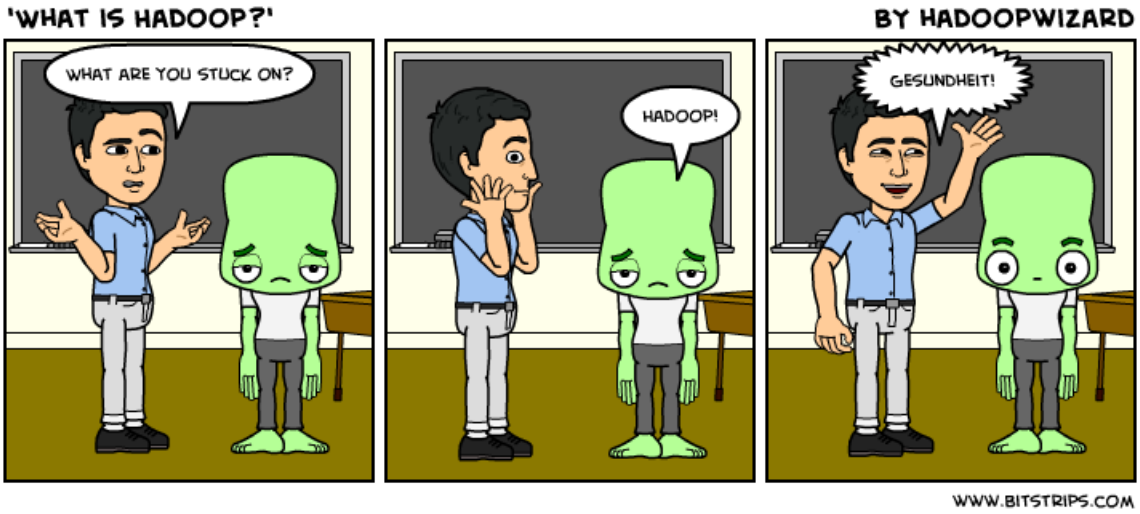
CS158 - Assignment 9

Faster Naive Bayes? Say it ain't so...

Part 1 due: Sunday, April 6 by 11:59pm
Part 2 due: Sunday, April 17 by 11:59pm



http://www.hadoopwizard.com/what-is-hadoop-a-light-hearted-view/

For this assignment we're going to revisit our Naive Bayes classifier and implement a parallelized version using MapReduce for Hadoop.

Programming Environment Setup

To get you ready for the programming portion of the assignment, you'll need to do a few things to get your programming environment setup and to get a version of hadoop setup. *You'll need to do this before you can fully complete either of the assignment parts*, though you can work on some portions of Part 1 in parallel if you get stuck.

Getting Hadoop running

Because setting up and maintaining a hadoop cluster can be challenging, instead, each of you is going to run your own mini-hadoop cluster as a virtual machine. You have two options for where to run hadoop, you can either run it on your laptop or you may use the machines in Edmunds 105.

- Install VirtualBox: If you're running on your laptop, install VirtualBox (`https://www.virtualbox.org/wiki/Downloads`). The machines in 105 already have it installed. VirtualBox is a virtual machine software that will allows you load an image of a machine and run it.

- Download VM image: Download the VM image from:

  `https://cs.pomona.edu/classes/cs158/assignments/assign9/Cloudera-Training-VM-4.1.1.c.vmdk`

  and save it somewhere where you'll remember.

  These images are very large (4+GB), so do feel free to delete it once the assignment is done.

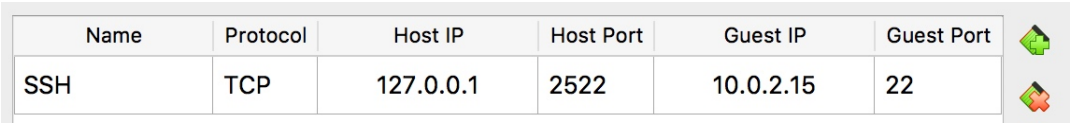
- Setup new VM

  - Start up VirtualBox (if you're on a Mac, you can just press cmd+spacebar and search for it or it should be installed in your Applications folder).
  - Click the "New" button to create a new VM.
  - Give it a name (I called mine "hadoop"), select "Linux" and then "Ubuntu (64-bit)" then continue on.
  - For memory size, enter something in the range of 512MB - 1024MB for memory (though you could do larger if it makes you happy :), then continue on.
  - Select "Use an existing virtual hard drive file" and then click on the little folder in the bottom right to select the vm.
  - In the pop-up window, click the "Add" button in the top left and then navigate to the location where you unzipped the Cloudera VM file and within that directory select the file "Cloudera-Training-VM-4.1.1.c.vmdk" (it should be highlighted).
  - Click "Choose" and then "Create".

    This should finish the installation and you should see a new VM in the VirtualBox manager window that says "Powered Off".

- Setup network communication for the VM

  1. Select the newly created VM in VirtualBox manage.
  2. Select the "Setting" button and then the "Network" tab.
  3. Under Adapter 1, make sure it says "NAT" and then expand the "Advanced" portion of the menu and click "Port Forwarding".
  4. In the upper right corner of the Port Forwarding menu there's a green diamond with a plus sign. Click this to add a port forwarding rule and then fill in the following information:



Name	Protocol	Host IP	Host Port	Guest IP	Guest Port
SSH	TCP	127.0.0.1	2522	10.0.2.15	22

5. Click "Ok" to close the Port Forwarding menu and "Ok" again to close the general setting menu.

You should now be able to select your VM image and click the "Start" button and it will boot the VM.

Transferring files to the VM

Once the VM boots, you should be able to interact with it like I've done in class. One thing you will want to do is to transfer files back and forth between your computer and the VM. When the VM is running, you can interact with it like any other linux server (i.e. `ssh` into it, `scp` files back and forth, etc.). The only catch is that the VM server doesn't have a name. However, we've set it up that any time we try and communicate with IP address `127.0.0.1` on port `2522`, it will connect with the VMs port 22, which is the port used for `ssh/scp`.

For example, if you want to copy the file `some_file.txt` to the virtual machine you would type:

```
$ scp -P 2522 some_file.txt training@127.0.0.1:file_name_on_the_vm.txt
```

When prompted for a password, enter "training". If you're rusty on `scp` lookup some examples online. Additionally, I recommend `ssh`ing into the machine to interact with it rather than try and do it via the VirtualBox window. Once the server is running, you can login to the server using `ssh`:

```
$ ssh -p 2522 training@127.0.0.1
```

again, with the password "training". This has the advantage of using your native terminal to interact with the server rather than through VirtualBox. The biggest benefit of this is that copy and paste will work, though I also find it to be more convenient in general.

Assuming this works, you have your hadoop cluster up and running. If you want, you can login to the VM and try out some of the `hdfs dfs` commands like we did in class to move files from the VM to the hdfs. For example, if you're sshed into the VM and you have a file called `some_file.txt` (i.e., you've scped that file to the VM), you can copy it to your home directory on the hdfs using:

```
$ hdfs dfs -put some_file.txt
```

This will just copy the file into hdfs in "your" home directory. You can see the contents of that file (from the hdfs) by typing:

```
$ hdfs dfs -cat some_file.txt
```

Java setup

There are two things that need to be configured to compile mapreduce programs. First, there are some .jar files that need to be included in the classpath. Second, *the VM has Java 1.6 installed* (sorry, no fancy stuff for this assignment), so we need to make sure that we compile our code to be compatible with this.

For this handout, I'm going to give instructions for how to accomplish these things for Eclipse, however, it is possible for other IDEs and also at the command-line. You can look online for how to do it or we can try and figure it out together.

The jar files that you need to include can be found at:

`http://www.cs.pomona.edu/classes/cs158/assignments/assign9/hadoop_libraries.zip`

Unzip this file and put the .jar files somewhere on your computer.

To setup a new project in Eclipse:

- Open Eclipse and create a new Java project. During creation:
  - Check "Create separate folders for source and class files".
  - Click "Next" and in the second screen, select the "Libraries" tab.
  - Click the "Add External Jars" button and then navigate to the location of the hadoop libraries folder and add *all* of the `.jar` files.
  - Click "Finish".
- To change the compiler version:
  - Right-click on the project and select "Properties".
  - Select "Java Compiler".
  - Deselect the button "Use compliance from execution environment..." and then select 1.6 from the "Compiler compliance level".

  Note this is still using whatever Java compiler you have installed (e.g., 1.8), but will compile the code as if it were 1.6 code.
- To test that this works:
  - Create a new package called "demos".
  - Create a new Java class called `WordCount`.
  - Go to the course webpage and find the `WordCount` demo from the class notes and copy and paste the contents into your `WordCount` class. If everything is setup correctly, this file should compile.

    `http://www.cs.pomona.edu/classes/cs158/lectures/hadoop_examples/`

Part 1: Hadoop Basics and MapReduce on Paper

For the first part of this assignment, you will just be submitting a single document. The document should have your last name(s), following by "8.part1" followed by the file extension, e.g. mine might be `kauchak9.part1.pdf`.

Answer each of the questions below (denoted **Question #:**) and put your answers into this document.

Hadoop Basics

At this point, you should have a working hadoop VM setup, you should be able to copy files to that VM and you should have your Eclipse environment setup to write MapReduce programs.

To run your own code on the cluster you'll need to go through a few steps:

1. Write your code in Eclipse and make sure that it's compiling without any errors.
2. In the shell (e.g. Terminal) go to the `bin` directory of your Eclipse workspace.
3. Create a `.jar` file of the compiled version of your code:

   ```
   $ jar -cvf name_of_jar_file.jar packages_to_include
   ```

   For example, if I wanted to do it for the `WordCount` class in the `demos` package, I would type:

   ```
   $ jar -cvf wordcount.jar demos
   ```

   Note that you need to do the full package structure, so it should be at the base of the `bin` directory in your workspace.
4. Copy this `.jar` file over to your VM server.
5. Finally, run it:

   ```
   $ hadoop jar name_of_jar_file.jar name_of_main_class
   ```

   For example, for the `WordCount` class it would be:

   ```
   $ hadoop jar wordcount.jar demos.WordCount
   ```

   Assuming everything worked well, you should see printed out:

   ```
   WordCount <input_dir> <output_dir>
   ```

Question 1: What are the results of running the `WordCount` demo code on the following text:

```
I do not like them in a house .
I do not like them with a mouse .
I do not like them here or there .
I do not like them anywhere .
I do not like green eggs and ham .
I do not like them , Sam-I-am .
```

One route to doing this would be:

- Create a copy of this file on your computer.
- `scp` the file to the VM.
- Use `hdfs` to create the input directory for your run (most hadoop programs operate on directories, even if it's just a single file).
- Use `hdfs` to copy the file from the VM to the hdfs.
- Run the hadoop `WordCount` program, specifying the output directory.
- Copy the output file(s) back to the VM using `hdfs` (either just `get` or `getmerge`) and then back to your computer using `scp`. (Of course, if the file is short, you could also just view it either in `hdfs` or on the VM).

MapReduce on paper

Viewing problems within the MapReduce framework can take a bit of practice. In addition, because these jobs run on a cluster, it can be very challenging to debug. Therefore, it is even more critical[1] to think through your MapReduce problems before you start coding them.

To give you practice with this, the main part of Part 1 of this assignment is to write pseudocode to solve a few problems within the MapReduce framework.

A pseudocode description of a MapReduce problem should include the following for both the `map` function AND the `reduce` function:

- The types for the input key/value pairs AND the output key/value pairs (that's four types). You should use the built-in hadoop types, specifically one of `Text, IntWritable, LongWritable, DoubleWritable and BooleanWritable`

[1] I know you all know that it's critical for normal programming too!

- Pseudocode describing what needs to happen in the function. These often will just be one or two statements. I don't want you to write actual code, but you should provide enough detail that someone reading it can understand exactly what the function should do and could implement it based on your pseudocode.

For example, for our word counting example, the following would be a reasonable submission:

```
- map
Input: key = LongWritable, value = Text
Output: key = Text, value = IntWritable

* split up the value text into words
* for each word, add an output key/value pair of <word, 1>

- reduce
Input: key = Text, value = IntWritable (or Iterator<IntWritable>, either way of
   writing this is fine for me)
Output: key = Text, value = IntWritable

* calculate the sum for each of the values associated with the key
* add an output key/value pair of <key, sum>, that is the word and how many times it
   occurred
```

Write MapReduce pseudcode for the following problems. You will be graded on how well you follow the guidelines, how appropriate your solution is to the MapReduce framework and the quality of your descriptions. Note, for some of these problems you may have to write more than one pair of map/reduce functions.

Question 2: Anagrams

Given a file containing text, the program should output key/value pairs where the value is a comma separated list of lines in the file that are anagrams, i.e. use exactly the same letters (ignoring spaces). The output key is up to you and will depend on your implementation. For example:

Input:

```
captain over rome
lives
cat
banana
emperor octavian
act
tac
elvis
```

Output:

```
<some_key>    cat, act, tac
<some_key>    elvis, lives
<some_key>    emperor octavian, captain over rome
<some_key>    banana
```

Question 3: K-NN (almost)

Given a CSV file containing a list of training examples (like we've been using all along in this class) the program should calculate the distance from each example in the file *from* a particular test example (which we'll assume is hard-coded into the code). The output from the program should be pairs of label/distance where label is the label of the training example and distance is the distance from that example to the test example. These examples should be output *in sorted order* by distance. You may assume:

- the test example is available as a variable (lets call it `test`) in both the map and reduce functions. We'll see next week how we can do this.
- access to a function `readExample` that takes an example line and gives back an `Example`.
- access to a function `getDistance` that gives the distance between two examples.

Question 4: Naive Bayes (almost)

Given a file containing labeled text examples where the the line consists of the label followed by the example text associated (e.g. like the wine data) we want to output the number of times a given feature (i.e. word) occurs in an example with a particular label, that is, the numerator for all features and labels in:

$$p(x_i|y) = \frac{count(x_i, y)}{count(y)}$$

You may choose any reasonable output key (be specific) that would allow someone reading the file to be able to figure out both the feature (word) and label. The value should be the number of times that feature/label combination occurred.

Question 5: Feature normalization

Given a file containing a list of examples of the form:

```
<label> <feature1> <feature2> <feature3> ... <featurem>
```

we want to generate a version of this file where the feature values have been *mean centered*. The output examples should include the label and should also appear in the same order as the original file.

You may assume:

- that each example has all of the features defined
- if (*hint!*) you use multiple map/reduce phases, you may assume that future phases have access (as, say, instance variables) to data produced in previous stages

When You're Done

Submit your a single document.

Part 2: Naive Bayes (again?)

For the last part of this assignment we're going to be implementing the Naive Bayes *training* algorithm within the MapReduce framework.

High-level requirements

Write a MapReduce program called `NBTrain` that takes two command-line arguments, an input directory containing training data and an output directory. Within the input directory, the training file(s) will consist of labeled, multi-class, examples of the form:

```
<label><tab><text>
```

where `<label>` is the class label, `<tab>` is a tab and `<text>` is the text representing the example. The wine training data is an example of such a file.

When the program is run, within the output directory it should generate two output directories: `priors` and `counts`. `priors` should contain one or more files that contain counts for how many times each label occurred in the training data in the form:

```
<label><tab><count>
```

The `counts` directory should contain one or more files that contain the feature label counts, that is the number of times each feature occurred in examples with each label in the form:

```
<label><tab><word><tab><count>
```

Additional requirements:

- If possible, all of your MapReduce programs must properly utilize a combiner class.
- To break up an example text into words, use the `parseFeatureLine` function in the `StringParser` class (see the description of the starter code below).

- All of your classes for this assignment should go inside the `ml.hadoop` package.
- The program should accept the input/output directory as command-line parameters. If the user does not enter the right number of parameters, you should display the usage.
- Try and be as memory and run-time efficient as possible. This means avoiding creating extra objects wherever possible and using static instance variables where appropriate (like we've done in class).

An Example Run

To make sure the specification is clear, here is an example running of the program.

Let's say you have a directory called `input` that has a single file in it with the following contents:

```
1    a b a c
0    b b
1    c d a
```

To run your program you all hadoop as follows:

```
$ hadoop jar assign9.jar ml.hadoop.NBTrain input output
```

After the program runs you should see an `output` directory. *Within* that output directory should be two directories, priors and counts. If you look inside the priors directory, you will find a file `part-00000` which has the following:

```
0    1
1    2
```

If you look inside the counts directory, there will also be a file called `part-0000` which has the following:

```
0    b    1
1    a    2
1    b    1
1    c    2
1    d    1
```

Notice, in particular, that we are counting the number of examples with a particular label that contain a particular feature (i.e. word) and NOT the number of times that a word occurs in text of a particular label, that is, if a word occurs multiple times in an example, it still only gets counted once.

Starter Code

At:

`http://www.cs.pomona.edu/classes/cs158/assignments/assign9/assign9-starter.zip`

I have included starter code for this assignment:

- `StringParser`: a class containing a static function to be used to split up the text in the examples.
- `NBClassifier`: a version of the Naive Bayes classifier that will take data as output from your MapReduce run and perform classification. This classifier should NOT be run on the hadoop cluster, but instead it should be run locally. You don't need to use this for this assignment, but I'm supplying it here in case you want to try it out and use your results to classify new examples.

One path to implementation

There are many ways to implement this problem. Below I outline one possible approach. No matter what approach you take, however, make sure that you work incrementally and test each step (map, reduce, etc.) as you go.

1. Think about how to solve the problem on paper. Specifically, think about how many MapReduce programs you will need and what their input and outputs will be.
2. Implement one or more MapReduce programs to generate the prior data. As always, break it up into writing the map, then the reduce step, debugging each one individually.
3. Implement one ore more MapReduce program to generate the counts data.
4. Write a final driver class that runs has the proper user interaction via the command-line and setups up and calls all of the MapReduce programs in the right sequence. If you've done it right, this should just be a few lines of code.

Hints/Advice

- As always, create some very simple data sets to test your system on where you know what the output should be. The example above is a good place to start, but I'd also recommend using a slightly more involved example (say with more than just two classes) and real words to make sure you're implementing everything correctly.
- Most of the times when you get errors, the feedback is pretty useful. I realize it prints out a lot, but take a look at what the main exception is and often this will point you in the right direction.
- Incremental development is critical here!

- Here are a couple of common errors you might see that aren't as obvious:
  - If you declare your mapper and reducer classes as nested classes, they *must* be declared as `static` classes. If you don't, you will get some error like:

    ```
    java.lang.RuntimeException: java.lang.NoSuchMethodException: ...Reducer.<init>()
    ```

  - Be careful to make sure that your driver setup matches your map/reduce input properly. For example, if your mapper outputs Text values, but in your driver you say it outputs IntWritable values, you might get something like:

    ```
    java.io.IOException: Type mismatch in key from map: expected
    ```

- If you get an exception, but your program still finishes fine, check the output directory. Due to the somewhat fragile nature of our hadoop cluster, you can occasionally get exceptions that are not your fault and that the cluster recovers from.
- If you're stuck on a particular exception, you *may* search the web to figure out how to solve the exception. Besides that, you should not be trying to find code to solve these problems beyond the documentation and class examples.
- I am asking you to try and be efficient about run-time and variable usage for these programs. Feel free to implement it however you like as a first pass. Once you have it working, go back and look at the code and see if you can come up with ways of reducing the number of variables used, making more things static, etc.

When You're Done

Make sure that your code compiles, that your files are named as specified and that you have followed the specifications exactly (i.e. method names, number of parameters, etc.).

Create a directory with your last name, followed by the assignment number, for example, for this assignment mine would be `kauchak9`. If you worked with a partner, put both last names.

Inside this directory, create a `code` directory and copy all of your code into this directory, maintaining the package structure.

`zip` this folder and submit that file.

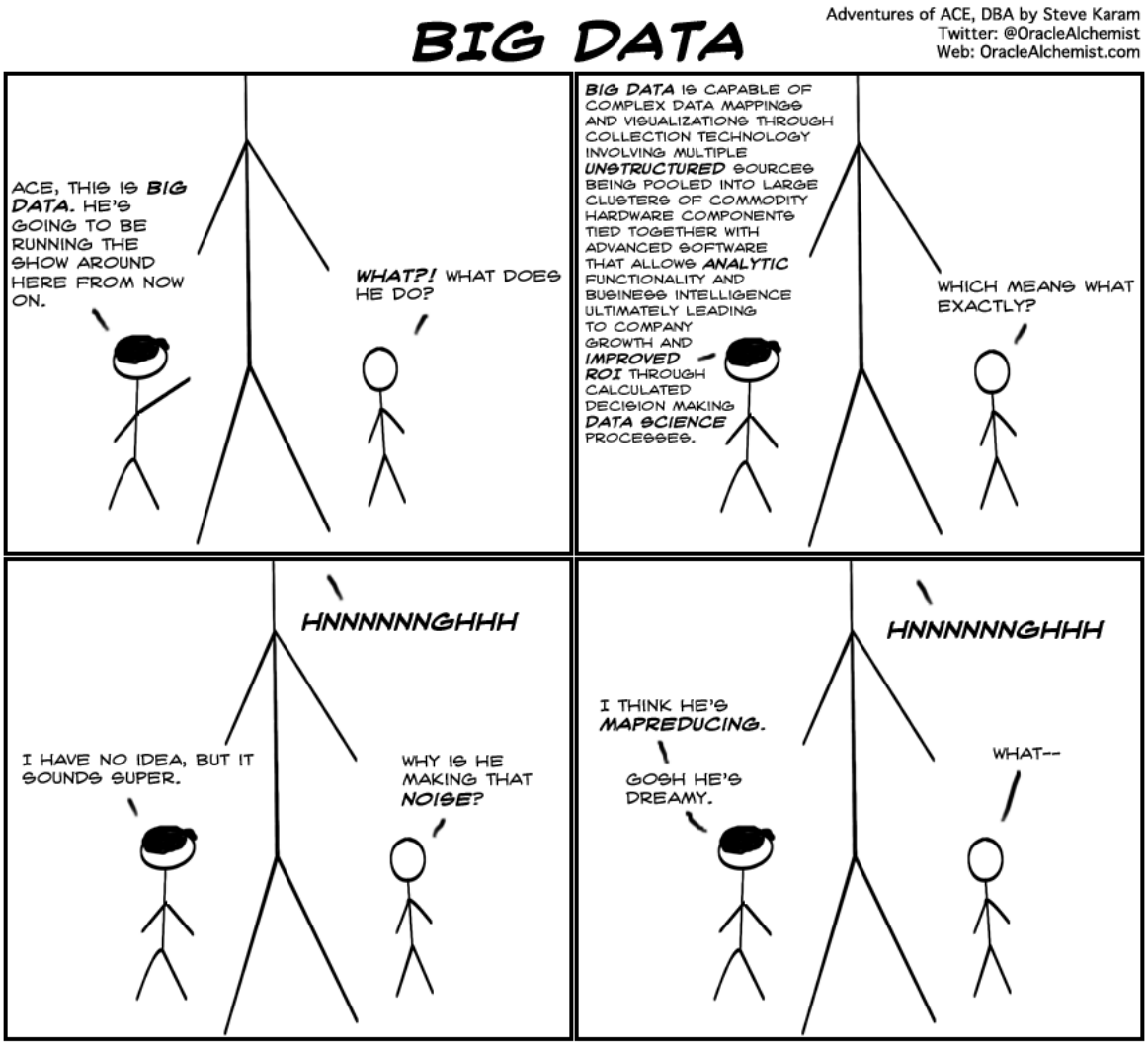
Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file
- Each class and method should have an appropriate JavaDoc

- If anything is complicated, it should include some comments.

There are many possible ways to approach this problem, which makes code style and comments very important here so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.



http://knowledgehubnetworks.com/information-technology/big-data-ace-comic/