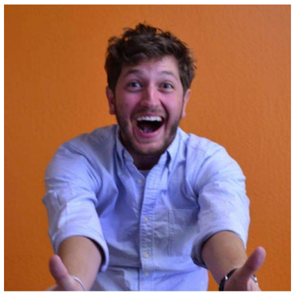
Intro to Data Science

Lesson 5 Notes

UDACITY

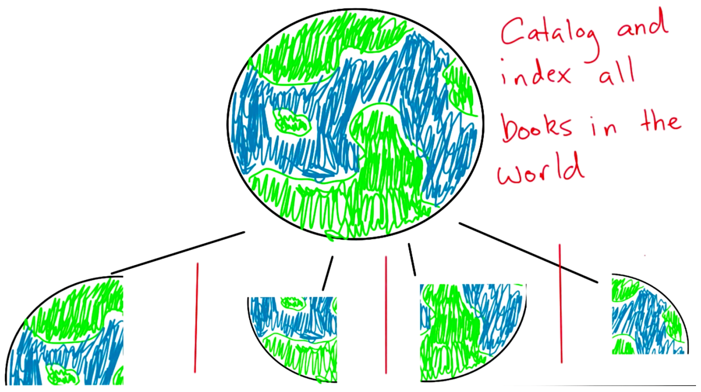
Map Reduce

Welcome to lesson 5



Welcome to lesson five. This may be shocking to hear, but thus far, all of the data that we've used in this course has been relatively manageable. What do I mean by that? Well, we can fit it onto one machine pretty easily, we can load it into memory and we can run all of our analysis in a serial fashion. But imagine that we are working with some very large data set. I'm not talking hundreds of gigabytes here, I mean terabytes, or maybe even petabytes. At this point, it would make sense to introduce something like the MapReduce programming model, which allows us to take our data, distribute it across many different machines, and run our computations in parallel. If you spent any amount of time around data science circles, you've probably heard tons of crazy terms like hadoop or hive. Or a MapReduce or distributed file systems. More generically, you've probably heard a lot about big data. During this lesson, we'll discuss how to use the MapReduce programming models to solve some very simple problems. We'll also discuss other big data tools like Hive at a high level. That way, next time when you find yourself at a data science cocktail party, you'll be able to keep up with the conversation. Alright, well why don't we get started then.

Big data and map reduce



Before we get started, let's make sure you're clear on when the MapReduce programming model is appropriate and when it isn't. People use the term big data a lot. But often data isn't actually that big, in the grand scheme of things. Some people might tell you that big data is anything too large to deal with easily in Excel. That's definitely not the case. It's generally safe to say that your data is quote unquote big if it's too large to fit onto one disk. A reasonable size at which to set the lower bound for big data at this point in time is probably about five terabytes. For example, say Google was trying to catalog and index all of the books in the world to find out which words have appeared most often. It'd be impossible to load the text from all the books in the world into a single disk. It's simply too much data. This is when we should look to the MapReduce programming model. Beyond this first constraint, because of the specifics of the MapReduce programming model, MapReduce only works for tasks where you hope to employ many workers simultaneously who do not have knowledge of each other's actions. Why is that? MapReduce splits a large job up into several smaller chunks that each fit onto one machine and occurs simultaneously. These machines do not communicate with each other while performing their computations. Certain tasks, that would be very easy to do using a SQL-like database or simple Python script, can become very complex when attempting to do them using MapReduce. Because of this, the smartest thing to do is to only use MapReduce when your data is truly big.

Quiz: Scenarios for MapReduce

You might be wondering, what are some of the ways that the MapReduce programming model, can be used to solve interesting problems? Well, in which of the situations below do you think map produce may have been used? Discover new oil reserves? Power an e-commerce website? Identify malware and cyber attack patterns for online security? Help doctors answer questions about patients' health?

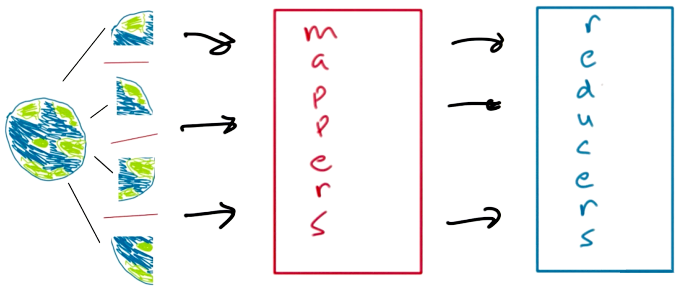
In which of the situations below do you think mapreduce may have been used?

[x] Discover new oil resources
[x] Power an e-commerce website
[x] Identify malware and cyber attack patterns for online security
[x] Help doctors answer questions about patients health

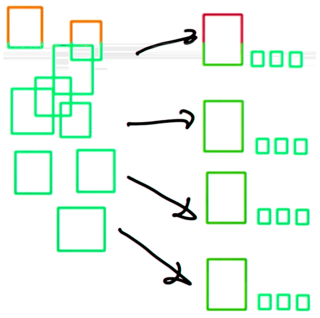
Answer:

It turns out that all four of these problems can and have been tackled using mapreduce. Chevron utilizes the mapreduce programming model to sort and process data from ships that search the ocean for seismic activity indicative of oil reserves. Tons of online commerce sites, eBay for one, use mapreduce to manage their huge amounts of data on sellers, buyers, and transactions. Some companies use mapreduce to process data to identify malware and cyber attack patterns. A company called IPTrust for example used mapreduce to assign trustability scores to IP addresses. Finally products developed by a company called Apixio leverage mapreduce to analyze large amounts of data and help healthcare professionals answer questions about their patients health. As you can see mapreduce can be applied to a solve a wide range of problems. I'm sure you're anxious to use mapreduce to solve our problems so let's discuss how the mapreduce programming model actually works.

Basics of Map Reduce



Okay. So, say I truly have more than five terabytes of data. Seems like MapReduce is the best tool for the job. How exactly does MapReduce work? MapReduce is actually just a parallel programming model for processing large data sets across a cluster of computers. The most important thing to understand about MapReduce at a high level is that computation is done via two functions. The mapper and the reducer. So, in the most general sense, we start out with a collection of documents or records. If we were indexing all of the books in the history of the world. As we discussed before maybe each book is a separate document. We send these documents in a distributed way to many mappers. Which each perform the same mapping on their respective documents and produce a series of intermediate key value pairs. We then shuffle these intermediate results and send all key value pairs of the same key to the same reducer for processing. We do this so that each reducer can produce one final key value pair for each key. If we didn't send all values corresponding to a given key to the same reducer, we would end up with many final, key value pairs for each key. Which is not desirable.



Programming Quiz: Counting words

Here is one way to explain the Mapreduce programming model. Say that I wanted to count the number of occurrences of each word that appears at least once in a document. Let's use the text of Alice in Wonderland. Here's a bit of text that says Alice was beginning to get very tired of sitting by her sister on the bank And of having nothing to do. If I wanted to solve this problem without Mapreduce, I might create a Python dictionary consisting of all the words and their counts. I could go through the document and say, for each word in the document, if there is a key for that word, add one. Otherwise, set the initial for that key equal to one. And instead of applying it to this short sentence fragment from the book, we'd apply it to the entire book. Before we solve this problem with Mapreduce, why don't you try to write a Python script along the lines of what we just discussed, that will get the job done. Given many lines of a text, create a dictionary with a key for each word, and a value corresponding to the count of the word in that text. Note that we want the words to be stripped of any capitalization and punctuation. We just want the basic words. Here's some code to get you started. First, we import system string. And then we initialize an empty dictionary, which will hold our words and values. We cycle through the lines of the input, and for each line we create an array, data. Which is essentially all of the words in that line, split by white space. So if we started with this line. Hello, how are you? It would become, hello, how, are, and you, in an array of length four. Your code should go here. After we split the line by white space, and before we print out the dictionary.

Answer:

Alright, so let's walk through this solution line by line. We first initialize an empty array, word counts. We then cycle through all of the lines in the document. For each line in the document we create an array, data. Data is the line stripped of any surrounding white space and tokenized based on the white space. For each string in data, we create a new key. And we remove any punctuation from the word and make sure that the word is lower cased. Then, we check if the key exists as a key in word counts. If it does, we add 1 to the value for that key. If not, we initialize that key in the word counts dictionary and we set it equal to 1. When we've cycled through all of the lines And all of the keys in each line. We print out the dictionary word_counts. Which should have a count of each word, in our document.

Counting words in Map Reduce part 1

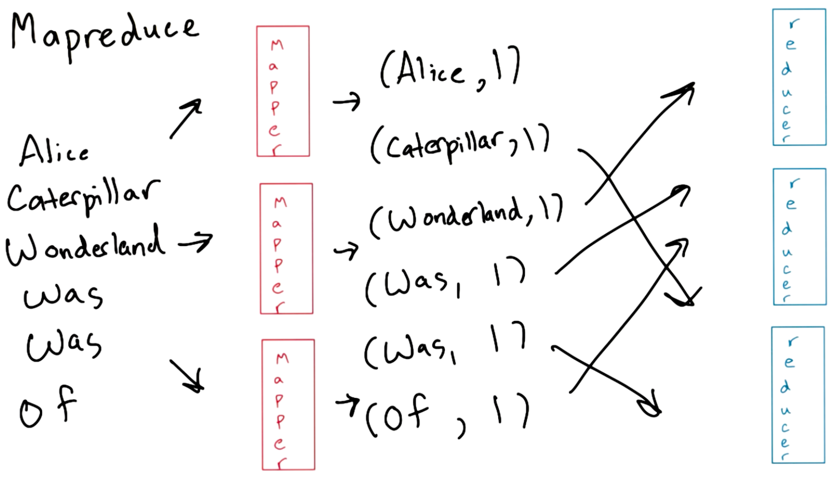
You can imagine that if a text that we wanted to analyze was really large, rather than a short book like Alice in Wonderland, maybe a document containing all of the books ever written. This script would take an extremely long time. It might even be impossible to run the scripts, due to the inability to fit all of that data onto one machine. In this case, it makes sense to employ MapReduce. How would I solve this problem using MapReduce? You can imagine that if the text we wanted to analyze was really large, rather than a short book like Alice in Wonderland. Maybe a document containing all of the books ever written, this script would take an extremely long time. It might even be impossible to run the scripts due to the inability to fit all of that data onto one machine. In this case, it makes sense to employ MapReduce. So how would I solve this problem using MapReduce?

Counting words in Map Reduce part 2

Let's break through the word count example by writing a mapper and reducer in Python. Recall that the mapper will take in a document. In this case, a collection of words that appear in Alice in Wonderland, and will return various intermediate key value pairs. In this case, each word in the value one. These key value pairs will then be shuffled to ensure that every key value pair with the same key ends up on the same reducer. And each reducer will perform some operation on all of the values corresponding to a particular
key. In order to produce one

final key value pair. In this particular case, we're counting up how many times each word occurred. So for Alice, Wonderland, Caterpillar, and Of, the final key value pair is Alice 1, Wonderland 1, Caterpillar 1 and Of, 1. Whereas was appeared twice, so the final key value pair there is Was 2.

Mapper



The code you see below is a Python implementation of a mapper for our word count problem. Imagine we have a huge document filled with many lines of words. We can send different chunks of the document to a number of different mappers to get the job done. What's this mapper code doing? Let's take a closer look. The mapper takes in a document. For each line of the document, it creates an array consisting of the words in that line of text. We then cycle through the words. We clean each word up so we remove any punctuation and make all the letters lower case. Then for each word we emit a key value pair. The word itself and the number one. Note that the key value pair is separated by a tab. So, if a particular word appears multiple times in the document, we will emit multiple identical intermediate key value pairs. The word and the number one. Over and over again. Let's say we ran this map around the following sentence. Hello, My name is Dave. Dave is my name. The mapper would split the words on the white space like so. The mapper would then emit the following key value pairs. Hello one, my one, name one, and so on. The next step in the map reduced design paradigm, is to shuffle the results produced by the mappers and send all of the key value pairs corresponding to certain keys to certain reducers. So, if we ran the job with just one reducer, that reducer would process all the keys. But, if we ran the job with two reducers, one reducer might process half the keys while the other reducer process the other half. So, let's go back to our word count example.

Reducer

Below is a Python implementation of a reducer for our word count problem. Let's walk through this reducer line by line and discuss what it's doing. First, we set initial values for word_count and old_key to 0 and None respectively. We take in our data which you'll recall will be a bunch of key value pairs separated by a tab. If we have a strange row with more or less than a key and a value, i.e., the length of the data array is not equal to, we'll just continue on to the next row. All of these lines down here are essentially summing up the count for every single key. We continue through all the key value pairs for a particular key until we notice that we're on to a new key. Once this happens, we print out the key and its final word count. We then set word count equal to 0 once again. You'll note that this implementation will not emit a key value pair for our final key. Because of that, we have to include this final bit of code down here. So if old key is not equal to none, let's print out one final key value pair. So, say that we ran this reducer on all of the intermediate key value pairs our mapper in the previous slide produced. We would go through all key value pairs with the key hello. There's only one, so we emit hello, 1. There are however, 2 key value pairs with the key my. So we would emit the final key value pair, my 2 for this key and so on. So now if we use the mapper and reducer we just wrote on a more significant example, say the text of Alice in Wonderland, we can check that it works and ensure that it produces the same type of result as the Python script that you wrote earlier in this lesson. We're going to simulate running this map reduced job by just running these scripts locally, using Python. Note this isn't a real MapReduce deployment, we're only running this on one computer. If we run the script, we see that we have an output where we have all of the words that appear in Alice in Wonderland and the number of times that each of those words occurs. Note that all of the words are lowercase and are stripped of all punctuation.

Map Reduce with Aadhaar Data

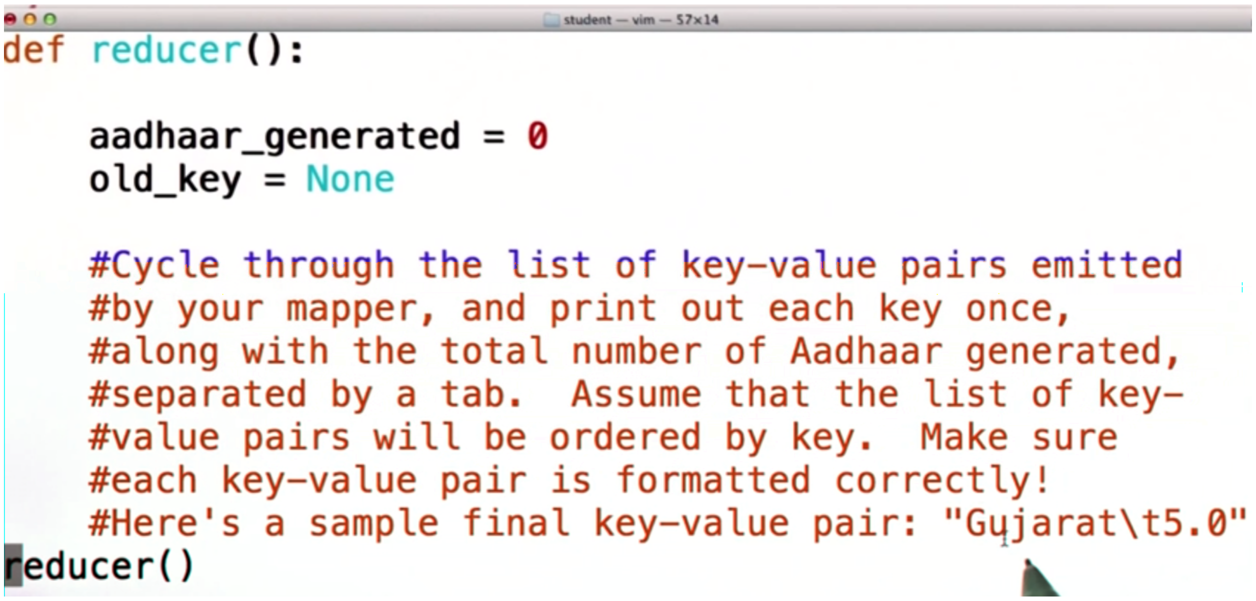
Alright, now we know how to use the Mapreduce programming model to count words. That's great, but counting words is pretty specific and it's not amazingly interesting. Can we use the Mapreduce programming model to do more interesting things? We sure can. Lets revisit the AADHAAR enrollment data from way back in lesson two. Say we want to know how many enrollments happened in each district. We can perform this calculation using the MapReduce programming model. Of course, we saw in lesson two, we could easily perform such a computation with a SQL query. Or a Python script. Using Mapreduce to solve this problem isn't necessary with data our size. But imagine if we had terabytes and terabytes of data. Say, line items for the AADHAAR enrollment of every single Indian citizen. We'd have no choice but to use MapReduce.

Programming Quiz: Mapper and Reducer with Aadhaar Data

Lets look at the CSV file containing our Aadhaar enrollment data once again. Each row has a number of columns such as, registrar, enrollment agency, state, district, Aadhaar generated, enrollment rejected, and a bunch of other information. If we want to count the number of Aadhaar generated per district. The columns that we're most interested in are district and Aadhaar generated. Can you fill in the missing pieces of the mapper? So if you wanted to complete this job using the mapper programming model, we would need to write a mapper and reducer. Why don't you give it a try. Here is the skeleton of a mapper for this job. We go through every single line in the input. In this case, it's going to be our CSV file containing all of the rows and our Aadhar-generated data. You're going to have to go through each line, which will be a list of comma-separated values. The header row will be included. Take a nice each row using the commas and emit a key value pair containing the district and the number of Aadhaar generated separated by a tab. Make sure that each row has the correct number of tokens and make sure it's not the header row. In order to count the number of Aadhaar generated per district using MapReduce, we'll also have to write a reducer. Here's the skeleton of a reducer function that you'll fill in. We initialize aadhaar_generated to 0, and set old_key to None. You'll cycle through the list of key value pairs emitted by your mapper, and print out each key once, along with the total number of Aadhaar generated, separated by a tab. You can assume that the list of key value pairs will be ordered by key. Make sure that each key value pair is formatted correctly before you process it. Here's a sample final key value pair. Gujarat\t5.0.

Answer:

Let's walk through the code you wrote to complete the mapper function. First, we create an array, data, which splits each row of data on the comma. If the length of the data array is not equal to 12, which is what we expect given our aadhaar data, or the first entry is equal to registrar, which would indicate that this is the header row, we continue on to the next row. Otherwise, we print the district and the number of aadhaar generated, separated by a tab. Now let's talk about the reducer function. Recall that our reducer function will consume the key-value pairs admitted by our mapper. So, we create an array, data, for every single line which will essentially be of length two, containing the key and the value. Note that we split on the tab which we input into the output of our mapper. If for some reason data is less than or greater than length two, we continue on. Because there's something wrong with this key-value pair and we shouldn't process it. Next we set this key, and count equal to the key and number of aadhaar generated in data. Recall that reducer receives the key-value pairs sorted by key. So, if this is a new key, let's emit the final key-value pair, the key and the total number of aadhaar generated separated by a tab. Then, let's set aadhaar generated equal to zero. Otherwise let's add the number of aadhaar generated in this particular key-value pair. So the total number of aadhaar generated for this key. And let's continue on to the next value. We include this last if clause for the last key in our data. Because there's no next key after the last key if we didn't have this we would not emit a key-value pair for the final key in our intermediate data. So here, after we've done all this other processing up here, we just say for the last key, hey let's make sure that we emit the key-value pair.



```
def reducer():

    aadhaar_generated = 0
    old_key = None

    #Cycle through the list of key-value pairs emitted
    #by your mapper, and print out each key once,
    #along with the total number of Aadhaar generated,
    #separated by a tab.  Assume that the list of key-
    #value pairs will be ordered by key.  Make sure
    #each key-value pair is formatted correctly!
    #Here's a sample final key-value pair: "Gujarat\t5.0"
reducer()
```

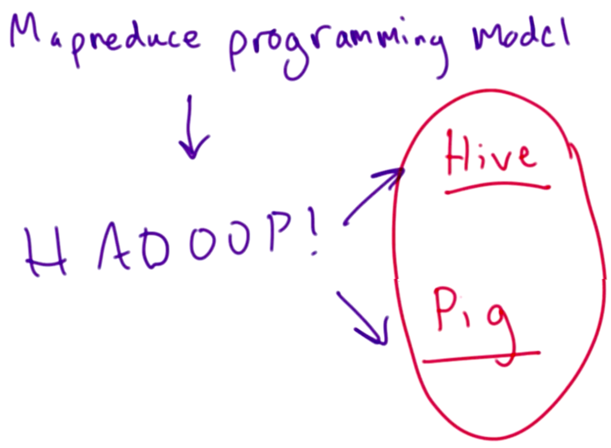
In order to count the number of aadhaar generated per district using MapReduce, we'll also need to write a reducer. Here's the skeleton of a reducer function that you'll fill in. Note that we said aadhaar generated equal to zero, and initialize old key to be equal to none. You'll cycle through the list of key-value pairs emitted by your mapper, and print out each key once, along with the total number of Aadhaar generated, separated by a tab. You can assume that the list of key-value pairs will be ordered by key. Make sure that each key-value pair is formatted correctly. Here's a sample final key-value pair: Gujarat tab 5.0. Feel free to use the reducer that we wrote earlier in this lesson for inspiration.

More complex MapReduce

Alright, so over the course of this lesson we've discussed how to perform some basic computations using the Mapreduce programming model. For example, counting words in a collection of text, or aggregating adhar generated in a particular district. These jobs are easy to write in the Mapreduce programming model, and even so, solving these problems with Mapreduce requires a lot more code than they would if we didn't use Mapreduce. For example, compare our original word count solution using Python dictionaries to our Mapreduce solution. As you can see, the Mapreduce implementation is quite a deal more involved. You can imagine that if we wanted to do something more complicated, say compute the similarity between the number of documents, or implement some machine learning algorithm, things could get pretty hairy, pretty fast.

MapReduce Ecosystem

During this lesson, we focused on the MapReduce programming model. A very common open source implementation of the MapReduce programming model is Hadoop. Hadoop couples the MapReduce programming model with a distributed file system. In order to more easily allow programmers to complete complicated tasks using the processing power of Hadoop, there are many infrastructures out there that either built on top of HADOOP, Or allow data access via Hadoop. Two of the most common are Hive and Pig. Hive was initially developed by FaceBook, and one of its biggest selling points is that it allows you to run map-produced jobs through a SQL like querying language, called the Hive Query Language. Pig was originally developed at Yahoo! And excels in some areas Hive does not. Pig jobs are written in a procedural language called pig Latin. This wins you a bunch of things. Among them are the ability to be more explicit about the execution of your data processing. Which is not possible in a declarative language like SQL syntax. And also the ability to split your data pipeline. Hive and Pig are two of the most common Hadoop-based products, but there are a bunch of them out there. For example Mahout for machine learning, Giraph for graph analysis, and Cassandra, a hybrid of a key value and a column oriented database.



Introducing Joshua

Yeah, so I was physics major in college and then I was physics major in grad school. And I continued on to get my PhD in physics. But what I found during my PhD is that I, I really love doing data analysis, and that my real passion was for sort of trying to understand the world through data. And I wanted to find a job where I could have a big impact and really make a difference.

MapReduce Tools

So all of our data is on Hoodoop, and, we use Pig to reduce it. But there are many similar tools like Hive that are used at other companies, but it's really important to be able to take all the data that, that's far too large to fit on any computer, and be able to aggregate it down or reduce it down. To a point where it's small enough that you can look at, look at it. Like, on a single machine. Or make a sort of, reduced plot of it. So being able to handle the data and pull the data from different sources is very useful.

Pig

So, what's really great about Pig is that it abstracts away from the user the actual implementation of how the calculation is done. So, you can write in a more abstract language what kinds of operations should be performed to your data. You can say, for example, join this data set to this data set, or filter out this data set, or reduce this data. But you're not telling it how to do the analysis. And then Pig can decide the fastest way to do it for you. So instead of having to write very detailed jobs describing exactly what should be done, you can just specify in an abstract sense the calculation that needs to be done, and you can let the computer do the hard work of figuring out the faster way to do the calculation.

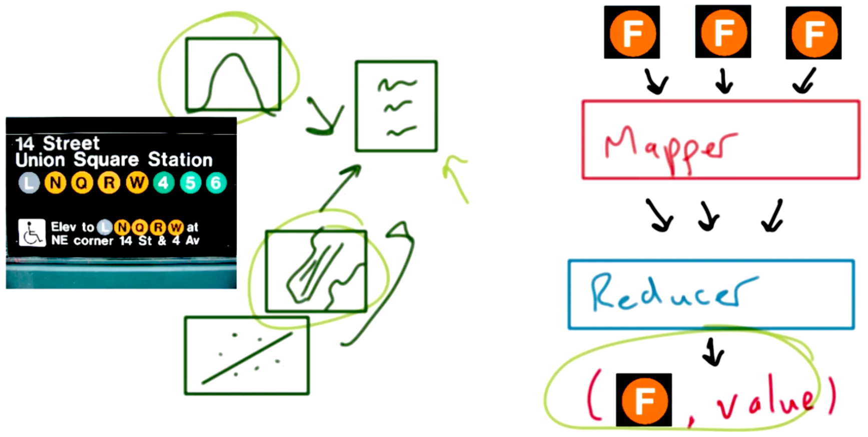
Best Part about Being a Data Scientist

So what I really love about being a data scientist, is I love that I can use data to help these software companies out. And the reason I love that is because at the software companies you're able to help build systems that scaled a hundreds of millions of users, that can have an enormous impact on the world. And I really love that you can bring quantitative reasoning and quantitative thinking to help out all of the users. I love the impact.

Map Reduce with Subway Data

Why don't we apply MapReduce to our class project. This week, we're going to synthesize all of our findings up to this point. The statistical analysis, the data driven predictions, and visualizations into a blog post your friends and family would be able to read if they wanted to learn some interesting facts, about the New York City subway system. It's often helpful to begin a blog post with some interesting, but basic facts about the subjects we will be discussing. You might recall earlier in this course I provided some statistics about the New York City subway, like daily ridership, number of stops, et cetera. I'd like you to collect some statistics about our New York City subway data. More specifically, how many subway riders were there over the course of May 2011, and how many people pass through each station on an average day. Using the MapReduce program on the MTA subway datta. Once you've produced these statistics, write a brief blog post that summarizes the results of your analysis from assignment three, incorporates the visualizations you made for assignment four, and your newly produced MapReduce findings. When this is complete, you should have an interesting and informative blog post about the New York City subway, that you can share with your friends and family.

Lesson 5 Recap



Well, that's it. You've completed Udacity's Introduction to Data Science course. I really hope that you've enjoyed it, and you've learned a lot. Having taken this course, you should now be familiar with a lot of the tools and techniques that data scientists are using on a daily basis. You've also completed an entire analysis project investigating ridership on the New York City subway system. And you can share that with your friends or your family. If you find yourself still interested to learn more about data science, I'd highly recommend you take some of the data science courses being offered by Udacity. They cover all of the topics that we've discussed in this course at a much higher level of depth. If nothing else, I hope that you've learned that data science isn't just something being used in Silicon Valley. More and more, data science is being used to solve problems of all different types in industries all over the world. Well, so long, and thanks for taking the course.