Master Big Data Analytics and Data Wrangling

BIG DATA, MAPREDUCE, HADOOP, AND SPARK WITH PYTHON

written by

https://lazyprogrammer.me

Big Data, MapReduce, Hadoop, and Spark with Python

Master Big Data Analytics and Data Wrangling with MapReduce Fundamentals using Hadoop, Spark, and Python

By: The LazyProgrammer (https://lazyprogrammer.me)

Introduction

What’s the big deal with big data?

It was recently reported in the Wall Street Journal that the government is collecting so much data on its citizens that they can’t even use it effectively.

A few “unicorns” have popped up in the past decade or so, promising to help solve the big data problems that billion dollar corporations and the people running your country can’t.

It goes without saying that programming with frameworks that can do big data processing is a highly-coveted skill.

Machine learning and artificial intelligence algorithms, which have garnered increased attention (and fear-mongering) in recent years, mainly due to the rise of deep learning, are completely dependent on data to learn.

The more data the algorithm learns from, the smarter it can become. The problem is, the amount of data we collect has outpaced gains in CPU performance. Therefore, scalable methods for processing data are needed.

In the early 2000s, Google invented MapReduce, a framework to systematically and methodically process big data in a scalable way by distributing the work across multiple machines.

Later, the technology was adopted into an open-source framework called Hadoop, and then Spark emerged as a new big data framework which addressed some problems with MapReduce.

In this book we will cover all 3 - the fundamental MapReduce paradigm, how to program with Hadoop, and how to program with Spark.

Advance your Career

If Spark is a better version of MapReduce, why are we even talking about it?

Good question!

Corporations, being slow-moving entities, are often still using Hadoop due to historical reasons. Just search for “big data” and “Hadoop” on LinkedIn and you will see that there are a large number of high-salary openings for developers who know how to use Hadoop.

In addition to giving you deeper insight into how big data processing works, learning about the fundamentals of MapReduce and Hadoop first will help you really appreciate how much easier Spark is to work with.

Any startup or technical engineering team will appreciate a solid background with all of these technologies. Many will require you to know all of them, so that you can help maintain and patch their existing systems, and build newer and more efficient systems that improve the performance and robustness of the old systems.

Amazingly, all the technologies we discuss in this book can be downloaded and installed for FREE. That means all you need to invest after purchasing this book is your effort and your time. The only prerequisites are that you are comfortable with Python coding and the command line shell. For the machine learning chapter you’ll want to be familiar with using machine learning libraries.

BONUS: At the end of this book, I’ll show you a super simple way to train a **deep neural network** on Spark with the classic MNIST dataset.

Formatting

I know that the e-book format can be quite limited on many platforms. If you find the formatting in this book lacking, particularly for the code or diagrams, please shoot me an email at info@lazyprogrammer.me along with a proof-of-purchase, and I will send you the original ePub from which this book was created.

Chapter 1: MapReduce Fundamentals

In this chapter we are going to look at how to program within the MapReduce framework and why it works.

You’ll see that MapReduce is effectively just a really fancy for-loop.

Once you master the MapReduce paradigm, coding with MapReduce will simply just become “regular programming”.

Because MapReduce is designed to scale to big data, there is a lot of overhead involved in setting things up. Once you understand this overhead (both in terms of computational overhead and effort on your part), big data programming is just programming.

What is MapReduce?

The big question.

You may have read that MapReduce is a framework that helps you process big data jobs in a scalable way.

You may have read that MapReduce scales by parallelizing the work to be done.

You may have read that MapReduce proceeds in 2 steps: mapping, followed by reducing, which are done by processes called mappers and reducers, respectively.

I don’t find these descriptions particularly helpful.

They certainly don’t get you any closer to actually coding MapReduce jobs.

They are more like a high-high-high-high-high level overview that perhaps a CEO who is not interested in the technical details would need to know.

As a programmer, the above description is not useful.

It is best to think about what the data “looks like” as we proceed from one step to the next.

The most straightforward way to think about big data processing is to think about text files.

Text files will consist of a number of lines, each of which can be read in one at a time.

In Python, it might look something like this:

```
for line in open(‘input.txt’):
  do_something(line)
```

As you can see, this time complexity of this code is O(N). It will take twice as long to process a file that is twice as large.

An obvious solution to this would be to break up the work.

If we have a dual-core CPU, we could split up the file into 2 files of equal size: input1.txt and input2.txt.

Run the same program twice on each of the files, each will run on a separate core, thus taking half the time.

If we have a quad-core CPU, we could split up the file into 4 parts and it would take a quarter of the time.

Before MapReduce came about, this is how you would scale. Just split up the data, and spread out the computation across different CPUs, even different machines (if they have network capabilities).

The problem with this is it’s not systematic enough. It’s not general enough that we can have just one framework that takes care of all the details for us. Some of these details might include:

- One machine dies during computation. How do you know what it was working on so that you can then pass that job to another machine?

- How would you even keep track of the fact that it died?

- One machine is taking significantly longer than the others. How can you make sure you’re spreading the work evenly?

The MapReduce framework includes a “master node” that keeps track of all these things, and even runs an HTTP server so that you can track the progress of a job.

To understand the MapReduce framework as an algorithm, you don’t need to understand “which machine does what”. What you do need to understand is the map function and the reduce function.

These are done by processes called mappers and reducers, respectively.

[Side note: We focus on implementation in this book, because when you join a new company as a big data hire, these systems are likely already setup by the DevOps team. Your job will be the coding part.]

The best way is to learn by example, so that is what we are going to do now.

The canonical example for MapReduce is word counting. You have a huge number of text documents (notice that this implies the data is already split up, not just one huge file), and you want to output a table showing each word and its corresponding count.

If you were to do this in regular Python, it would look something like this:

```
# count the words
word_counts = {}
for f in files:
  for line in open(f):
    words = line.split()
    for w in words:
      word_counts[w] = word_counts.get(w, 0) + 1

# display the result
for w, c in word_counts.iteritems():
  print “%s\t%s” % (w, c)
```

Note that in big data processing, we often print data out in a table format. Implementation-wise, this means we use CSV or some other character-delimited format (tabs and pipes are also common).

What would this look like in MapReduce?

mapper.py

```
for line in input:
  words = line.split()
  for w in words:
    print “%s\t%s” % (w, 1)
```

reducer.py

```
for key, values in input:
  print “%s\t%s” % (key, sum(values))
```

“input” here is abstracted and can be different in different contexts. In mapper it can represent lines of a text file, or it can be instances of an object defined elsewhere in your code. In reducer it can be a HashMap (dictionary) of keys and corresponding values collected from the mapper, or it can also just be lines of text as you’ll see when we discuss Hadoop Streaming.

This short example illustrates a few key points in the MapReduce framework.

First, the reducer receives something that is very much like a (Python) dictionary. In Java (the language Hadoop and Spark are written in) the corresponding object is a HashMap.

The key is the first thing that the mapper produces. The value is the second thing that the mapper produces.

Notice that this current example doesn't restrict the mapper from returning more things. In Java, the mapper actually returns an object which has a key and a value.

However, our focus will be on Hadoop Streaming, which doesn't require us to use Java and is much more flexible. We will thus stick to the Hadoop Streaming way of doing things, which is to produce lines of text as output.

The reducer then receives the key and values. Notice that's values plural, not value.

Why?

With the word count example, it's easy to see - we're going to come across the same word more than once.

The word is the key. The count is the value.

When the reducer receives this data, it gets the key, and all the corresponding values at the same time (a list of 1s).

Since we have a 1 for every time the word shows up in the corpus, then the sum of those 1s will be the word count for that word.

How does the MapReduce framework know to collect all the values by key in between the mapper and reducer?

This is an intermediate step called "combining", done by a process appropriately named the “combiner”.

Again, we are not worried about the particular components, but how to write the code.

How can we make this more efficient? Well you can imagine combining a bunch of 1s takes time. In the end we know all we want is the sum anyway.

So we can make the mapper produce a partial sum. The reducer remains the same.

Better mapper.py:

```
counts = {}
for line in input:
  words = line.split()
  for w in words:
    counts[w] = counts.get(w, 0) + 1

for w, c in counts.iteritems():
  print “%s\t%s” % (w, c)
```

This mapper will collect only unique words and their counts for a given document, so that the combiner has less to deal with when it passes on data to the reducer.

You’ll see that this looks almost the same as our single-for-loop word count. What’s different is that you can potentially have 1000s of mappers running simultaneously across machines.

Before we move on to writing code, let’s discuss how you’re going to setup your development environment (if you haven’t done so yet).

Setting up your Development Environment

In this section we’re going to discuss how to set up your development environment.

All the work we do will work on Mac, Linux, and Windows, although our focus will be on Mac and Linux.

There are various installers available for Windows (just Google “Hadoop for Windows”), but for a consistent experience I would recommend installing a VM or spinning up a VM in the cloud with Ubuntu. You are welcome to work on the platform of your choice but if you have trouble with the exact commands provided in this book, then I would suggest installing a VM and trying the same thing there.

In any case, when you start working on big data at your real job, you’re not going to be running MapReduce jobs from your Windows or Mac anyway.

Typically, a MapReduce cluster will be setup on AWS (Amazon Web Services) or some other cloud provider, where the OS is usually some flavor of Linux. So it’s better to get used to it now.

Computers are powerful enough these days, and virtualization technologies mature enough, that running a lightweight Ubuntu distro on your main machine will have near-native performance.

That said, let’s get to specifics. If you’re not on Windows or you just want to use an Ubuntu VM, download Virtualbox from virtualbox.org.

Next, grab a lightweight version of Ubuntu, like Lubuntu, from lubuntu.net

Installation should be straightforward from there. A few clicks, and you’re done.

As a bonus, if you’re on Mac or Linux, Python is already installed. No more to do for now.

Full Code for Word Count

Because we still haven’t discussed Hadoop, realize this is a pared down version of MapReduce just so you can see it work. Note that all the code for this class can be downloaded from https://github.com/lazyprogrammer/big_data_class.

The folder simple_word_count contains the relevant files for this example.

We will use this simple input file, input.txt (of course you are free to substitute this with your own):

```
cat cat dog dog rabbit dog cat
rabbit rabbit dog rabbit cat dog cat cat
```

Next, mapper.py:

```
class Mapper:
  def map(self, data):
    returnval = []
    counts = {}
    for line in data:
      words = line.split()
      for w in words:
        counts[w] = counts.get(w, 0) + 1
    for w, c in counts.iteritems():
      returnval.append((w, c))
    return returnval
```

Next, reducer.py

```
class Reducer:
  def reduce(self, d):
    returnval = []
    for k, v in d.iteritems():
      returnval.append(“%s\t%s” % (k, sum(v)))
    return returnval
```

And finally, main.py, which emulates what the MapReduce framework is doing when it calls your mapper and reducer.

```
from mapper import Mapper
from reducer import Reducer

class JobRunner:
  def run(self, Mapper, Reducer, data):
    # map
    mapper = Mapper()
    tuples = mapper.map(data)

    # combine
    combined = {}
    for k, v in tuples:
      if k not in combined:
        combined[k] = []
      combined[k].append(v)

    # reduce
    reducer = Reducer()
    output = reducer.reduce(combined)

    # do something with output
    for line in output:
      print line

runner = JobRunner()
runner.run(Mapper, Reducer, open(‘input.txt’))
```

This is the largest file and thus, why we’ve spent so much time explaining what happens around calling the mapper and reducer. Note that we could add even more detail by creating more mappers and having them each process a different line of the input file. We could even do them in separate threads, effectively parallelizing the map portion. Similarly, we could parallelize the reduce portion using threads as well.

We will of course not do this since we’ll soon be moving to Hadoop, which already does that stuff.

The output is of course:

```
dog     5
rabbit  4
cat     6
```

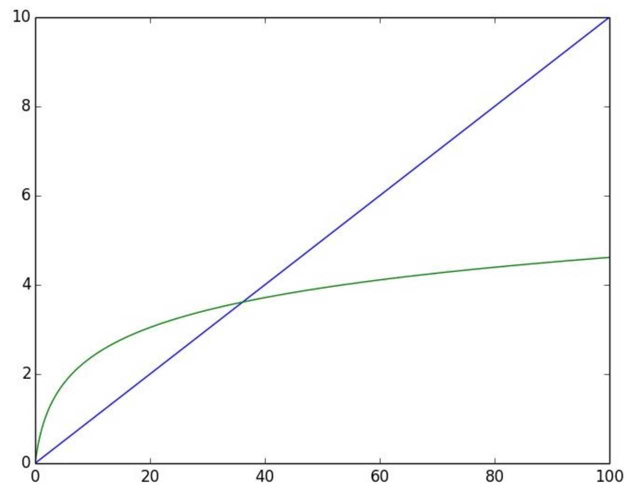
MapReduce Performance Characteristics

Many engineers (and managers) who are new to big data often jump to the conclusion that they must use big data technology in their stack too early. If you have 1GB, or even 10GB of data, that’s not necessarily “big”.

You should take your specific needs into account. Is your data likely to remain at 10GB, or is it growing by 10GB everyday?

Keep in mind, there is a lot of overhead involved with setting up a MapReduce job. This figure shows typical time vs. input size curves for both a traditional for loop and MapReduce jobs. The growth of a for loop simply grows linearly with the data. Twice as much data takes twice as much time to process. MapReduce starts with a steep curve at the beginning (there is always some set cost to starting a MapReduce job due to overhead), but the cost increases slow down as the data gets larger.

Note that this is simplified - there are a lot more variables such as the number of machines in your cluster. This figure just illustrates a point.



When the size of your data grows large, the advantage of using MapReduce is most pronounced. However, there is still an area where a regular for-loop outperforms MapReduce, and that’s for “small data” problems.

Aside from clock performance, there is still the issue of engineering effort. MapReduce jobs are not as straightforward as simple single-CPU scripts. Costs to maintain your big data jobs and debug when things go wrong are also higher.

Thus, one should generally be conservative before throwing “big data technology” at data, and consider carefully if it is really needed in the first place.

Chapter 2: Hadoop

While Google invented MapReduce and has their own internal version of it, Hadoop is the rest of the world's MapReduce framework.

Thus, it's the one we'll be using. Most courses on Hadoop start out with Java, but unfortunately 90% of the code is usually cruft that takes away from the main message.

It's the fundamentals that I want to strengthen in this book, so we'll only look at Java where absolutely necessary.

What is Hadoop?

A lot of the time, people think of Hadoop as synonymous with MapReduce. This is sort of sloppy use of terminology.

Hadoop is actually an entire ecosystem of technologies, which implements the MapReduce framework.

For instance, the Hadoop File System (HDFS for short) is a file system which uses replication and sharding to minimize the probability of failure.

With big data, because there are so many machines involved, the probability of failure is high. This is related to a "paradox" called "The Birthday Problem" (it's not actually a paradox, it follows directly from the math).

One result from the birthday problem is that if you have 23 people in a room, the chance that 2 people share a birthday is 50%!

With 57 people, the chance that 2 people share a birthday is 99%!

This is despite the fact that between any 2 people, the chance that they share a birthday is 1/365.

Similarly, the chance of a single machine failing is miniscule. But when you have thousands of them, you’ll have machines failing every hour.

Your algorithms need to account for this.

That’s where HDFS comes in. Your data exists as multiple copies in parts spread across machines to create lots of redundancy. That way, if anything fails, a copy still exists somewhere else.

Spark, a new framework which we will discuss later, can also process files that live on HDFS. Hadoop technologies such as HBase, Cassandra, and Hive are also compatible with Spark.

So Hadoop is not “just MapReduce”. It’s an entire ecosystem of technologies.

What we are really interested in here is:

1) How do we get Hadoop? (i.e. install the ecosystem of technologies that will allow us to write distributed MapReduce jobs)

2) How do we run a MapReduce job in Hadoop? (We already know how MapReduce works in general from the last chapter so it’s just a matter of “fitting” that code to this specific framework)

The rest of this chapter will focus on (1), and the next chapter will focus on (2).

How to Install Hadoop

We will focus on a single-node cluster setup, because, as stated previously, if you’re working with multi-node clusters at your job, it’s most likely already setup for you.

Besides, the single-node cluster setup is complicated enough as it is. You’ll really appreciate Spark when you see how much simpler everything is, from installation to writing code.

Note that the following instructions are for Ubuntu.

The first thing you’ll need to do is install Java (Hadoop is written in Java, and the programs are JAR files).

First update your sources list:

```
$ sudo apt-get update
```

Then install Java:

```
$ sudo apt-get install default-jdk
```

You can verify it is installed by running:

```
$ java -version
openjdk version “1.8.0_91”
OpenJDK Runtime Environment (build 1.8.0_91-8u91-b14-3ubuntu1~16.04.1-b14)
OpenJDK 64-Bit Server VM (build 25.91-b14, mixed mode)
```

Next, we’re going to add a special Hadoop user to do our Hadoop things:

```
$ sudo addgroup hadoop
$ sudo adduser —ingroup hadoop hduser
```

Next, if you don’t already have it, install SSH:

```
$ sudo apt-get install ssh
```

Hadoop uses SSH to communicate between nodes, so we need to configure SSH access to the local machine.

First, switch to the Hadoop user:

```
$ su hduser
```

Next, generate an SSH key:

```
$ ssh-keygen -t rsa -P “”
```

Add your public key to the list of authorized keys:

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Next, we are going to install Hadoop. Rather than copy this exact command, you should go to Hadoop’s website (https://hadoop.apache.org/) and downloaded the latest stable release (we’ll be using 2.6.4 in this book, but if there is a later 2.6.* version by the time you read this, use that, but do not use 2.7, as there may be too many changes between 2.6 and 2.7).

```
$ wget http://apache.mirror.gtcomm.net/hadoop/common/hadoop-2.6.4/hadoop-
2.6.4.tar.gz
```

Note that that’s the binary, not source.

Decompress the file:

```
$ tar xvzf hadoop-2.6.4.tar.gz
```

Add hduser to the sudoers list by first opening a new command line window (so that you’re no longer hduser but your main user with root access), then:

```
$ sudo adduser hduser sudo
```

Now switch users again back to hduser:

```
$ sudo su hduser
```

Move the files to /usr/local/hadoop:

```
$ sudo mkdir /usr/local/hadoop
$ sudo mv hadoop-2.6.4/* /usr/local/hadoop
$ sudo chown -R hduser:hadoop /usr/local/hadoop
```

Next, the most annoying step: modifying configuration files. One wonders why these things weren't set by default.

First, determine where your Java installation is located (this will be your JAVA_HOME environment variable):

```
$ update-alternatives —config java
There is only one alternative in link group java (providing /usr/bin/java):
/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java
Nothing to configure.
```

Copy just the first portion of the path (up to just before /jre).

Next, add the following lines to ~/.bashrc

```
$ vi ~/.bashrc
```

```
#HADOOP VARIABLES START
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/native
export HADOOP_OPTS=”-Djava.library.path=$HADOOP_INSTALL/lib”
#HADOOP VARIABLES END
```

Then run:

```
$ source ~/.bashrc
```

Next, let’s add the following line to hadoop-env.sh:

```
$ vi /usr/local/hadoop/etc/hadoop/hadoop-env.sh
```

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Next, we’re going to create a tmp directory for Hadoop to use:

```
$ sudo mkdir -p /app/hadoop/tmp
$ sudo chown hduser:hadoop /app/hadoop/tmp
```

Next, add the following to core-site.xml in between the <configuration> tags.

```
$ vi /usr/local/hadoop/etc/hadoop/core-site.xml
```

```
<configuration>
 <property>
  <name>hadoop.tmp.dir</name>
  <value>/app/hadoop/tmp</value>
  <description>A base for other temporary directories.</description>
 </property>
 <property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:54310</value>
  <description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation.  The
  uri’s scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class.  The uri’s authority is used to
  determine the host, port, etc. for a filesystem.</description>
 </property>
</configuration>
```

Next, we’ll need to make a copy of mapred-site.xml.template and add some configuration information to it.

First make the copy:

```
$ cp /usr/local/hadoop/etc/hadoop/mapred-site.xml.template
/usr/local/hadoop/etc/hadoop/mapred-site.xml
```

Next, add the following configuration information:

```
<configuration>
 <property>
  <name>mapred.job.tracker</name>
  <value>localhost:54311</value>
  <description>The host and port that the MapReduce job tracker runs
  at.  If “local”, then jobs are run in-process as a single map
  and reduce task.
  </description>
 </property>
</configuration>
```

Finally, we need to modify hdfs-site.xml. Before we do that, we’ll need to specify which directories will be used as the namenode and datanode. These are different nodes in the Hadoop cluster, but don’t worry about what they do right now.

```
$ sudo mkdir -p /usr/local/hadoop_store/hdfs/namenode
$ sudo mkdir -p /usr/local/hadoop_store/hdfs/datanode
$ sudo chown -R hduser:hadoop /usr/local/hadoop_store
```

Now that the directories exist, you can edit hdfs-site.xml:

```
$ vi /usr/local/hadoop/etc/hadoop/hdfs-site.xml
```

```
<configuration>
 <property>
  <name>dfs.replication</name>
  <value>1</value>
  <description>Default block replication.
  The actual number of replications can be specified when the file is created.
  The default is used if replication is not specified in create time.
  </description>
 </property>
 <property>
   <name>dfs.namenode.name.dir</name>
   <value>file:/usr/local/hadoop_store/hdfs/namenode</value>
 </property>
 <property>
   <name>dfs.datanode.data.dir</name>
   <value>file:/usr/local/hadoop_store/hdfs/datanode</value>
 </property>
</configuration>
```

Next, we need to format the Hadoop file system:

```
$ hadoop namenode -format
```

And finally, let’s start Hadoop:

```
$ cd /usr/local/hadoop/sbin
$ sudo su hduser
$ start-all.sh
```

You can check if Hadoop is running by entering the command:

```
$ jps
10934 NameNode
11481 NodeManager
11374 ResourceManager
13550 Jps
11230 SecondaryNameNode
11039 DataNode
```

It should print out each node’s name and its process ID.

Each of the nodes has a web interface. For example, you can open this URL in the browser:

http://localhost:50070/

To view the NameNode’s web interface.

Thankfully, you’ll never have to do this again.

Chapter 3: Hadoop Streaming

As promised, this chapter will focus on actually running MapReduce jobs on Hadoop.

We turn to a technology called “Hadoop Streaming”, which allows us to run *any* executable within the Hadoop framework.

This is especially powerful because it does not limit us to any particular programming language. For instance, if you really want to use sci-kit learn, which is a Python package, you can, provided that you install it on every machine that will be running sci-kit learn code.

Why use Java with a ton of cruft when you can write elegant and concise Python code?

Let us return to our word count example first. You’ll see that it’s almost exactly like the word count code we previously wrote. That’s the beauty of Hadoop Streaming.

Inside mapper.py

```
#!/usr/bin/env python


import sys
for line in sys.stdin:
  line = line.strip()
  words = line.split()
  for word in words:
    print ‘%s\t%s’ % (word, 1)
```

Inside reducer.py

```
#!/usr/bin/env python


import sys


current_word = None
```

```
current_count = 0
word = None

for line in sys.stdin:
  line = line.strip()

  # parse the input we got from mapper.py
  word, count = line.split('\t', 1)

  # convert count (currently a string) to int
  try:
    count = int(count)
  except ValueError:
    # count was not a number, so ignore this line
    continue

  # this works because all the data is sorted first
  if current_word == word:
    current_count += count
  else:
    if current_word:
      print '%s\t%s' % (current_word, current_count)
    current_count = count
    current_word = word

# print the last word
if current_word == word:
  print '%s\t%s' % (current_word, current_count)
```

Note that both these files start with the #! symbol. That means they are executable, i.e. we can run ./mapper.py instead of python mapper.py.

You’ll have to tell the file system they are executable by running:

```
$ chmod +x mapper.py
$ chmod +x reducer.py
```

How do we actually run this code? Simple! With the command:

```
$ hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.6.4.jar -file
/home/hduser/wordcount/mapper.py -mapper /home/hduser/wordcount/mapper.py -file
/home/hduser/wordcount/reducer.py -reducer /home/hduser/wordcount/reducer.py -input
/small.txt -output /output
```

That’s pretty ugly, so if we were to break it up into several lines we would get:

```
$ hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.6.4.jar \
-file /home/hduser/wordcount/mapper.py \
-mapper /home/hduser/wordcount/mapper.py \
-file /home/hduser/wordcount/reducer.py \
-reducer /home/hduser/wordcount/reducer.py \
-input /small.txt \
-output /output
```

Which is still pretty ugly.

Alright, so there’s some crazy stuff going on here. First, Hadoop Streaming is itself a Java program. That’s where the first part “hadoop jar ..hadoop-streaming*.jar” comes from.

Next, we have to deliver files to the MapReduce job by using the -file option. We’re passing it the mapper and reducer above. Note that I’m assuming they live in hduser’s home folder inside a subfolder called wordcount.

Next, we need to tell Hadoop Streaming which is the mapper executable and which is the reducer executable. Those are the options -mapper and -reducer above.

Finally, we need to tell the job where to find the input and dump the output. That’s “-input /small.txt” and “-output /output”.

Note: This does NOT mean my input data is at / on my local file system!

Nor does it mean the result of the job will be dumped to a file called output at / on my local file system.

This is where HDFS comes into play.

For us, the Hadoop file system will live on the local file system. In production environments, one usually runs a cluster of servers that act as the Hadoop file system. You can also retrieve and store files on scalable data stores like Amazon S3.

This brings us to the next question - how do we get our data into HDFS in the first place?

Once we do that, we can run the job.

Using HDFS

HDFS commands are accessed through the “hadoop fs” prefix.

For example, you can list the files in the root directory:

```
$ hadoop fs -ls
```

There should be nothing there at the moment.

You can create directories:

```
$ hadoop fs -mkdir /mynewdir
```

Note that with HDFS, a directory is also kind of like a file. The output of our job will be saved to /output as specified, but that's actually a directory that contains a number of files.

The number of files corresponds to the number of reducers. So you can imagine running a bunch of reducers in parallel, and each is just dumping their output to some directory in HDFS. Parallelism!

Let's copy small.txt (from the class repo) onto HDFS. That's just:

```
$ hadoop fs -copyFromLocal small.txt /small.txt
```

(Assuming you're in the same folder as small.txt)

You can display the output of a file using:

```
$ hadoop fs -cat /small.txt
```

Once you've done that, you can run the job using the command above.

Once it's completed, you can verify the results:

```
$ hadoop fs -cat /output/*
```

Suppose you make a mistake and you try to run the job again. Hadoop may complain that the directory /output already exists.

To delete it, you can run the command:

```
$ hadoop fs -rm -r /output
```

Database Joins using MapReduce

Word counting is pretty easy. Here’s something people don’t think about often - how does a database perform a join?

Let’s suppose we have 2 tables, users, which contains user IDs and names, and fruits, which contains user IDs and that user’s favorite fruit.

We want to join these tables so that we know a user’s favorite fruit by their name.

users:

user_id	name
1	Alice
2	Bob
3	Carol

fruits:

user_id	fruit
3	apple
2	orange
1	pear

The result of joining these 2 tables would be:

user_id	name	fruit
1	Alice	pear
2	Bob	orange

3	| Carol | apple

In pseudocode, you might get something like this (note that we only want to keep user IDs that exist in both tables):

```
def join(t1, t2, k):
  output = {}
  for row in t1:
    user_id = row[k]
    if user_id in t2:
      output[k] = [row, t2.get_row(user_id)]
  return output
```

Not super difficult to envision the basic algorithm. How can we achieve this with MapReduce?

Remember, there is no notion of “tables”, although the input data may be formatted with comma-separated values or tab-separated values, which is essentially a table.

But then, we don’t know which table we’re looking at, because MapReduce just reads in each file line by line. It doesn’t know which file it is reading.

In that case, we can just format our input as JSON, e.g. a line for the users table could be represented like:

```
{“user_id”: 1, “name”: “Alice}
```

And a line for the fruits table could be represented like:

```
{“user_id”: 1, “fruit”: “pear”}
```

In this case, MapReduce already does all the work for us! Remember, it sorts everything by key before we reach the reducer. Thus, if we make user_id the key, then by the time we get to the reducer, we'll have all the information for that key simultaneously.

It would be easy to switch from inner join to outer join to left join or right join, because we'd just need to check which fields exist for that key.

The code for this example is in the class repo in the folder "join". Let's look at mapper.py first:

```
#!/usr/bin/env python
import sys
import json

for line in sys.stdin:
  line = line.strip()
  j = json.loads(line)
  print “%s\t%s” % (j[‘user_id’], line)
```

Note that this redundantly adds the user ID and then reprints the entire line. Not the most efficient, but it's okay for learning purposes.

Now let's look at reducer.py:

```
#!/usr/bin/env python
import sys
import json

def print_user(j):
  if ‘user_id’ not in j:
    return
  print “%s\t%s\t%s” % (j[‘user_id’], j.get(‘name’, ”), j.get(‘fruit’, ”))
```

```
current_user_id = None
current_data = {}

for line in sys.stdin:
  line = line.strip()

  user_id, j = line.split("\t", 1)

  if current_user_id != user_id:
    # print data for current user and reset
    print_user(current_data)
    current_data = {}
    current_user_id = user_id

  j = json.loads(j)
  for k, v in j.iteritems():
    current_data[k] = v

# print the last user
print_user(current_data)
```

Next, we'll look at how we can do database joins even more easily using Spark.

Debugging Hadoop Streaming Jobs

Sometimes, you end up with bugs in your code and you need to figure out what went wrong before you can proceed.

Unfortunately, Hadoop's error messages are often not helpful.

You can try going to the job tracker URL (another web interface) to see the status of your job and where things went wrong. The URL will be printed to the console when your job runs.

Another method is to test your job without MapReduce. Notice that the code we wrote doesn’t actually depend on the MapReduce framework.

You can simply run the mapper and reducer as follows:

```
cat data.txt | ./mapper.py | sort | ./reducer.py
```

If something goes wrong during this process, you’ll get an error from Python, which is hopefully more helpful.

If this works, but your Hadoop job still doesn’t work, then you know the problem may lie somewhere else.

Chapter 4: Spark

As discussed earlier, Spark is like a better, faster version of MapReduce. We won't get into the details of why or how, since that would require knowing some of the details of MapReduce very intimately.

Rather, we'll appreciate the fact that Spark allows us to write much simpler code, and as a bonus, is way easier to install as well.

How to Install Spark

Earlier versions of my tutorials demonstrated how you could install Spark via the source code (which was essentially just 2 steps). Nowadays, things are even easier as Spark comes pre-built.

Just head over to spark.apache.org/downloads.html and choose a Spark release (latest is 2.0.0 at the time of this writing).

The second option is "Choose a package type" with options like "Pre-built for Hadoop 2.7 and later". You should of course choose "Pre-built for Hadoop 2.6", if you've been following the earlier instructions in this book and that is the Hadoop version you have installed.

Download the .tgz file, untar it, and move the binaries wherever you like. I moved mine to /usr/local/spark to mirror what we did with Hadoop.

You can test out your installation by going to /usr/local/spark and opening the PySpark console:

```
$ cd /usr/local/spark
$ ./bin/pyspark
```

Type in "sc" and hit enter and you will see that a SparkContext object has already been provided for you.

```
> sc
```

You can also load files from the local filesystem! No need to copy to HDFS anymore. Spark has the flexibility to load files from HDFS and S3 as well, if that's where your files are located.

```
> lines = sc.textFile("/home/hduser/wordcount/small.txt")
```

Print everything in the text file (don't worry, it's small, just don't do this on a huge text file):

```
> lines.collect()
```

You can also use .first() to print the first line:

```
> lines.first()
```

Files in Spark are loaded as RDD objects. RDD stands for "resilient distributed dataset". More instance methods are available from the online Spark documentation, but we will simply use them as needed.

The difference between the PySpark shell and the regular Python shell (other than the fact that it has the SparkContext which can open huge distributed files) is that when you run commands on the data, it knows it has the power to run computations on an entire cluster of machines.

You're in the shell of the master node, which is basically the helm of the ship. Any operation you perform will be distributed across the worker machines.

Now that you're used to playing around with Spark, let's move on to a more substantial example.

Database Joins in Spark

Unlike Hadoop MapReduce, Spark actually allows us to treat data as tables. There is actually a lot of flexibility in how we can choose to represent the data.

We can use Pandas dataframe-like objects. We can use SQL. Or we can just use the familiar plain text format.

Dataframes in Spark are the same as relational SQL tables.

Conveniently, the way that Dataframes/SQL tables are loaded in Spark is via rows of JSON documents, which is what we already have!

Since technically a join is between 2 tables, I have split the input data into names.txt and fruits.txt.

Assuming we are inside the PySpark shell, we first create a Spark session:

```
> from pyspark.sql import SparkSession
> spark = SparkSession.builder.getOrCreate()
```

Next, we load the data:

```
> names = spark.read.json("/path/to/names.txt")
> fruits = spark.read.json("/path/to/fruits.txt")
```

Keep in mind that the current user must have access to these files. So if you're hduser, make sure hduser has access to the files. Spark most likely won't give you a very descriptive error if you get it wrong.

Dataframes have functions similar to R/Pandas dataframes. So you can do things like:

```
> names.show()
```

And that will print all the data inside the dataframe.

To do the join, simply call the join function:

```
> joined = names.join(fruits, names.user_id == fruits.user_id)
> joined.show()
+——+—-+—+—-+
| name|user_id| fruit|user_id|
+——+—-+—+—-+
|Alice|      1|  pear|      1|
|  Bob|      2|orange|      2|
|Carol|      3| apple|      3|
+——+—-+—+—-+
```

Let’s discuss some other interesting functions you can call on a Spark dataframe that will help you with your work. These are the basic building blocks that can be manipulated to create bigger, more powerful programs.

You can print the schema of a table:

```
> names.printSchema()
root
 |— name: string (nullable = true)
 |— user_id: long (nullable = true)
```

You can select only the name column:

```
> names.select(“name”).show()
+——+
| name|
+——+
|Alice|
|  Bob|
|Carol|
+——+
```

This would be equivalent to doing:

```
> names.registerTempTable(“names”)
> result = spark.sql(“SELECT name FROM names”)
> result.show()
+——+
| name|
+——+
|Alice|
|  Bob|
|Carol|
+——+
```

You can filter a table by some criteria:

```
> names.filter(names[‘user_id’] == 1).show()
+——+—-+
| name|user_id|
+——+—-+
|Alice|      1|
+——+—-+
```

As an exercise, try to do the equivalent operation using the SQL WHERE keyword.

You can of course count the rows in a table:

```
> names.count()
3
```

Now of course, in a production environment you’re not going to be running jobs by hand in the PySpark shell. You’ll want to be able to automate the process by running a script.

Python scripts do not automatically come with a SparkContext, so you'll need to create one.

Running the script is also not simply "python myscript.py" - you need to tell Spark where your master node is so that it can distribute the work across the cluster.

In this book, we're working with a local single-node cluster, but my course on SQL and Spark shows you how to setup and run a distributed Spark script on AWS (Amazon Web Services). The course is available at: https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

So what will our script look like? This is spark_join.py, also in the course repo:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

sc = SparkContext("local", "My Simple App")
spark = SparkSession.builder.getOrCreate()

names = spark.read.json("/home/hduser/join/names.txt")
fruits = spark.read.json("/home/hduser/join/fruits.txt")

joined = names.join(fruits, names.user_id == fruits.user_id)
joined.show()
```

You can run the program using the command:

```
/usr/local/spark/bin/spark-submit —master spark://localhost:7077 spark_join.py
```

We use the spark-submit script to submit the job to the machine specified by the —master option. Much simpler than Hadoop MapReduce!

In the next chapter, we will get a little more ambitious with Spark and show how you can make predictions and learn the structure of your data using machine learning.

Chapter 5: Machine Learning with Spark

Now that you know how to run a basic Spark job, you can combine that knowledge with your existing programming skills and write anything your heart desires.

One major component in the data science world is machine learning. Here is where things can get messy.

Machine learning algorithms are much more complicated than counting words and doing database joins. I’ve created entire courses based on just a single machine learning algorithm.

There is lots of math involved. If you think you’re going to be a data science big shot but you hate calculus and linear algebra, you’re in for a big surprise.

On top of all that, you need to then re-structure that algorithm so that the work can be distributed across multiple machines (which is not always possible) - well that takes an expert with years of experience in both machine learning *and* big data.

Realize that there are people out there where it’s their full-time job to write such things. They work in big teams and have entire communities dedicated to testing their code.

Thus, this chapter isn’t about how to write machine learning algorithms in a distributed manner, although doing so would be an admirable goal.

This chapter will focus on how to use machine learning algorithms that have already been written for Spark, in a library called MLlib.

This chapter is what you would get if you combined the earlier chapters of this book and the main (highest abstraction-level) ideas from my machine learning courses.

We know that the data is going to exist as a matrix of inputs and targets. We know that we want to train a model on some training data and then test it on some test data.

The real question here is - what’s the right format for MLlib? How do we use Spark to convert it from a regular CSV that you'd use with Sci-Kit Learn into something that MLlib can read? What will the code look like?

Let's get right into it with an example.

Supervised Learning with Deep Neural Network

We are going to be using the MNIST dataset, which is available from Kaggle in CSV format and from https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html in "libsvm" format.

Oddly, Spark does not automatically turn CSVs into tables (small win for Hadoop MapReduce). So if you wanted to use CSV you would need to load it in as a text file. In any case, you can't just load a table and run your machine learning algorithm on it. How will it know which columns are the inputs and which are the targets? Rather, we work with objects of type LabeledPoint, which contains both the input vector and corresponding label.

We are going to end up using the "libsvm" format, but just in case you need to work with CSVs in the future, this is how you would do it:

```
from pyspark.mllib.regression import LabeledPoint

def parse_line(line):
  values = [float(x) for x in line.split(' ')]
  return LabeledPoint(values[0], values[1:])

data = sc.textFile("rawdata.csv")
parsed = data.map(parse_line)
```

This would give you a RDD of objects of type LabeledPoint. You could then use a classifier in MLlib to train on the parsed data.

Now here are some weird facts about MLlib, and demonstrate its immaturity at this stage (although it is growing fast!).

Some classifiers train on RDDs with LabeledPoints.

Some classifiers train on DataFrames loaded via libsvm format.

Some classifiers can only do binary classification.

Some classifiers can do multiclass classification.

Some classifiers can do multiclass classification, but only in Scala and Java and not Python.

It’s truly a mess.

The real story is, Spark 2.x is moving toward a Dataframe-based machine learning API. It will continue to support RDDs in MLlib, as more features are added to the new API, since the new API does not yet replicate everything the old API can do with RDDs.

The Spark team expects to remove the RDD-based API from MLlib by Spark 3.0.

Since the Multilayer Perceptron in MLlib works on DataFrames, we’ll load the MNIST dataset in libsvm format, for which there are functions readily available without needing to do much work.

Oddly, the data is of dimensionality 780, rather than the usual 784=28x28. We will take that into account when building our model.

Let’s get to the code:

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

```
data = spark.read.format("libsvm").load("mnist.scale")

# Split the data into train and test
splits = data.randomSplit([0.6, 0.4], 1234)
train, test = splits

trainer = MultilayerPerceptronClassifier(maxIter=30, layers=[780, 100, 10],
blockSize=128, seed=1234)

model = trainer.fit(train)

# compute accuracy on the test set
result = model.transform(test)
predictionAndLabels = result.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Accuracy: " + str(evaluator.evaluate(predictionAndLabels)))
```

To run the script:

```
/usr/local/spark/bin/spark-submit —master spark://localhost:7077 mlp.py
```

You should get an accuracy of about 92% after 30 epochs. The relevant file in the codebase is mnist/mlp.py.

Unsupervised Learning with K-Means Clustering

Now that we've done supervised learning, let's do some unsupervised learning. We are going to use the CSV data this time, which you can get from https://www.kaggle.com/c/digit-recognizer.

Because we can't exactly visualize 784 dimensions, we are going to print the "within set sum of squared error" to tell us how good our clustering is.

“Where are the dimensionality reduction algorithms?”, you might ask. PCA and SVD are available in Scala and Java, but sadly not in Python yet.

After running this script (mnist/kmeans.py) you should achieve a WSSSE of about 6.65 x 10e7.

The command to run the script is of course:

```
/usr/local/spark/bin/spark-submit —master spark://localhost:7077 kmeans.py
```

The contents of kmeans.py are as follows:

```
import numpy as np
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans, KMeansModel

sc = SparkContext(“local”, “My Simple App”)

data = sc.textFile(“/home/macuser/train.csv”)

# skip header
data = data.filter(lambda line: line[0] != ‘l’)

parsed = data.map(lambda line: np.array([float(x) for x in line.split(‘,’)[1:]]))

clusters = KMeans.train(parsed, 10, maxIterations=10, runs=1,
initializationMode=“random”)

def error(pt):
  center = clusters.centers[clusters.predict(pt)]
  return np.sqrt(sum([x**2 for x in (pt - center)]))
```

```
wssse = parsed.map(lambda pt: error(pt)).reduce(lambda x, y: x + y)
print(“Within set sum of squared error: %s” % wssse)
```

At this point, you can imagine loading a dataset for which you do not know the labels, and then running many jobs with different values of K. The K that gives you the best objective might be the “natural” number of clusters for that data.

Epilogue: Where do we go from here?

In this book, we saw how we could process huge amounts of data by distributing jobs amongst multiple machines.

One issue we didn’t talk about was that these are all batch jobs.

The typical workflow is as follows:

1) Data is gathered into log files throughout the day (or hour, or any other unit of time).

2) A batch job runs on the data for that time period.

3) Results are pushed to a place where they are needed (i.e. reporting application, input into another batch job, etc…).

What if we have huge amounts of data to process in real-time? (i.e. not a batch job)

If you’re Twitter, you want to process tweets as they happen.

People often “live tweet” big events.

How do these tweets go from one user’s phone, to Twitter’s worldwide distributed database, to every other user’s Twitter feed almost instantly?

In a future book, we will discuss the limitations of batch jobs and how to build scalable services that work in real-time.

Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

I do 1:1 coaching and consulting as well.

Do you want to learn more about deep learning? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

My first course in deep learning is a lot like the book, but you get to see me derive the formulas and write the code live. There are also a ton more examples, like how to use deep learning to optimize your e-commerce store and facial expression recognition.

Data Science: Deep Learning in Python

https://udemy.com/data-science-deep-learning-in-python

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to the book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

Data Science: Practical Deep Learning in Theano and TensorFlow

https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

When you've got the basics of deep learning down, you're ready to explore alternative architectures. One very popular alternative is the convolutional neural network, created specifically for image classification. These have promising applications in medical imaging, self-driving vehicles, and more. In this course, I show you how to build convolutional nets in Theano and TensorFlow.

Deep Learning: Convolutional Neural Networks in Python

https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow

In part 4 of my deep learning series, I take you through unsupervised deep learning methods. We study principal components analysis (PCA), t-SNE (jointly developed by the godfather of deep learning, Geoffrey Hinton), deep autoencoders, and restricted Boltzmann machines (RBMs). I demonstrate how unsupervised pretraining on a deep network with autoencoders and RBMs can improve supervised learning performance.

Unsupervised Deep Learning in Python

https://www.udemy.com/unsupervised-deep-learning-in-python

Would you like an introduction to the basic building block of neural networks - logistic regression? In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in the first deep learning course too challenging.

Data Science: Logistic Regression in Python

https://udemy.com/data-science-logistic-regression-in-python

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on Natural Language Processing, or NLP:

Data Science: Natural Language Processing in Python

https://www.udemy.com/data-science-natural-language-processing-in-python

Are you interested in learning SQL - structured query language - a language that can be applied to databases as small as the ones sitting on your iPhone, to databases as large as the ones that span multiple continents - and not only learn the mechanics of the language but know how to apply it to real-world data analytics and marketing problems? I even show you how to write SQL code in Spark, so you can apply all the skills you learned in my big data book! Check out the course here:

SQL for Marketers: Dominate data analytics, data science, and big data

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

Are you interested in stock prediction, time series, and sequences in general? My Hidden Markov Models course is where you want to be. I teach you not only all the classical theory of HMMs, but I also show you how to write them in Theano using gradient descent! This is great practice for writing deep learning models and it will prepare you well for its sequel, Deep Learning Part 5: Recurrent Neural Networks in Python. You can get the HMM course here:

Unsupervised Machine Learning: Hidden Markov Models in Python

https://udemy.com/unsupervised-machine-learning-hidden-markov-models-in-python

Recurrent Neural Networks also focus on time series but are much more powerful than Hidden Markov Models because they do not rely on the Markov assumption and do not suffer from certain computational limitations that HMMs do. Learn more about the popular LSTM (long short-term memory) unit and GRU (gated recurrent unit) in this course:

Deep Learning: Recurrent Neural Networks in Python

https://udemy.com/deep-learning-recurrent-neural-networks-in-python

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-part intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don’t forget to hit “like”!)