import data as RDDs
  3. Transform and perform actions on RDDs
  4. Export results
- Spark applications are not declarative in nature, however, they are still coded at a high level of abstraction
  - In contrast, $tasks$ are at a very low level of abstraction
  - Tasks are not created by programmers, rather they are created at runtime by Spark

Drivers and Executors

- Spark applications consist of a driver process and a set of executor processes
- The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things:
  - Maintaining information about the Spark application
  - Responding to a user's program or input
  - Analyzing, distributing, and scheduling work across the executors (defined momentarily)
- The driver process is absolutely essential—it's the heart of a Spark application and maintains all relevant information during the lifetime of the application
- The executors are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things:
  - Executing code assigned to it by the driver
  - Reporting the state of the computation, on that executor, back to the driver node

Spark's Basic Architecture

- The cluster manager controls physical machines and allocates resources to Spark applications
- This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos
- This means that there can be multiple Spark applications running on a cluster at the same time



Start PySpark

- To start PySpark on the VM:
  - Open a terminal window and type the following commands:

```
cd ~/ROI
pyspark
sc
spark
x =
sc.textFile('/home/student/ROI/Spark/datasets/text/shakespeare.txt')
x.count()
x.take(10)
```

- To write a Python program from scratch, you have to initialize `sc` and `spark` manually
  - `initspark.py` is a helper module you can copy and use in your own scripts

```
from initspark import *
sc, spark, conf = initspark()
sc, spark, conf = initspark(appname = 'appname', servername =
'sparkservername', cassandra = '127.0.0.1')
```

Load Data

- The `sc` object is the Spark context and allows you to call methods to load and manipulate data

```
x = sc.parallelize(range(1, 11))
x.collect()
x.take(5)
sc.textFile('hdfs://localhost:9000/categories').collect()
```

- Load a local file

```
x = sc.textFile('file:///home/student/ROI/Spark/datasets/
    northwind/CSV/categories/categories.csv')
```

- Load a local folder

```
x = sc.textFile('file:///home/student/ROI/Spark/datasets/
    northwind/CSV/categories')
```

- Load a hdfs folder

```
x = sc.textFile('hdfs://localhost:9000/categories')
```

Actions and Transformations

- Once you have an RDD, you can invoke methods on it
- Methods can either be:
  - An action which causes it to do some work and possibly return data back to the client
  - A transformation which is lazy evaluated and is only run when an action is called
- Transformations can be chained together to create multiple operations on the data, but none are executed until an action is called. This allows the entire chain of transformations to be internally optimized by Spark before execution.
- Transformations can also be either:
  - Narrow: can operate on the data in a single node
    - Like a map operation in MapReduce
  - Wide: requires data with the same key to be shuffled around to the same nodes
    - Like a reduce operation in MapReduce

Processing Data

- The data is loaded into an RDD (Resilient Distributed DataFrame)
  - Very similar to a Python list except it is spread across many nodes in the cluster
  - Has many built-in methods to process the data
- Loading data from a text file basically creates a list of strings
- Some useful actions to look at the data are:
  - `rdd.collect()` – returns the entire RDD as a Python list to the client
  - `rdd.count()` – returns a count of how many items are in the RDD
  - `rdd.take(x)` – returns `x` number of items from the RDD as a list
  - `rdd.takeOrdered(x, key=function)` – returns `x` rows of an RDD after sorting it first using a function
  - `rdd.top(x, key=function)` – returns the opposite of `takeOrdered`
  - `rdd.takeSample(replacement, count, seed)` – returns a sample of a larger data set
  - `rdd.foreach(function)` – executes the function once for each element of the RDD

Saving Data

- There are a lot of methods to save data to different formats
  - `rdd.saveAsTextFile()` – saves the RDD as a plain text file
  - `rdd.saveAsHadoopFile()` – saves the RDD as a key/value pair file suitable for Hadoop
  - `rdd.saveAsSequenceFile()` – saves the RDD as a Hadoop sequence file
  - `rdd.saveAsPickleFile()` – saves the RDD as a Python pickle file

Transformations

- Transformations are used to create a recipe of changes you want to make to the data
  - String parsing, data conversion, calculations
  - Filtering
  - Matching
  - Sorting
  - Aggregating
- Some useful transformations:
  - Narrow transformations
    - `rdd.map()` – applies a function to each element of the RDD
    - `rdd.flatMap()` – applies a function and flattens the elements
    - `rdd.filter()` – applies a function to determine if an element is returned
  - Wide transformations
    - `rdd.sort()` – orders the RDD
    - `rdd.groupBy()` – accumulates items with a key into a tuple of the key and list of the items
    - `rdd.reduce()` – runs a function on items for a key to return an aggregated value
    - `rdd.join()` – matches elements in one RDD to another

Lambda

- Many actions and transformations take a function as a parameter to allow customization of how the method works
- You could pass it a function name if you have one defined, but in many cases the functions are trivial
- Python allows you to create a function on the fly that can be passed as a parameter without the need to create the function in advance
- If all you need to do is create a simple function that takes one or more parameters and return a calculation that can be done in a single statement, then a lambda is a good choice

```
def isEven(x):
    return x % 2
isEven = lambda x : x % 2

rdd.filter(isEven)
rdd.filter(lambda x: x % 2)
sc.parallelize(range(1, 11)).sortBy(lambda x : (x % 2, x))
```

Chapter Summary

In this chapter, we have:

- Reviewed the basics of Hadoop
- Reviewed the history of Apache Spark
- Looked at the architecture and components of Apache Spark
- Loaded files into RDD
- Processed RDD using actions and transformation

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

CHAPTER 2: DATAFRAMES

Chapter Objectives

In this chapter, we will:

- Introduce DataFrames
- Show how to create a structured object using DataFrames
- Apply transformations and actions on DataFrames

DataFrames

- Spark 2.0 introduced a more feature rich and easier to use version of RDDs known as a DataFrame
  - Modeled to be similar to Pandas DataFrame so it is easily familiar
  - Is an RDD but has column names and data types
  - Has transformations and actions that are easier to use than RDD versions
  - Attempts to be more SQL-like for even more familiarity
  - Can read and write many more file formats than basic RDDs could

Make a DataFrame

- Spark 2.0 introduced the `SparkSession`, simply called spark in PySpark
  - Provides easier access to the different spark contexts
  - `spark.sparkContext` is the same as the old `sc`
- Once we have the spark context, we can start using DataFrames

```
x = sc.parallelize([(1,'alpha'),(2,'beta')])
x0 = spark.createDataFrame(x)
x0.show()

+---+-----+
| _1|   _2|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+
```

Column Names

- To make the DataFrame more useful, column names can be applied

```
x1 = spark.createDataFrame(x, schema = ['ID', 'Name'])
x1.show()
x1.describe()

+---+-----+
| ID| Name|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+

DataFrame[summary: string, ID: string, Name: string]
```

Schemas

- To make the DataFrame even more useful, a schema with data types can be applied

```
x2 = spark.createDataFrame(x, 'ID:int, Name:string')
x2.show()
print(x2)
+---+-----+
| ID| Name|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+

DataFrame[ID: int, Name: string]
```

Convert RDD to DataFrame

- An existing RDD can also be turned into a DataFrame using the `toDF` method
  - Using the credit card csv file from before:

```
cc = sc.textFile ('/home/student/ROI/Spark/datasets/
     finance/CreditCard.csv')
first = cc.first()
cc = cc.filter(lambda x : x != first)
import datetime
cc = cc.map(lambda x : x.split(','))
cc = cc.map(lambda x : (x[0][1:], x[1][1:-1],
  datetime.datetime.strptime(x[2], '%d-%b-%y').date(),
  x[3], x[4], x[5], float(x[6])))
df = cc.toDF()
df.show()
df = cc.toDF(['City', 'Country', 'Date', 'CardType',
  'TranType', 'Gender', 'Amount'])
df.show()
```

Selecting Columns

- DataFrames have methods with names similar to SQL commands

```
df.select('City', 'Country', 'Amount').show(10)

df.select('City', 'Country').distinct().show()

df.sort(df.Amount).show()

df.sort(df.Amount, ascending = False).show()

df.select('City', 'Amount').orderBy(df.City).show()
```

Calculated Columns

- New columns can be added to a DataFrame

```
df2 = df.withColumn('Discount', df.Amount * .03)
df2.show()
```

- Columns can be removed when not needed

```
df3 = df2.drop(df2.Country)
df3.show()
```

Filtering Data

- DataFrames can be filtered like a SQL table using either the `filter` or `where` method
  - They are the exact same method with different aliases

```
df3.filter(df3.Amount < 4000).show()
df3.filter('Amount < 4000').count()
df3.where('Amount < 4000').count()
df3.where(df3.Amount < 4000).count()
df3.where((df3.Amount > 3000) & (df3.Amount <
4000)).count()
df3.where('Amount > 3000 and Amount < 3000').count()
```

Sorting

- The `sort` and `orderBy` methods are different aliases for the same function

```
df.sort(df.Amount).show()
```

- They sort a DataFrame in ascending order or descending order if you pass the `ascending = False` parameter

```
df.sort(df.Amount, ascending = False).show()
```

- You can sort on multiple columns

```
df.select('City', 'Amount').orderBy(df.City,
df.Amount).show()
```

- Custom sort functions can use the `withColumn` method

JOIN

- DataFrames can be joined to other DataFrames just as you would in SQL and all the expected types are supported
  - INNER
  - LEFT
  - RIGHT
  - FULL

```
tab1 = sc.parallelize([(1, 'Alpha'), (2, 'Beta'), (3,
'Delta')]).toDF('ID:int, code:string')

tab2 = sc.parallelize([(100, 'One', 1), (101, 'Two', 2),
(102, 'Three', 1), (103, 'Four', 4)]) \
.toDF('ID:int, name:string, parentID:int')

tab1.join(tab2, tab1.ID == tab2.parentID).show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'left').show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'right').show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'full').show()
```

Grouping and Aggregating

- Grouping in Spark works a little differently—the `groupBy` method creates a grouped DataFrame which can then have aggregate methods called on it

```
tab3 = sc.parallelize([(1, 10), (1, 20), (1, 30), (2,
40), (2,50)]).toDF('groupID:int, amount:int')
x = tab3.groupby('groupID')
```

- There are various different syntaxes to accomplish the same results
  - Call the method after grouping

```
x.max().show()
```

  - Use the `agg` method with a dictionary

```
x.agg({'amount':'sum', 'amount':'max'}).show()
```

  - Use the `agg` method with the function names

```
from pyspark.sql import functions as F
x.agg(F.sum('amount'), F.max('amount')).show()
```

Reading Files

- There are many file formats directly supported for reading and writing
  - `csv`
  - `json`
  - `orc`
  - `parquet`
  - `jdbc`
- Other formats can be loaded using custom Java classes
  - Cassandra
  - Mongo
  - HBase
  - AVRO

Reading CSV Files

- There are many different syntaxes that you will see but they all do the same thing
- The `sep` parameter can also be used to indicate different separators like `\t` for tab

```
filename = '/home/student/ROI/Spark/datasets/
           finance/CreditCard.csv'

df4 = spark.read.load(filename, format = 'csv',
  sep = ',', inferSchema = True, header = True)

df4 = spark.read.format('csv').option('header','true').
  option('inferSchema','true').load(filename)

df4 = spark.read.csv(filename, header = True,
  inferSchema = True)
```

Writing Files

- The write method on a DataFrame can be used just like the `read` function using many different options
- Some options are built in, such as:
  - `jdbc`
  - `json spark.read.json(filename)  df.write.json(file)`
  - `orc spark.read.orc(filename)  df.write.orc(file)`
  - `parquet spark.read.parquet(file)  df.write.parquet(file)`
  - `text spark.read.text(file)  df.write.text(file)`
- Other formats can use the option to supply a custom Java class that can be downloaded and installed on the computer
  - AVRO:

```
spark.read.format("com.databricks.spark.avro").load("kv.avro")
```

  - Cassandra:

```
sqlContext.read.format("org.apache.spark.sql.cassandra").options
(table = table_name, keyspace = keys_space_name).load()
```

Miscellaneous Useful Methods

- A lot of standard SQL is supported by Spark
- Using the `expr` function in combination with `withColumn`, you can add calculated columns to a DataFrame if you can code the calculation as standard SQL

```
from pyspark.sql.functions import expr
x2.withColumn('uppername', expr('upper(name)')).show()
```

- Sometimes you just want to easily rename a column
  - `withColumnRenamed(oldname, newname)`
- Like `collect()`, the `toLocalIterator()` method will return all the results to the driver node, but it does it as a generator instead of a `list`
- Just like SQL there are methods `union`, `unionAll`, `subtract`, and `intersect`

User Defined Functions

- Sometimes it is necessary to write complex functions using Python
- Import the helper functions in `pyspark.sql`

```
from pyspark.sql.functions import udf
from pyspark.sql.types import *
from pyspark.sql.functions import to_date
```

- Write whatever custom function you need:

```
def city(x):
    return x[:x.find(',')]
def country(x):
    return x[x.find(',') + 1 :]
```

- Call the built-in function or use the `udf` function to wrap and call your UDF:

```
df4.withColumn('City', udf(city, StringType())(df4.CityCountry)) \
.withColumn('Country', udf(country, StringType())(df4.CityCountry)) \
.withColumn('Date', to_date(df4.Date, 'dd-MMM-yy')) \
.drop(df4.CityCountry)
```

Chapter Summary

In this chapter, we have:

- Introduced DataFrames
- Shown how to create a structured object using DataFrames
- Applied transformations and actions on DataFrames

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

CHAPTER 3: SPARK SQL

Chapter Objectives

In this chapter, we will:

- Introduce Spark SQL
- Use SQL on DataFrames
- Discuss Hive integration

Spark SQL

- Spark 2.0 introduced not only DataFrames, but the ability to query them in two different ways
  - Using methods on the DataFrame variable
    - Syntax is programmatic, so more familiar to programmers not already well versed in SQL
    - Many more methods exist than the SQL language supports
  - Using SQL queries
    - Uses familiar syntax based on HQL (Hive Query Language)
    - Allows data analysts to leverage already existing queries and knowledge to get results quicker
- Can combine the two methods together to mix and match where appropriate for solving a query

Hive Integration

- Spark 2.0 can directly query a table in the Hive catalog if you add Hive support when making the Spark session object

```
spark=SparkSession.builder.enableHiveSupport().getOrCreate()
```

- Then using the SQL method, you can run any Hive query

```
regions = spark.sql('select * from regions')
regions.show()
```

- This will simply use the metadata definition in the Hive catalog and do all the processing in Spark
  - Hive is not doing the work, Spark is

Temporary View

- A Hive table is nothing more than a stored definition in the Hive catalog
- Even in Hive, the table is virtual and only exists during processing
- The `createDataFrame` method can be called on an existing DataFrame to temporarily put it into a local copy of the Hive catalog for the current Spark session and treat it as if it were a Hive table

```
x1 = spark.createDataFrame(x, schema = ['ID', 'Name'])
x1.createOrReplaceTempView('MyTable')
```

- Once created, this virtual table or view can be queried with Spark SQL using the HQL dialect of SQL

```
x2 = spark.sql('select * from MyTable')
x2.show()
```

Writing Results To Table

- Not only could you read from a Hive table, but you could also write the results of a query to a Hive table
- This has the effect of writing the output to HDFS and keeping a copy of the schema information in the Hive catalog
- Could use the `saveAsTable` method

```
x1.write.saveAsTable('Table1', mode = 'overwrite')
```

- Or using `CREATE TABLE AS` syntax of SQL

```
spark.sql('CREATE TABLE Table2 AS SELECT * FROM MyTable')
```

- `x1` and `x2` are DataFrames, `MyTable` is a temporary view, `Table1` and `Table2` become permanent entries in the Hive catalog

Methods vs. SQL

- Once you have a temporary view or Hive table, you can use a combination of SQL queries and method calls
- These two queries are equivalent, although SQL tends to be not case sensitive whereas Spark and Python are, so sometimes you need to watch out for case

```
sql = """select r.regionid, r.regionname,
t.territoryid, t.territoryname
from regions as r
join territories as t on r.regionid = t.regionid
order by r.regionid, t.territoryid"""
rt = spark.sql(sql)
```

```
tr = regions.join(territories, regions.regionid ==
territories.RegionID).select('regions.regionid',
'regionname', 'TerritoryID', 'TerritoryName')
```

Interacting with SQL Servers

- Data in any SQL server such as Oracle, MySQL, Sybase, or Microsoft SQL Server is stored in one central server and can be processed using that server's dialect of SQL
  - Does not run in a cluster or scale
  - SQL is used to store and process
- Spark can read and write data stored in a SQL Server into a Spark DataFrame and process it
  - Uses the cluster and scales
  - Only uses SQL as storage

Interacting with SQL Servers (continued)

- Spark can connect to any JDBC or ODBC data source
  - Write a DataFrame to a new SQL table

```
regions.write.format("jdbc").options(url="jdbc:mysql:/
/localhost/northwind", driver='com.mysql.jdbc.Driver',
dbtable='regions', user='test', password = "password",
mode = "append", useSSL = "false").save()
```

  - Read a SQL table into a DataFrame

```
regions2 = spark.read.format("jdbc").
options(url="jdbc:mysql://localhost/northwind",
driver="com.mysql.jdbc.Driver", dbtable= "regions",
user="test", password="password").load()
regions2.show()
```

Calculated Columns

- Creating the `regions2` DataFrame does not execute anything yet
- Making DataFrame into a Temp View, then running a Spark SQL query, tells Spark to read the SQL data into an RDD in the Spark Cluster
- Once in the cluster, it can be processed using Spark methods or Spark SQL

```
regions2.createOrReplaceTempView('regions2')
spark.sql('select * from regions2
           where regionid < 3').show()
```

Mixing Python and SQL

- HQL is predefined to have most of the standard functions in the SQL language
- Sometimes Python has a function that SQL does not
- You could always use Spark methods to call a Python function, as we showed in the last chapter

```
from pyspark.sql.functions import expr, udf
from pyspark.sql.types import *
t2 = spark.sql('select * from territories')
t2.printSchema()
t2 = t2.withColumn('upperName', expr('UPPER(TerritoryName)'))
t2.show(5)
t2 = t2.withColumn('titleName', udf(lambda x : x.title(),
StringType())(t2.upperName))
t2.show(5)
```

User Defined Functions

- To make it easier to use, you could take a Python function and turn it into a UDF
- This is similar to how you make a DataFrame into a Temp View
- Registering the Python function makes it a UDF callable within Spark SQL queries

```
def reverseString(x):
    return x[::-1]

spark.udf.register('reverse', reverseString, StringType())

spark.sql('select *, reverse(TerritoryName) as Reversed
from Territories').orderBy('Reversed').show()
```

Complex Data Types

- In addition to the typical primitive SQL datatypes, Spark has a few complex types that are similar to Python collection types and Hive's complex datatypes
  - `ArrayType` is similar to a list
  - `MapType` is similar to a dictionary
  - `StructType` and `StructField` allow you to build a table structured object
- Can be used to create more complex shaped objects than is possible in traditional SQL relational model
- Using the `ArrayType`, multiple children values can be embedded into a column instead of using a related table
- You could create a dynamic on the fly collection of Key Value pairs using `MapType`
- Using `StructType` and `StructField`, you could create a table structure that could embed an entire sub-table into a field

Use Case

- Typically `Join`, `Group`, and `Aggregate` operations tend to be more expensive because they are wide transformations that require a lot of shuffling of data around the nodes
- An alternative data modeling technique would be to embed children records inside the parent so that in a sense they are pre-joined to the parent
- This reduces the need to do joins and to shuffle data when doing aggregates

collect_list

- Spark has two aggregate functions called `collect_list` and `collect_set` that can be used to group together elements into either a list or a unique set

```
from pyspark.sql.functions import collect_list
territories.groupBy(territories.RegionID). \
agg(collect_list(territories.TerritoryName)).show()
```

- HQL has a similar function and is much easier to use

```
tr1 = spark.sql("SELECT RegionID,
collect_list(TerritoryName) AS TerritoryList
FROM Territories
GROUP BY RegionID")
tr1.show()
tr1.printSchema()
print(tr1.take(1))
```

Complex `collect_list`

- While it could be done using pure Spark methods, it is much easier to use HQL's `NAMED_STRUCT` function to create a Spark array of structs

```
SELECT r.RegionID, r.RegionName,
COLLECT_SET(NAMED_STRUCT("TerritoryID", TerritoryID,
"TerritoryName", TerritoryName)) AS TerritoryList
FROM Regions AS r
JOIN Territories AS t ON r.RegionID = t.RegionID
GROUP BY r.RegionID, r.RegionName
ORDER BY r.RegionID
```

```
DataFrame[RegionID: int, RegionName: string,
TerritoryList:
array<struct<TerritoryID:string,TerritoryName:string>>]
```

Exploding

- Sometimes you get data that is nested in this form and you want to flatten it out
- This could happen if you read from a complex data source like JSON, XML, or a Hive table that is structured this way
- To flatten out an embedded array of values, there is a Spark function called explode

```
from pyspark.sql.functions import explode
tr1.select('RegionID', explode('TerritoryList')).show()
```

- HQL has a similar function

```
tr1.createOrReplaceTempView('RegionTerritories')
sql = """SELECT RegionID, TerritoryName
FROM RegionTerritories
LATERAL VIEW EXPLODE(TerritoryList)
EXPLODED_TABLE AS TerritoryName
ORDER BY RegionID, TerritoryName"""
spark.sql(sql).show()
```

Complex Exploding

- If the object you are exploding contains a structure, you simply use dot syntax to drill down into the subelements

```
sql = """SELECT RegionID, RegionName,
Territory.TerritoryID AS TerritoryID,
Territory.TerritoryName AS TerritoryName
FROM RegionTerritories
LATERAL VIEW EXPLODE(TerritoryList) EXPLODED_TABLE AS
Territory"""
spark.sql(sql).show()
```

User Defined Functions

- Sometimes it is necessary to write complex functions using Python
- Import the helper functions in `pyspark.sql`

```
from pyspark.sql.functions import udf
from pyspark.sql.types import *
from pyspark.sql.functions import to_date
```

- Write whatever custom function you need

```
def city(x):
    return x[:x.find(',')]
def country(x):
    return x[x.find(',') + 1 :]
```

- Call the built-in function or use the `udf` function to wrap and call your UDF

```
df4.withColumn('City', udf(city, StringType())(df4.CityCountry)) \
.withColumn('Country', udf(country, StringType())(df4.CityCountry)) \
.withColumn('Date', to_date(df4.Date, 'dd-MMM-yy')) \
.drop(df4.CityCountry)
```

Closing Thoughts

- Ultimately, Spark is just a processing engine that can read data from almost any source
  - Local files would be read in and distributed into the cluster
  - HDFS files are already in a cluster and could be parallel loaded into Spark nodes simultaneously
  - Data in a SQL table would be similar to a local file and would be fed into the cluster as it retrieves it from the SQL server
  - Data in a NoSQL cluster is already partitioned in a cluster and like HDFS could be parallel loaded
- You will get the fastest load performance for both reading and writing when the data is already stored in a clustered environment
- There are many built-in data sources
  - JSON
  - CSV
  - Parquet
  - ORC

Closing Thoughts (continued)

- You can also use JDBC and `format` to load in custom formats
  - AVRO
  - Cassandra
  - Mongo
  - HBase
- The `os.environ` setting is used to add the custom jars necessary to support these features
- Here is an example of starting a Spark session which could read and write data in a Cassandra cluster

```
import os
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages
com.datastax.spark:spark-cassandra-connector_2.11:2.3.0 --conf
spark.cassandra.connection.host=192.168.0.123,192.168.0.124
pyspark-shell'
table_df = sqlContext.read \
.format("org.apache.spark.sql.cassandra") \
.options(table=table_name, keyspace=keys_space_name) \
.load()
```

Chapter Summary

In this chapter, we have:

- Introduced Spark SQL
- Used SQL on DataFrames
- Discussed Hive integration

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

CHAPTER 4: CLUSTER ANALYSIS

Chapter Objectives

In this chapter, we will:

- Explore cluster analysis
- Use K-Means algorithms

Chapter Concepts

Cluster Analysis

Algorithms

Chapter Summary

Cluster Analysis

- Analysis tool to help make sense of the data before feeding it into other models
- Unsupervised
  - More about discovering patterns in data
  - Not about predicting values for unknown values
- Looks for natural groupings among the data
  - Voter groups (is it just left vs. right, or left, right, center, or more)
  - Species identification (are two groups of organisms different enough to be considered a different species or not)
  - Identify different types of customers we may have
- Often helpful as a preparatory step before classification to determine how many categories we may want to predict

Types of Cluster Analysis

- There are two main approaches to solve this
  - Top down (K-Means)
  - Bottom up (Hierarchical clustering)
- Both rely on the notion of similarity
  - Objects are similar if they share common attributes to others
  - The more similar they are, the closer they are to one another
  - If something is far away in similarity to one thing, it may be closer to something else
- Ultimately the goal is to take a large sample of data and break it up into a small number of meaningful groupings that shed insight as to what the data means

Dataset

- For this example, we have a small easy to follow dataset of the latitude and longitudes of a few Tesla superchargers

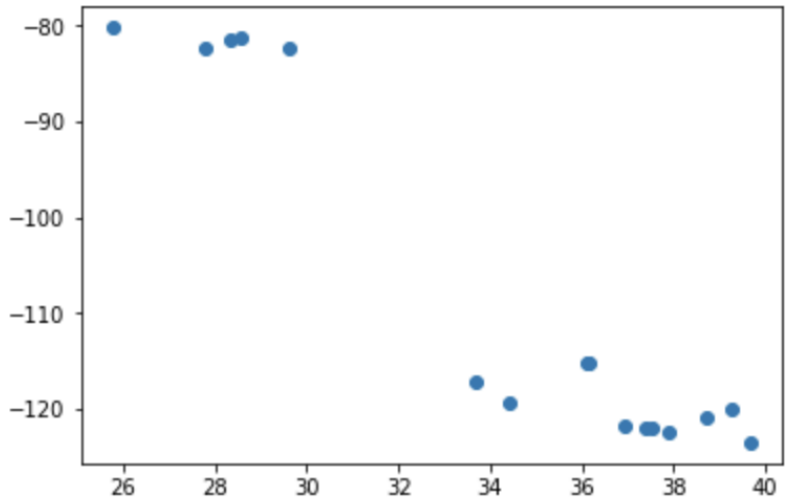
```
filename = 'superchargers.csv'
df = spark.read.csv(f'/home/student/ROI/Spark/{filename}',
header = True, inferSchema = True)
display(df)
```

	lat	lng
0	33.679646	-117.174095
1	28.331356	-81.532453
2	37.413353	-121.897995
3	37.525905	-122.006624
4	37.919969	-122.348976
5	38.730606	-120.788085
6	39.250765	-119.948927
7	36.916349	-121.773512
8	34.441994	-119.258898
9	36.116710	-115.168258

Visualize the Data

- It is often helpful to visualize the data by plotting it
  - There are only two features in this set so it's easy to plot
  - You can also plot a 3D graph for three features
  - Beyond that, it's hard to visualize more features

```
p = df.toPandas()
import matplotlib.pyplot as plt
plt.plot(p.loc[:,'lat'],p.loc[:,'lng'],'o')
```



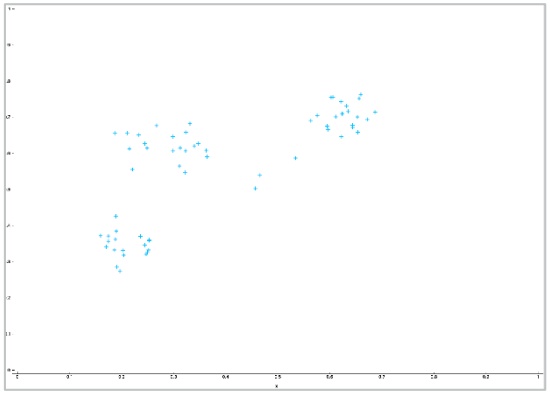
Chapter Concepts

Cluster Analysis

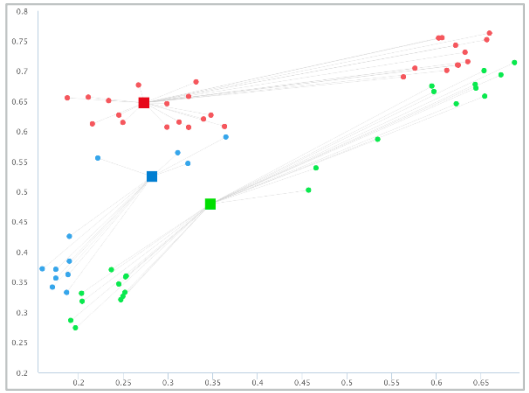
Algorithms

Chapter Summary

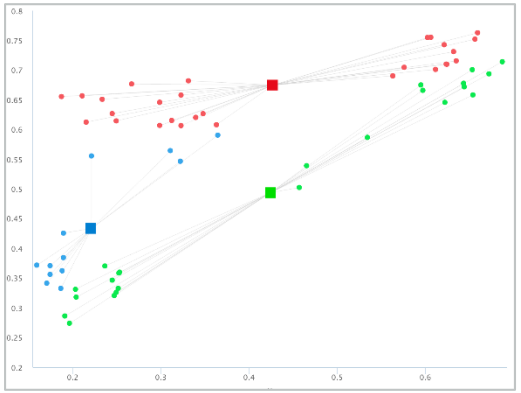
K-Means in Actions



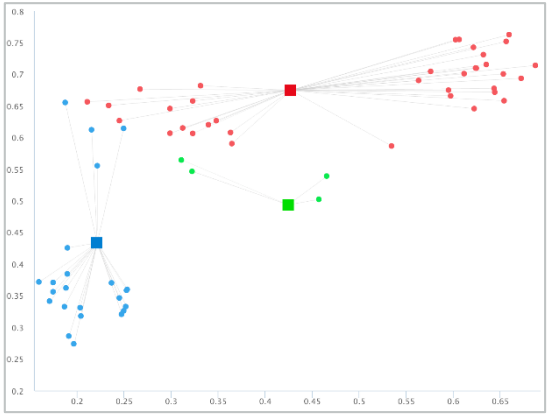
Random Data



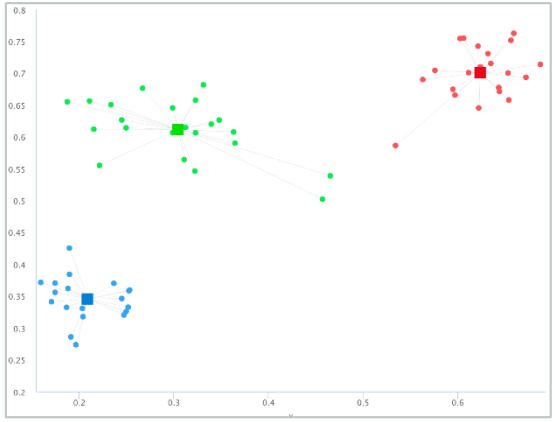
Random Centroids



Adjust Centroids



Reassign Membership

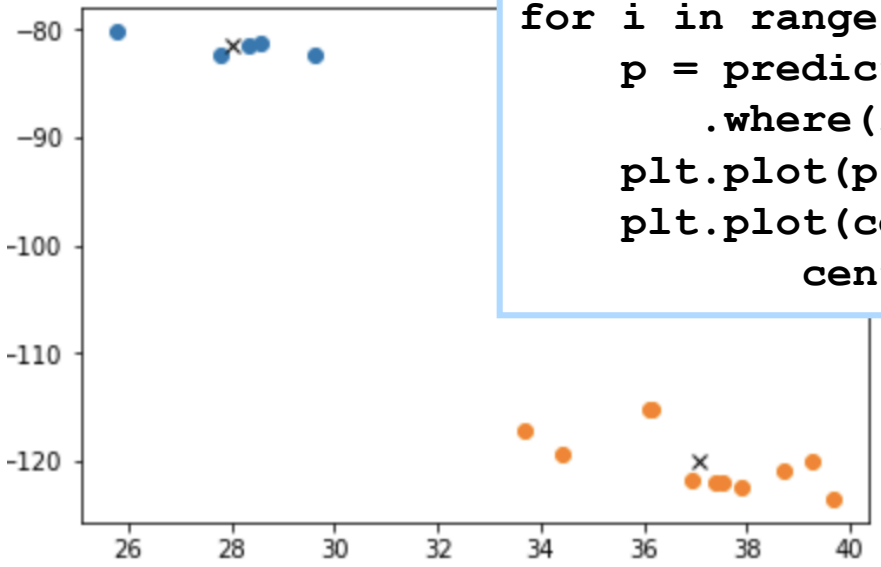


Keep Doing Until Stops Changing

Run K-Means

- Just eyeballing it, let's try out two clusters and plot the results

```
import matplotlib.pyplot as plt

CLUSTERS = 2
kmeans = KMeans().setK(CLUSTERS).setSeed(1)
model = kmeans.fit(dfML.select('features'))
predictions = model.transform(dfML)
centroids = model.clusterCenters()

for i in range(CLUSTERS):
    p = predictions.select('lat', 'lng') \
       .where(f'prediction = {i}').toPandas()
    plt.plot(p.loc[:,'lat'],p.loc[:,'lng'],'o')
    plt.plot(centroids[i][0],
             centroids[i][1],'kx')
```

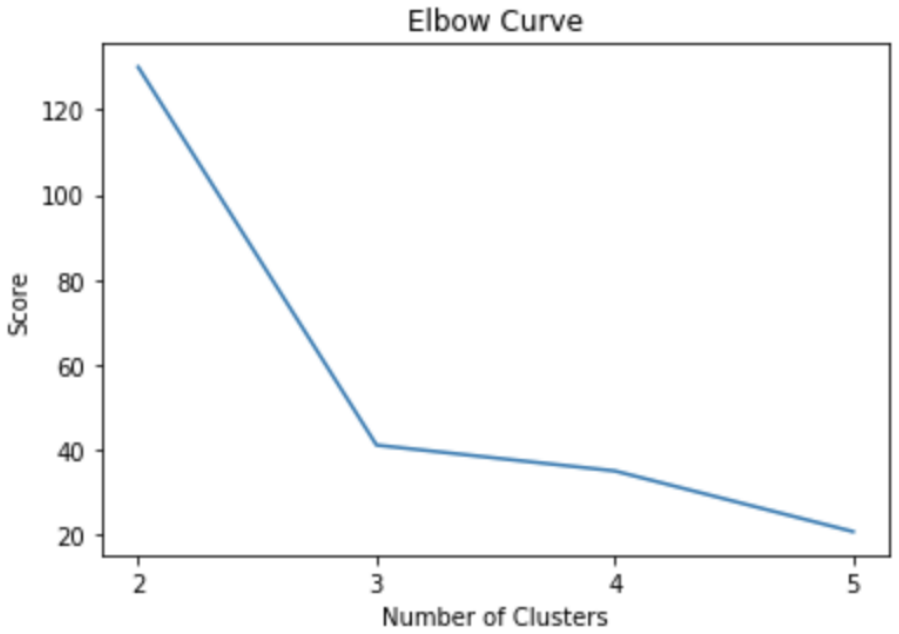


Evaluate K-Means

- With many features and rows it is difficult to visualize how many clusters if right
- There are two measures that are helpful, lower numbers are better
  - Within set sum of squared errors
  - Silhouette with squared Euclidean distance
- But it's not just about a lower number, it's about finding the marginal difference between each number until more clusters don't add any more distinctions
- Use the helper functions to view these numbers and the centroids
- For the small set of data we have, there is a big gain going from 2 to 3 clusters, but not much more going from 3 to 4, so 3 clusters is probably the right value

Elbow Chart

- Here the results are very clear cut, but sometimes the data overlap and don't fit nicely into a particular cluster
- It is often helpful to run a chart that helps figure out how many clusters is ideal
  - Too few and the items are too dissimilar
  - Too many and the additional distinctions become trivial
    - Is there much difference between a brown poodle and a chocolate poodle?



Chapter Concepts

Cluster Analysis

Algorithms

Chapter Summary

Next Steps

- The unsupervised models of clustering do not make predictions so much as they help understand the data
- Another unsupervised model to explore is association rules
  - Used to describe patterns like "people who like X also like Y"
- Principal Component Analysis
- Dimension Reduction

Chapter Summary

In this chapter, we have:

- Explored cluster analysis
- Used K-Means algorithms

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

CHAPTER 5: REGRESSION ANALYSIS

Chapter Objectives

In this chapter, we will:

- Introduce Linear Regression
- Explore data preparation
- Train and test regression model

Chapter Concepts

Regression Analysis

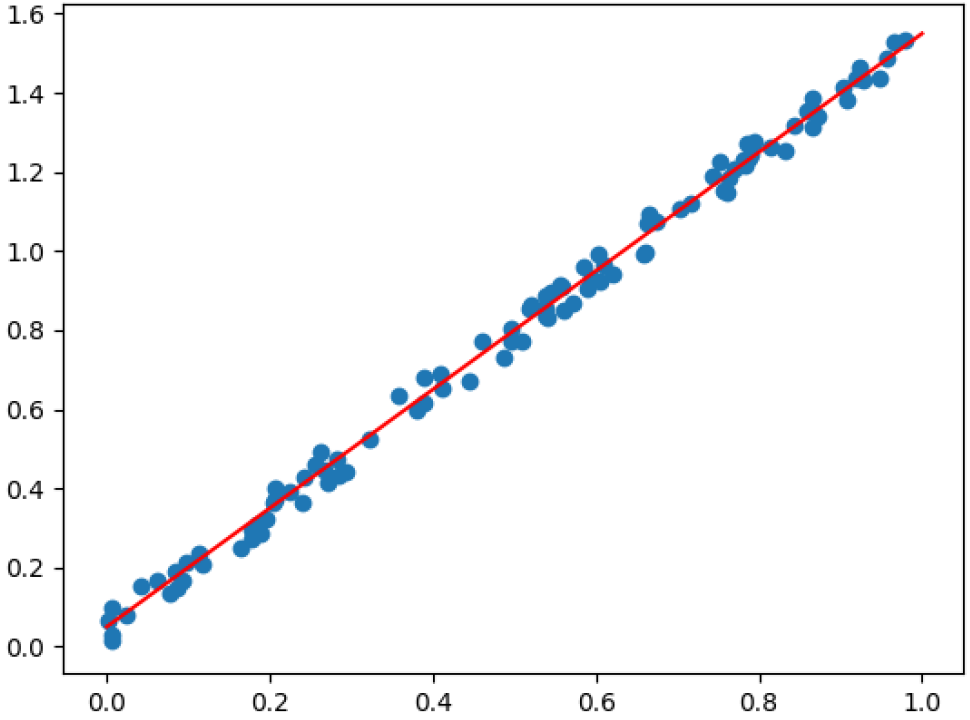
Data Preparation

Algorithms

Chapter Summary

Linear Regression

- Given a collection of X, Y points, you could easily see there is a pattern
- If you remember enough algebra, you could describe the pattern of dots as roughly following the red line, which could be described with the formula y = 1.5x + .01



Linear Regression (continued)

- The idea is that the line that best describes the pattern of dots is the one that has the least distances of the dots from the line
- The formula that describes the line could then be used to predict a value that we have not observed
  - The better the line and formula are at describing that pattern of dots, the more accurate that prediction should be
- Extrapolate this idea onto more than just two axes and instead try to find a line that goes through many different dimensions and you have the idea of multiple linear regression
  - $y = \alpha + \beta 1 x1 + \beta 2 x2 + ... + \beta i xi + \varepsilon$
- Has many use cases
  - Predicting a stock or commodity price
  - Predicting election results
  - Predicting crime rate

Linear Regression (continued)

- Is a supervised model that requires training from a known set of data and testing to see how good it is at predicting before using it for real predictions
- Only works with numeric values
  - Categorical data needs to be dummy encoded
- Does not deal well with missing data, so must be fixed by removing or replacing with central tendency
- There are many algorithms to do this, each with its own pros and cons

Chapter Concepts

Regression Analysis

Data Preparation

Algorithms

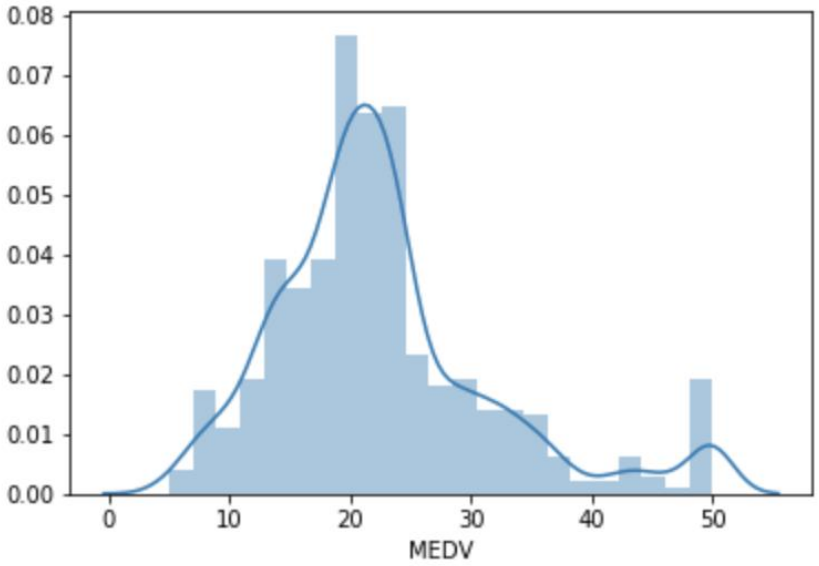
Chapter Summary

Dataset

- For our examples, let's use a public data set of housing data

```
import pandas as pd
import seaborn as sns
sns.distplot(df.toPandas()['MEDV'])
```

- Plotting the distribution of Prices shows that they are normally distributed, except for some outliers, so let's try comparing the model with them and then later filter out




```
dfRaw = dfRaw.where('MEDV < 48')
```

Convert Categorical Features

- Categorical data cannot stay as string, so it must be converted to a numeric format and then into a vector format
- `pyspark` has a class which will transform a column into indexed numbers for each unique string value

```
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol = col, outputCol = col+'_Index')
x = indexer.fit(df).transform(df).select(col, col+'_Index').distinct()
display(x.orderBy(col))
display(x.orderBy(col+'_Index'))
```

- For convenience, use this helper function we made:

```
display(pyh.StringIndexEncode
(df, ['TOWN', 'TRACT']))
```

	TOWN	TOWN_Index
0	Arlington	23.0
1	Ashland	64.0
2	Bedford	61.0
3	Belmont	17.0
4	Beverly	27.0
5	Boston Allston-Brighton	18.0
6	Boston Back Bay	32.0
7	Boston Beacon Hill	54.0
8	Boston Charlestown	31.0
9	Boston Dorchester	12.0

	TOWN	TOWN_Index
0	Cambridge	0.0
1	Boston Savin Hill	1.0
2	Lynn	2.0
3	Boston Roxbury	3.0
4	Newton	4.0
5	Somerville	5.0
6	Boston South Boston	6.0
7	Quincy	7.0
8	Boston East Boston	8.0
9	Brookline	9.0

One-Hot Encoding

- Numerical indexes are good for some algorithms such as Naive Bayes and Decision Trees, but ones that use distance calculations would get distorted
- Need to re-encode this as One-Hot Encoding which creates a separate column for each unique value and fills the columns with zeros and ones
- In Spark, this column needs to be a single Vector column, unlike Pandas which makes a lot of unique columns
- Sparse vectors are hard to interpret visually, but they are not meant for human eyes
- Must first re-encode data with `StringIndexer`

```
from pyspark.ml.feature import OneHotEncoderEstimator
encoder = OneHotEncoderEstimator(inputCols=[col + '_Index'],
outputCols=[col+'_Vector'])
display(encoder.fit(df).transform(df))
```

One-Hot Encoding (continued)

- `SparseVector` version

	TOWN	TOWN_Index	TOWN_Vector
0	Cambridge	0.0	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
1	Boston Savin Hill	1.0	(0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
2	Lynn	2.0	(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
3	Boston Roxbury	3.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
4	Newton	4.0	(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...
5	Somerville	5.0	(0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, ...
6	Boston South Boston	6.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, ...
7	Quincy	7.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, ...
8	Boston East Boston	8.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, ...
9	Brookline	9.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...

- Helper function to call `StringIndexer`, then `OneHotEncoder`

```
display(pyh.OneHotEncode(df, ['TOWN', 'TRACT']))
```

Putting It All Together

- You have to `OneHotEncode` all categorical data, then assemble all the features into one vector and the target variable into another
- Spark provides the VectorAssembler class to do this
- Our helper function makes the whole process more convenient
- Just pass in a DataFrame, list of categorical, numeric, and target columns and it returns a DataFrame with the two columns needed for machine learning algorithms

```
dfML = pyh.AssembleFeatures(df, categorical_features,
numeric_features, target_label = 'target', target_is_categorical
= False))
```

Explore Numerical Features

- Generally, you want to take a look at the numerical features and get standard measurements like min, max, mean, std
  - DataFrames have a `describe` method which makes that easy
- The provided helper functions make that easier

```
numeric_features = ['totalvolume','PLU4046', 'PLU4225',
'PLU4770', 'smallbags', 'largebags', 'xlargebags']
display(df.select(numeric)describe())
```

	summary	CRIM	ZN	INDUS	CHAS	NOX
0	count	487	487	487	487	487
1	mean	3.663863696098563	10.944558521560575	11.155215605749499	0.059548254620123205	0.5544979466119098
2	stddev	8.745039991517844	22.587028902677194	6.820162970724796	0.23689130625554344	0.11678383814441988
3	min	0.00632	0.0	0.74	0	0.385
4	max	88.9762	100.0	27.74	1	0.871

Prepare the Data

- Data must be in a DataFrame of two vectorized objects
  - Features will contain all the independent variables
  - Target will be the dependent variable we are trying to predict
- The provided helper functions make that easier
- Then split the data into a train and test set with the `randomSplit` function

```
import pyspark_helpers as pyh

numeric_features = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', \
             'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO']
categorical_features = ['TOWN', 'TRACT']
target_label = 'MEDV'
df = dfRaw.select(categorical_features + numeric_features +
[target_label])
dfML = pyh.MakeMLDataFrame(df, categorical_features, \
       numeric_features, target_label, False)

train, test = dfML.randomSplit([.7,.3], seed = 1000)
```

Chapter Concepts

Regression Analysis

Data Preparation

Algorithms

Chapter Summary

Run the Model

- Create and instance of the regression class
- There are several to choose from
  - LinearRegression
  - GeneralizedLinearRegression
  - DecisionTreeRegressor
  - RandomForestRegressor
  - GBTRegressor
  - AFTSurvivalRegression
  - IsotonicRegression

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol = 'features', labelCol='target', \
     maxIter=10, regParam=0.3, elasticNetParam=0.8)
lrModel = lr.fit(train)
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
print("Root Mean Squared Error: {}\nR Squared (R2) {}" \
  .format(lrModel.summary.rootMeanSquaredError, lrModel.summary.r2))
```

Run the Test

```
lrPredictions = lrModel.transform(test)
display(lrPredictions.select("prediction","target","features"), 30)
from pyspark.ml.evaluation import RegressionEvaluator
lrEvaluator = RegressionEvaluator(predictionCol="prediction", \
            labelCol="target",metricName="r2")
testResult = lrModel.evaluate(test)
print("Root Mean Squared Error on Test set: {}" \
            .format(testResult.rootMeanSquaredError))
```

```
Coefficients: [-0.09835471749252538,0.005332754299371162,-0.10096929421151506,0.0,-5.571116
7,-0.042132054505705695,-0.42612539816791184,0.0,-0.004595425300618804,-0.5955258325154016]
Intercept: 9.933834380915346
Root Mean Squared Error: 4.054438240179903
R Squared (R2) 0.6990028588000552
```

25	5.084273	13.8	[18.4982, 0.0, 18.1, 0.0, 0.668, 4.138, 100.0,...
26	18.237093	14.0	[0.2909, 0.0, 21.89, 0.0, 0.624, 6.174, 93.6, ...
27	15.081058	14.3	[0.88125, 0.0, 21.89, 0.0, 0.624, 5.637, 94.7,...
28	18.499367	14.3	[5.58107, 0.0, 18.1, 0.0, 0.713, 6.436, 87.9, ...
29	16.793500	14.8	[5.66637, 0.0, 18.1, 0.0, 0.74, 6.219, 100.0, ...

```
Root Mean Squared Error on Test set: 4.370645041749265
```

Chapter Concepts

Regression Analysis

Data Preparation

Algorithms

Chapter Summary

Next Steps

- Regression has a lot more complexity to it once you master the basics
- Some subjects to explore in this area:
  - Under- and over-fitting a model
  - Correlation between the independent variables

Chapter Objectives

In this chapter, we have:

- Introduced Linear Regression
- Explored data preparation
- Trained and test regression model

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

CHAPTER 6: CLASSIFICATION MODELS

Chapter Objectives

In this chapter, we will:

- Understand the use cases for Classification models
- Discuss and compare various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Classification

- There are many business use cases for wanting to predict whether a record will fall into one category or another
  - Is a credit card swipe fraudulent or not?
  - Does a patient have a disease?
  - Will an applicant be a profitable customer?
- Classification can make such predictions by using historical data to train the model to look for patterns
- To see how good a job the model does, you test it with another set of data that was not used to train the model
  - By comparing the known values to the predicted ones, you can judge how well the model performs
- If the model guesses better than a coin flip or random guess, it may not be perfect, but it's better than nothing and could be used to create actionable business decisions with a best guess
- There are many different algorithms that can do classification, so it's best to run the same data through many different models to see which works best for your data

Steps to Classification

- There are some consistent steps you need to do regardless of which algorithm you use, although some models have different requirements
- Usually, categorical data needs to be re-encoded as a vector that is OneHotEncoded
- All models should take the data set and split it into a training and testing set
- First, you fit the model with the training set
  - Can take some time for large datasets
- Once the model is trained, you can see how good it is at making predictions by using a predict function and comparing those values to known values for the variable you are trying to predict
- After you've picked the model that does the best job, you can use it to predict values either individually or in a batch for new data as it comes in

Notes on Classification

- Often you are trying to predict an either/or value
  - Is a card swipe fraudulent or legitimate?
  - Does a patient have cancer or not?
- Not limited to just two choices, you could predict whether a record falls into a category with many different values
  - It just gets trickier sometimes to interpret the results
- The math and techniques behind the scenes can get complicated, but you don't really need to know any of it to use the algorithms
- Just knowing how to identify that you need a classification model and how to prep the data and interpret the results is often good enough to get started
- As you get more sophisticated, you can learn the math behind the scenes to tweak the results and try to get better results

Preparing the Data

- Let's look at some data first, and use this with several different algorithms
- The dataset is whether borrowers defaulted on a loan or not

```
filename = 'bank.csv'
df =
spark.read.csv(f'/home/student/ROI/Spark/datasets/finance/{filename}',
header = True, inferSchema = True)
display(df)
```

- Let's keep the following numerical and categorical features and try to predict whether they default yes/no

```
numeric_features = ['age','balance', 'duration', 'pdays']
categorical_features = ['job', 'marital', 'education', 'housing',
'loan', 'contact', 'campaign', 'poutcome', 'deposit']
target_label = 'default'

df = dfRawFile.select(numeric_features + categorical_features +
[target_label])
```

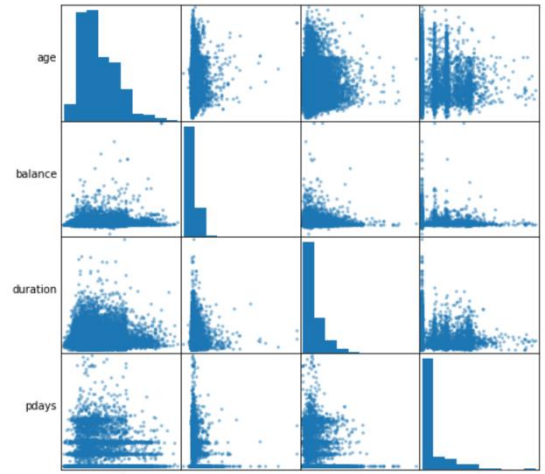
Explore Numerical Features

- Scatter plots are helpful to explore the numerical data

```
pyh.describe_numeric_features(df, numeric_features)
pyh.scatter_matrix(df, numeric_features)
```

	summary	age	balance	duration	\
0	count	11162	11162	11162	
1	mean	41.231947679627304	1528.5385235620856	371.99381831213043	
2	stddev	11.913369192215518	3225.413325946149	347.12838571630687	
3	min	18	-6847	2	
4	max	95	81204	3881	

	pdays
0	11162
1	51.33040673714388
2	108.75828197197717
3	-1
4	854



Change Categorical Column

- Use the helper function we saw earlier in the regression chapter

```
dfML = pyh.MakeMLDataFrame(df, categorical_features,
numeric_features, target_label)
display(dfML)
dfML.printSchema()
display(dfML.groupBy('label').count())
```

	label	features
0	0.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
1	0.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
2	0.0	(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
3	0.0	(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...
4	0.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
5	0.0	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
6	0.0	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
7	0.0	(0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, ...
8	0.0	(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
9	0.0	(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...

```
root
 |-- label: double (nullable = false)
 |-- features: vector (nullable = true)
```

	label	count
0	0.0	10994
1	1.0	168

Saving Processed Data

- If you want, you can save the newly structured DataFrame for future use
- `libsvm` is a good format choice for vectorized data

```
# write
dfML.write.format('libsvm').save('testsave')

# read
dfML = spark.read.format('libsvm').load('testsave')
x.printSchema()
display(x)
```

Splitting the Data

- Split the data into training and testing sets using the `randomSplit` function

```
train, test = dfML.randomSplit([.7,.3], seed = 1000)
print (f'Training set row count {train.count()}')
print (f'Testing set row count {test.count()}')
```

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Decision Trees

- Data is split up based on some factor among the independent variables, then evaluated for how good a job it did
- Recursively keeps applying this algorithm over and over until it comes up with a good set of rules
- Pros
  - It's easy
  - Performs well with categorical data and continuous data
  - Transparency lets you see how it made its choices
- Cons
  - Calculations take a lot longer as you add more and more columns
  - Becomes difficult to understand the decision tree as it gets larger

Apply Decision Tree

- Load the module, create a model and train it

```
from pyspark.ml.classification import
DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol = 'features', \
                            labelCol = 'label', maxDepth = 3)
dtModel = dt.fit(train)
```

- You can save the results of the trained model for future use

```
filename1 = filename.replace('.','_') + '_DT_trainedModel'
dtModel.write().overwrite().save(filename1)

# load a saved trained model
dtModel2 = DecisionTreeClassifier.load(filename1)
```

Interpret the Results

- After you train the model, you want to make predictions on the reserved test set and compare them to the known labels to see how well it did
  - `predictions = model.transform(test)`
- There are a lot of measures to see how good of a job it did
  - For convenience, we have wrapped them into a helper function in the `pyspark_helpers` package
  - `predict_and_evaluate` will return the predicted results and show how well it did

```
dtPredictions, dtLog = pyh.predict_and_evaluate(dtModel, test)
```

```
Test Area Under ROC 0.7710594424880141
+----------+-----+
|prediction|count|
+----------+-----+
|       0.0| 3400|
|       1.0|    2|
+----------+-----+

+-----+-------------+----------+--------------------+
|label|rawPrediction|prediction|         probability|
+-----+-------------+----------+--------------------+
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[1378.0,46.0]|       0.0|[0.96769662921348...|
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[1378.0,46.0]|       0.0|[0.96769662921348...|
```

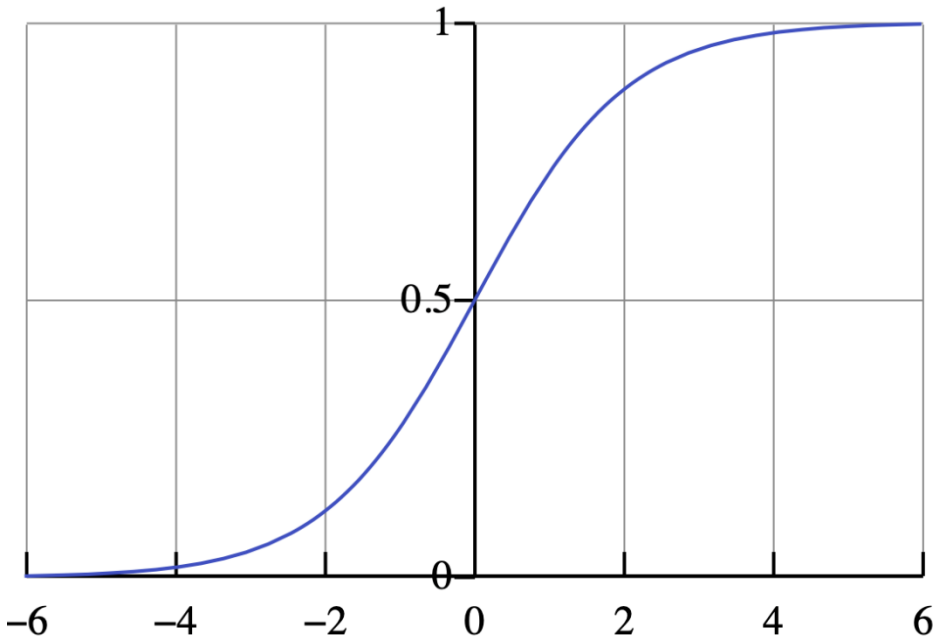
```
Area under ROC = 0.7710594424880141
Area under AUPR = 0.05689643768647529

Overall
tprecision = 0.9891240446796002
recall = 0.9891240446796002
F1 Measure = 0.9891240446796002
```

```
Label 0.0
tprecision = 0.9897058823529412
recall = 0.9994059994059994
F1 Measure = 0.9945322890498005

Label 1.0
tprecision = 0.0
recall = 0.0
F1 Measure = 0.0

Confusion Matrix
[[3.365e+03 2.000e+00]
 [3.500e+01 0.000e+00]]
 PC FP
 FN PW
[[9.89124045e+01 5.87889477e-02]
 [1.02880658e+00 1.08759553e+00]]
```

Logistic Regression

- Another alternative to categorizing, but with a twist
  - Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- The math behind it involves logarithms and finding coefficients of the independent variables

$$\ell = \log \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$



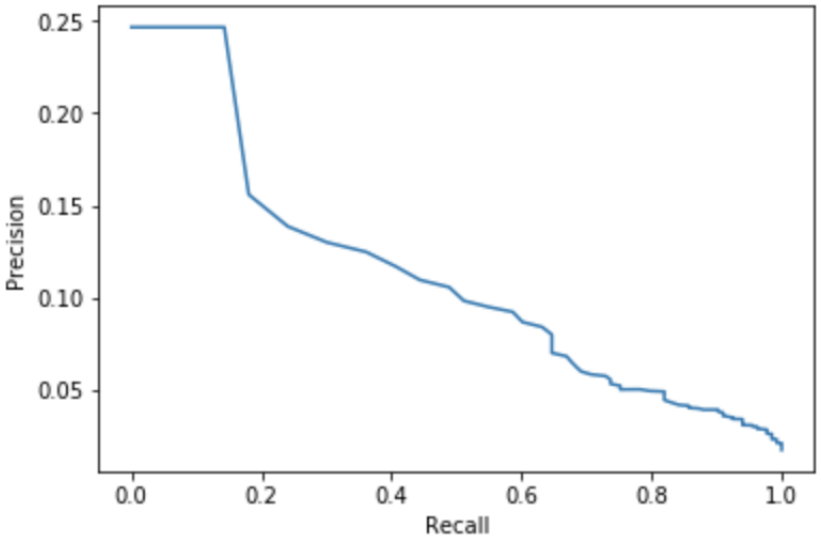
Logistic Regression (continued)

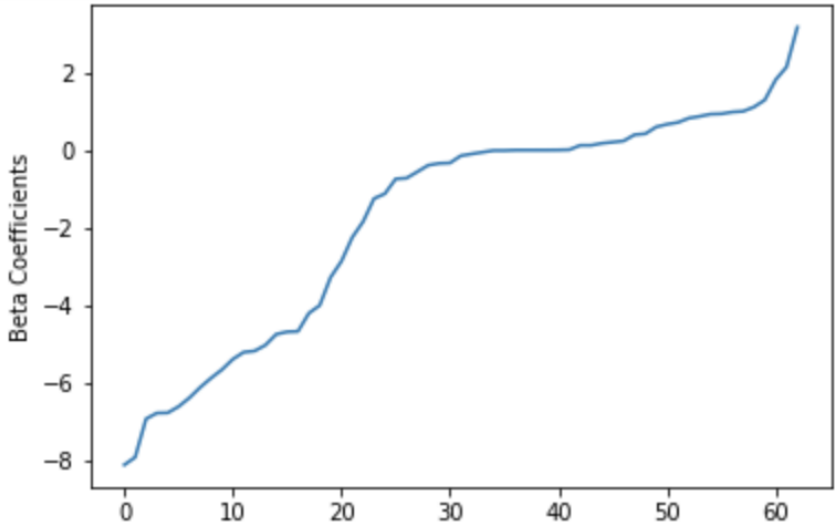
- Data needs to be dummy encoded skipping one value as a reference value
- Pros
  - Works better in cases with low signal to noise ratio
  - Allows for tweaking of false positives and false negatives
  - Transparency lets you see how it made its choices
- Cons
  - Does not perform well with too many features (independent variables)
  - Not good with large number of categorical values within a feature because of dummy encoding

Apply Logistic Regression

- Has some additional measurements to see how well it did

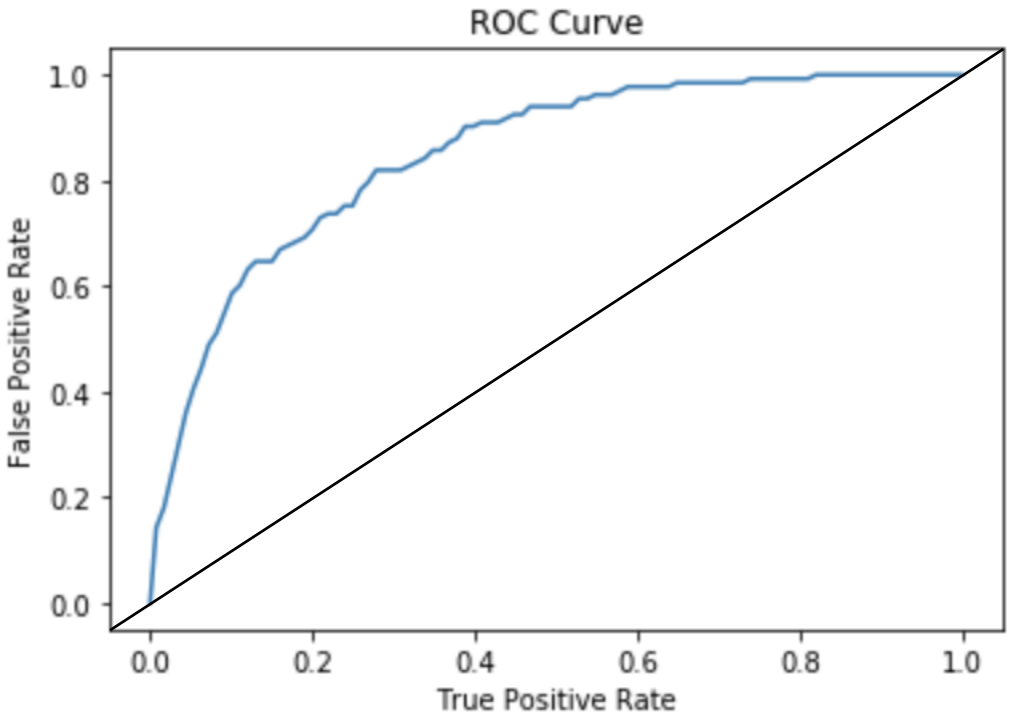
```
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(featuresCol = 'features', \
           labelCol = 'label', maxIter=10)
lrModel = lr.fit(train)
lrPredictions, lrLog = pyh.predict_and_evaluate(lrModel, test)
```





ROC Curve

- Shows the tradeoff between accuracy and sensitivity in adjusting the False Positives
- The closer the curve is to the left or top border, the more accurate it is
- The closer to the 45-degree line, the less accurate it is



Random Forest

- Creates Decision Trees on randomly selected samples of the training set
- Performs multiple iterations and gets prediction results
- Votes on the best random sample
- Pros
  - Often highly accurate due to the strength of multiple predictions
  - Usually does not suffer from overfitting
  - Can see the relative feature importance which is useful in revising the model
- Cons
  - Slow to generate because of multiple iterations
  - Compared to a Decision Tree you cannot really see the path of the tree

Apply Random Forest

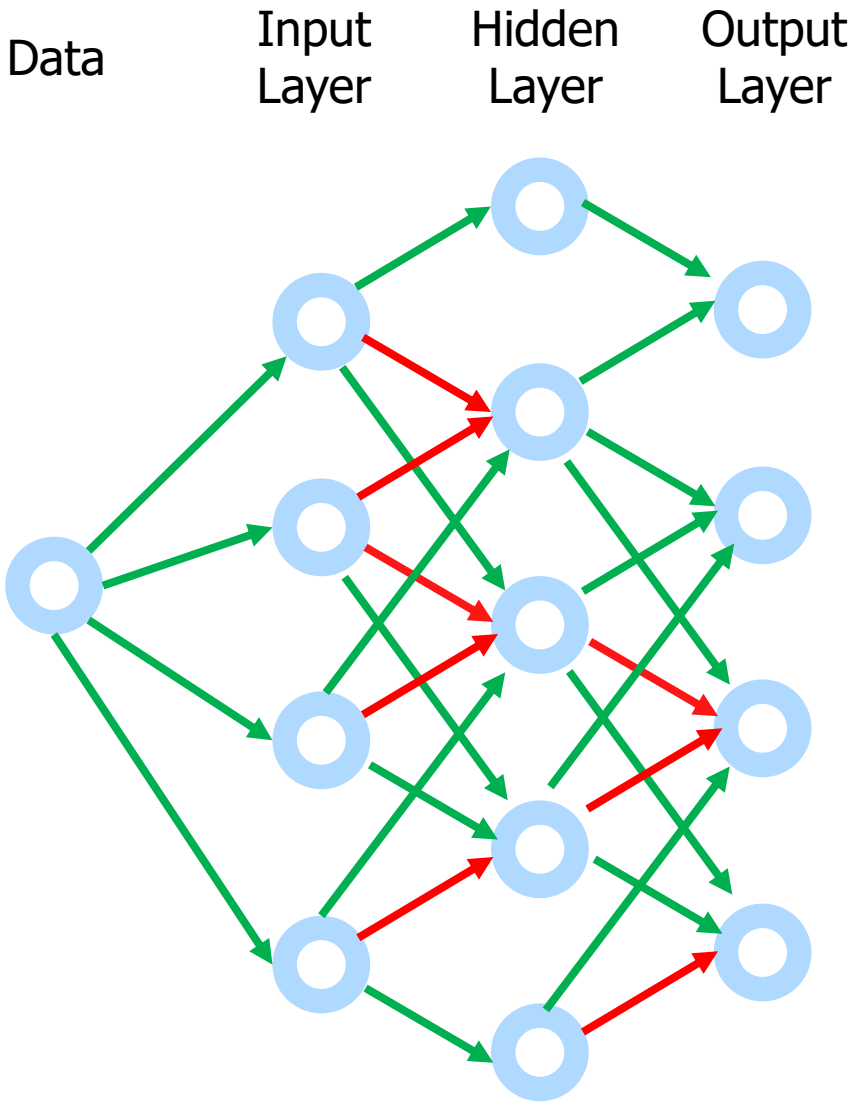
- Load the module, create a model and train it

```
from pyspark.ml.classification import
RandomForestClassifier
rf = RandomForestClassifier(featuresCol = 'features', \
    labelCol = 'label')
rfModel = rf.fit(train)
```

Neural Networks

- Simulates the way the human brain solves
  - Uses a perceptron, algorithm, or function run in multiple layers
- Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- Pros
  - Often perform better than others which can be important where accuracy is desired (predicting cancer)
  - Good for unusual data like image, video, audio
- Cons
  - Black box, you don't know how it made its decision
  - Not appropriate in cases where transparency is important
  - Require a lot more data to train than other models
  - Computationally expensive
- Cool visualization of Neural Network from Google
  - https://playground.tensorflow.org/

Neural Network Visualized



Apply Neural Network

- Need to dummy encode categorical features

```
from pyspark.ml.classification import
MultilayerPerceptronClassifier
from pyspark.ml.evaluation import
MulticlassClassificationEvaluator

# specify layers for the neural network:
# input layer of size 13 (features), two intermediate of size
5 and 4
# and output of size 2 (classes)
layers = [13, 5, 4, 2]

nn = MultilayerPerceptronClassifier(maxIter=100,
layers=layers, blockSize=128, seed=1234)
nnModel = nn.fit(train)
```

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Classification Review

- Classification is one of the most widely used models
- It is supervised
- Good at predicting either/or or multiple-choice categories
- Lots of algorithms
- No one algorithm is best for all situations so often it involves running many of them, documenting the results, and choosing the best for your data and business case
- Can save the results of a lengthy training to a file and reload it for use with the predict function when needed

Next Steps

- Classification is one of the most useful models
- We have explored several different algorithms here and there are tons more, each with its own strengths and weaknesses

Chapter Summary

In this chapter, we have:

- Understood the use cases for Classification models
- Discussed and compared various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark for Data Engineers

COURSE SUMMARY

Course Summary

In this course, we have:

- Explored the Spark platform for using the power of clusters to solve complex queries
- Started with RDDs and seen how they evolved into DataFrames
- Used DataFrames and SparkSQL to run queries on virtually any kind of data
- Taken Datasets and seen how they can be transformed for running machine learning algorithms
- Discovered the basic machine learning models of Cluster, Regression, and Classification

ROI's Training Curricula

Agile Development	.NET and Visual Studio
Amazon Web Services (AWS)	Networking and IPv6
Azure	Oracle and SQL Server Databases
Big Data and Data Analytics	OpenStack and Docker
Business Analysis	Project Management
Cloud Computing and Virtualization	Python and Perl Programming
Excel and VBA	Security
Google Cloud Platform (GCP)	SharePoint
ITIL® and IT Service Management	Software Analysis and Design
Java	Software Engineering
Leadership and Management Skills	UNIX and Linux
Machine Learning and Neural Networks	Web and Mobile Apps
Microsoft Exchange	Windows and Windows Server

Please visit our website at www.ROItraining.com for a complete list of offerings

Ways to Stay in Touch

ROITRAINING
MAXIMIZE YOUR TRAINING INVESTMENT

ROITraining.com

Follow us!
@ROITraining

Connect with us!
/company/roi-training

Like us!
/ROITraining

Follow us!
/roitraining

ROI TRAINING
MAXIMIZE YOUR TRAINING INVESTMENT

The management and staff of **ROI TRAINING** would like to thank you for your commitment to continuing education.

Our mission is to deliver customized technology and management training solutions to large corporations and government agencies around the world. We strive to provide business professionals with the skills and knowledge necessary to increase work performance. We hope the training facilitated by your ROI instructor enables you to successfully apply what you have learned to your work.

We wish you continued success in your career and hope to see you again in the near future.

Best regards,

Brian Reimer and *David Carey*
Co-Founders