Practical Introduction to Spark

SHANG GAO

About Me

PhD Student in the Bredesen Center Data Science Program

Work with Arvind at ORNL

Area of research: Deep Learning for Natural Language Processing

- Develop methods to automatically process information from cancer pathology reports

I’m not an expert in Spark

- I’ve used Spark before on several projects
- Deployed Spark clusters on Amazon Web Services
- If you have questions, I might not know the answer immediately but I’ll try my best to find the answer

Distributed ML Background

In 2004, Google invented MapReduce, a programming framework that could distributed algorithms across a compute cluster

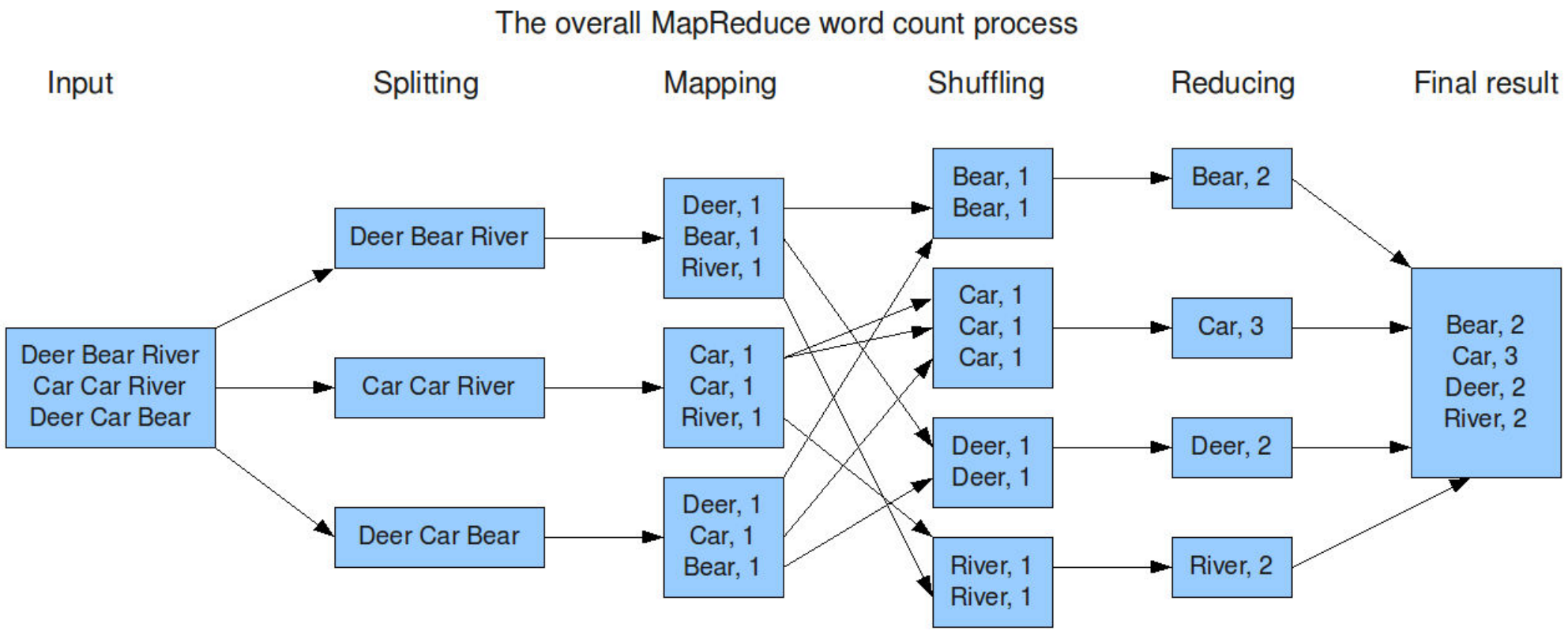
- Split your dataset across multiple machines
- Map – apply the same operation to each entry in parallel
- Reduce – combine the results back together

In 2006, Hadoop was created with the MapReduce algorithm built into the framework

Big companies like Twitter, Facebook, and LinkedIn started using Hadoop

In 2008, Hadoop is acquired by Apache

MapReduce Example



Benefits of MapReduce

MapReduce is not the fastest method for implementing parallelism, there are distributed databases that run faster

Despite that, MapReduce has several major advantages:

- Simple coding method – programmer doesn't need to distribute the data or implement the parallelism
- Scalable – can scale easily across thousands of nodes
- Supports unstructured data – images, sound, text files, etc
- Fault tolerance – failure of individual nodes doesn't stop the process from completing

What is Spark?

Spark started in 2009 in UC Berkeley as a class project on cluster management

Spark was purchased by Apache in 2013

Spark has some overlap with Hadoop, but there are also differences

- Hadoop has a very robust distributed filesystem, and MapReduce allows for easy distributed data manipulation
- MapReduce can be clunky for complex data analytics and machine learning tasks
- Spark was designed to fill in this niche of speeding up complex distributed data analytics operations
- Because Spark was designed with the Hadoop filesystem in mind, many companies use Hadoop for storing data and Spark for data analysis
- Spark has built-in support for SQL queries, streaming data operations, and machine learning algorithms

Spark vs. Hadoop

Hadoop's MapReduce writes to hard-disk after every MapReduce operation

- Most complex problems require many MapReduce operations
- Hadoop will read data from the cluster, perform an operation, write results to the cluster, read updated data from the cluster, perform next operation, write next results to the cluster, etc…

Spark performs most operations in memory

- Spark will read data from the cluster, perform all of the requisite analytic operations at once in memory, and write final results to the cluster

Spark can be up to 10x – 100x faster than Hadoop for complex data analytics operations

Lazy Evaluation

Spark uses lazy evaluation, which means that operations are not run until the final output is requested

For example, suppose we have a dataset with 20 preprocessing steps followed by a print command

- Eager evaluation – run each preprocessing step as soon as it is called, saving the result to memory
- Lazy evaluation – save the instructions for each preprocessing step until the output is required (print), then optimize the instructions and run the minimal required for the output

Lazy evaluation allows for additional optimization of operations to save memory and computing time

Language Support

Spark is written in Scala

Spark also supports Java, Python, and R

- Python (and R) code runs slower on Spark than Scala
- However, Python remains a popular choice for Spark because Python has many more libraries for machine learning and natural language processing than Scala

This tutorial will focus on PySpark

RDD vs DataFrames

Spark has two ways of storing data:

- Resilient Distributed Dataset (RDDs)
- DataFrames (DFs)

RDDs:

- Rows of unstructured data
- Operations are done on the entire row
- Good for functional programming like map, filter, and reduce operations

DataFrames:

- Similar to a database table (or Pandas DataFrame)
- Each column has a type
- Operations can be done on specific columns
- Good for SQL-like queries
- Faster and more optimized than RDDs

RDD vs DataFrames

When data is imported into Spark, it is saved into an RDD or DataFrame

All subsequent Spark operations are done either the RDD or DataFrames

RDDs can easily be converted in DataFrames, and vice versa

RDDs and DataFrames have ML different libraries

- RDDs use MLlib
- Dataframes use ML

This tutorial will focus on DataFrames

- Dataframes are newer than RDDs
- Dataframes are a bit easier to use (require less programming)

Getting Started

Running PySpark locally:

- Install Java
- Install Anaconda
- Download Spark
- Pip install pyspark
- Set environment variables

Setting up a cluster is a lot more complicated

- Maybe use flintrock if setting up temporary clusters

Spark Session

Must initiate a Spark Session before you can use anything Spark related in your script

- Allows you to use DataFrames
- Not to be confused with SparkContext, which allows you to use RDDs

Configuring your Spark Session:

- Basic requirements: name
- Lots and lots of advanced configuration options available, see PySpark documentation

```
11 #start spark session
12 spark = SparkSession\
13         .builder\
14         .appName("TweetClassification")\
15         .getOrCreate()
```

Importing Data

Spark can directly import a wide range of data formats:

- Csv
- Json
- Text

Spark.read directly converts raw data into a Spark DataFrame

- Can infer datatypes when importing

```
17 #import csv
18 df = spark.read.csv("airline_tweets.csv",header=True,inferSchema=True)
```

Examining Data

Spark DataFrames have built in commands to let you examine the contents of the DataFrame

```
20 #show dataframe
21 df.show()
22
23 #count total records
24 print("total records:", df.count())
25
26 #show schema
27 df.printSchema()
28
29 #show one column
30 df.select('text').show(n=5,truncate=False)
31
32 #show one record
33 df.where(df['tweet_id'] == '570306133677760513').show()
```

Casting Data

Spark doesn’t always capture the correct data types when importing data

You can manually cast a column from one datatype to another

- Important because machine learning operations require integers or floats as input, not strings

Common Spark datatypes:

- string, integer, float, double, Boolean, date, timestamp, array

```
35 #casting
36 df = df.withColumn("airline_sentiment_confidence", df["airline_sentiment_confidence"].cast("float"))
37 df = df.withColumn("negativereason_confidence", df["negativereason_confidence"].cast("float"))
38 df.printSchema()
```

Removing Missing Data

`df.where` allows you to filter data by criteria

Example usage:

- `df.where(df['column'] > 5)`
- `df.where(df['column'] != 0)`
- `df.where(df['column'].isNull())`
- `df.where(df['column'].isNotNull())`

```
40 #remove rows missing rating or tweets
41 print("total records:", df.count())
42 df = df.where(df['airline_sentiment'].isNotNull())
43 print("records with sentiment:", df.count())
44 df = df.where(df.['text'].isNotNull())
45 print("records with sentiment and tweet text:", df.count())
```

Restructuring DataFrames

`df.select('col1', 'col2')` allows you to create a new DataFrame using the selected columns from an existing DataFrame

`df.drop ('col1', 'col2')` allows you to create a new DataFrame by dropping columns from an existing DataFrame

`df.join(df2, df1['index'] == df2['index'])` allows you to create a new DataFrame by joining two existing DataFrames on an index column

```
47 #rearrange or drop columns
48 reduced_df = df.select("tweet_id","text","airline_sentiment","airline")
49 reduced_df.show()
50 reduced_df.printSchema()
```

User Defined Functions

User defined functions allow you to apply a custom Python function to every entry in a specified column of a DataFrame

- Similar to mapping in RDDs
- Useful for tasks like data transformation and data cleaning

Step 1: define a Python function

- Should take in one argument if operating on one column
- Can take in one argument for each column if operating on multiple columns

Step 2: convert it into a Spark UDF (user defined function)

- `udf_function = udf(python_function,DataType())`

Step 3: apply the UDF to a DataFrame

- `df = df.withColumn(“output_col", udf_function(“input_col"))`

Data Statistics

`df.count()` counts the total number of records in a DataFrame

- Can be used in conjunction with `df.where` to count entries satisfying specific criteria

`df.describe()` generates summary statistics for each column in a DataFrame

`df.agg({'column':'statistic'})` calculates a statistic on a specified column

- Replace `'statistic'` with `'max'`, `'min'`, or `'avg'`

`df.groupBy('col1').agg({'col2':'statistic'})` groups records in a specific column, then calculates a statistic for each group on another column

Saving DataFrames

DataFrames can be saved to disk in parquet format (stores type information, headers, and other metadata) or other formats:

- `df.write.parquet('filename.parquet')`
- `df.write.csv('filename.csv')`

DataFrames can be reloaded using the `spark.read` method

- `df = spark.read.parquet('filename.parquet')`
- `df = spark.read.csv('filename.csv')`

Data Preprocessing Library

The pyspark.ml.feature module has many useful feature transformation tools:

- Tokenizer and N-Grams
- Word counts and TF-IDF
- One-hot encoding
- Feature Scaling
- PCA and Feature Selection

```
107 #tokenize words
108 tokenizer = Tokenizer(inputCol="clean_text", outputCol="tokens")
109 reduced_df = tokenizer.transform(reduced_df)
110 reduced_df.show()
111 reduced_df.printSchema()
```

Machine Learning Library

The pyspark.ml.classification module contains many useful machine learning algorithms for classification

- Naïve Bayes
- Logistic Regression
- Random Forests
- Support Vector Machines

PySpark ML library also has modules for regression and clustering

```
139 #apply naive bayes
140 nb = NaiveBayes(featuresCol="tfidf", labelCol="airline_sentiment", predictionCol="NB_pred",
141                  probabilityCol="NB_prob", rawPredictionCol="NB_rawPred")
142 nbModel = nb.fit(train)
143 test = nbModel.transform(test)
144 test.show()
```

Launching Jobs on Spark Clusters

Python can be used to run a PySpark script locally on one machine

The `spark-submit` command is used to run a PySpark job on a Spark Cluster

- The arguments passed to the `spark-submit` command determine the number of nodes and configuration settings used to run the job
- Example: `spark-submit --master 192.12.34.56 --deploy-mode client --driver-memory 30g --executor-memory 7g --num-executors 16 --executor-cores 2 --files script.py`
- For additional information on how to use `spark-submit`, see https://spark.apache.org/docs/latest/submitting-applications.html

Additional Resources

PySpark documentation is thorough and has many examples

- http://spark.apache.org/docs/latest/api/python/
- https://spark.apache.org/docs/latest/quick-start.html
- https://spark.apache.org/docs/latest/sql-programming-guide.html

DataCamp also offers helpful tutorials for PySpark

- https://www.datacamp.com/community/tutorials/apache-spark-tutorial-machine-learning
- https://www.datacamp.com/courses/introduction-to-pyspark