Computing 1 - Introduction to Linux and programming in Bash

Hrvoje Stojic

September 15, 2017

Unix

Unix is any Linux or Mac operating system. Below I will use Unix and Linux interchangeably, but bear in mind that Mac is included as well.

Why Unix? Unix is arguably the best operating systems to learn and apply data science methods. Couple of advantages for which we value Unix over Windows environment:

1. Linux relies less on graphical user interfaces and induces solving problems programatically.
2. Most of the servers are running on Linux.[1] Serious computational tasks are too big for any desktop computer and you will rely a lot on servers, clusters or supercomputers, for running such computations.
3. Almost all of the software that will be used in the courses run exclusively on Linux machines or it is much easier to handle on Linux machines.
4. Last but not the least, you will become part of the open source movement!

[1] https://en.wikipedia.org/wiki/Usage_share_of_operating_systems

I strongly encourage you to read the following classic tutorial on "How to Become a Hacker" to get some insights on Linux and the hacking culture.[2] Many (if not all) of the ideas put forward in the tutorial apply to the data science program.

[2] http://www.catb.org/esr/faqs/hacker-howto.html

Getting started with Bourne again shell

The command line (also known as the command line interface, or CLI, or sometimes the terminal), is a plain text-based interface for executing commands on a computer. Linux has a powerful interpreter called shell - there are different types of Linux shells and we will focus on Bourne again shell or Bash, most widely used shell. Shell is actually an interface between the kernel that communicates with the hardware and the user.

Bash is probably a classic shell and has been used in Unix environments since the 80's. It may look a bit old-fashioned, but on the very contrary, it is a very powerful and efficient interface and is packed

with useful applications. More importantly, because servers are powered almost exclusively with Linux, knowing a bit of shell will allow you to easily work with servers, e.g. Amazon Web Services servers that you would use for heavy computing tasks.

I don't think it's crucial for you to become an expert in programming with Bash, but knowing the basics that are largely covered here is essential and will make you more productive.

How to access the command line and shell

Linux: You can search for the "Terminal" application from the Dash. On Ubuntu there is a shortcut `Ctrl+Alt+T`. Bash is installed by default on Ubuntu and in most other Linux distributions.

Mac OS X: Go to /Applications/Utilities and click on "Terminal" or search for "Terminal" in Spotlight. Bash is the default shell in Macs, but bear in mind that not all commands that are by default present in Linux are available in Mac, so you might need to install additional libraries (e.g. `ssh` is not available by default). To install additional libraries you will probably want to install and use a command line tool called Homebrew.

Windows: Windows is a special case, it does have a command line, but with its own, relatively limited system.[3] In case you still don't have Unix OS, it is possible to install Cygwin, which emulates Bash (www.cygwin.com/).

[3] Go to the Start Menu and click "Run", then type "cmd" and hit enter.

Bash basics

The shell prompt typically looks like

```
[user@host dir]$
```

where user is your username, host is the name of your machine or server and dir is the current directory. To use the shell, you simply have to type commands on the shell prompt and press 'Enter'.

An example of a command would be:

```
touch README.md
```

This command will create a file called `README.md`, if it doesn't already exist. Commands generally take the format:

```
[name of the command] [option] [option] [option] ...
```

where options will modify the behavior of the command. For example, you could execute

```
touch -d "next Thursday" README.md
```

which will modify the time stamp of when the file was created of the `README.md` file to next Thursday's data. To get help on the command you can type

```
touch --help
```

to get a summary of the command and its options. For a more detailed version type

```
man touch
```

Note that these help files might not be available for some non-default libraries.

When you execute a command, it matters from which directory, your `dir` as shown above. It matters when your command involves a file or directory name - you have to specify either a **relative** or **absolute** path to them. Specifying a file or directory as a **relative path** means you are specifying where it sits relative to the directory you're in. For example, in the command above the `README.md` file was created in `dir` directory. Specifying a file or directory as an **absolute path** means you are specifying where it sits on the computer in absolute terms, starting from the top level. Hence, if you execute

```
touch /some_other_dir/README.md
```

the file will be created in a folder `some_other_dir`. If you use an absolute path, the command will do the same thing no matter what directory you execute it from. **With a slash before the filename you indicate whether you will use a relative or an absolute path**:

- **Starting a path with a slash** means you want to give the entire path and ignore what directory you're currently in.
- **Not starting a path with a slash** means you want to give the path starting from the directory you're in.

Tips & tricks

Note that the shell keeps a **history** of commands. You can browse past commands you typed by using the keyboard arrows. Another way is by pressing `Ctrl+R` which starts a search mode where you start typing and whole history will be searched for the typed pattern.

Also note that the bash has an **autocompletion** function. By pressing the **tab** key, the bash will complete writing commands for you (if it can infer what you are typing). For instance, if you want to type the `touch` command, simply type `tou`, press `tab` and the shell will complete to write the command for you.

There are various others shortcuts that will make you far more efficient in using the shell, see for example here.

Filesystem

File and directory paths in Linux use the forward slash `/` to separate directory names in a path.

Example	Description
`/`	`root` directory
`/usr`	directory `usr` (sub-directory of `/` `root` directory)
`/usr/STRIM100`	`STRIM100` is a subdirectory of `/usr` (relative path)
`../src`	sibling directory to parent into `src` (relative path)
`../../../src`	Up three directories, into `src` (relative path)

Hello World!

Command	Description
`echo string`	Prints string on the console.

Browsing via the command line

Example	Description
`pwd`	Show the present working directory.
`cd /`	Change current directory to your `root` directory.
`cd`	Change current directory to your `HOME` directory.
`cd /usr/STRIM100`	Change current directory to `/usr/STRIM100`.
`cd INIT`	Change current directory to `INIT` which is a sub-directory of the current directory.
`cd ..`	Change current directory to the parent directory of the current directory.
`cd ~/Desktop`	Change current directory to the Desktop directory in your `HOME` directory.

Listing directory contents

Command	Description
`ls`	List a directory

Command	Description
`ls -l`	List a directory in long (detailed) format
`ls -a`	List the current directory including hidden files. Hidden files start with "."

Moving, renaming, and copying files

Command	Description
`cp file1 file2`	Copy a file
`mv file1`	Move or rename a file
`rm file1`	Remove or delete a file
`rm -r dir1`	Recursively remove a directory and its contents, BE CAREFUL!
`mkdir dir1`	Make a directory
`rmdir dir1`	Remove an empty directory

Instead of specifying an exact file name or directory name, you can also specify a general **file pattern** that might match multiple files. This is a feature you will often use. The most basic version is using the asterisk (*), which matches anything. It's also known as a wildcard.

Command	Description
`rm *`	Delete any file in the current directory
`rm *.txt`	Delete any file that ends in `.txt`
`rm data*`	Delete any file that starts with `data`

Note that deleting a file in shell deletes it forever, there is no trash bin where you could potentially recover the file.

Viewing and editing files

Command	Description
`cat filename`	Dump a file to the screen in ascii.
`more filename`	Progressively dump a file to the screen: ENTER = one line down SPACEBAR = page down q=quit
`less filename`	Like more, but you can use Page-Up too.
`vi filename`	Edit a file using the vi editor. All UNIX systems will have vi in some form.
`head filename`	Show the first few lines of a file.
`head -n  filename`	Show the first n lines of a file.
`tail filename`	Show the last few lines of a file.
`tail -n filename`	Show the last n lines of a file.
`wc -l filename`	How many lines are in a file.
`wc -w filename`	How many words are in a file.
`wc -c filename`	How many characters are in a file.
`grep word filename`	Show the lines that contain the string word.

Some other very useful shell tools you can explore on your own: `sed`, `cut`, `awk`, `find`, `locate`. Another useful shell capability that can be combined with many commands is **regular expressions**. With this tool in hand, `grep` will allow you to get quick info on some patterns in a dataset, which otherwise might take you a while to do.

You will find Vi in pretty much any Linux based machine. And relatively often you will find it useful to edit a file directly in terminal, especially when working on servers. Vi is somewhat strange if you have never used such type of text editor, but it is actually extremely optimized for working with the keyboard.[4] You should get to know few basic Vi commands that will allow you to quickly correct a file in a terminal.

[4] If you google a bit, you will find that Vi has many zealous followers. Equally strange editor at first is Emacs, which has equally strong suite of fans. Many Internet pages are filled with battle between these two sides, arguing which editor is the best in the world - Vi or Emacs.

Redirection

Command	Description
`echo "hello" > file`	Redirects the output of the echo command to a file `file`. If `file` does not exist, it will be created. If it already exists, it will be overwritten.
`echo "hello" >> file`	Appends the output of the echo command to the end of `file`.

Pipes

The pipe symbol `|` is used to direct the output of one command as an input of another.

Example	Description
`ls -l | head`	This commands takes the output of the long format directory list command and then shows only its `head`

Command Substitution

You can use the output of one command as an input to another command in another way called command substitution. Command substitution is invoked when by enclosing the substituted command in backwards single quotes.

Example	Description
``ls `echo ~` ``	List the content of the `HOME` directory.

File compression: gzip, and bzip2

A common compression utility is gzip (and gunzip). These are the GNU compress and uncompress utilities. The suffix for gzipped files is .gz. The bzip2 utility has (in general) even better compression than gzip, but at the cost of longer times to compress and uncompress the files.

Example	Description
`gzip data.csv`	Creates a compressed file `data.csv.gz`
`gunzip data.csv.gz`	Extracts the original file from `data.csv.gz`

Reading and writing tapes, backups, and archives

The tar command stands for "tape archive". It is the "standard" way to read and write archives (collections of files and whole directory trees). Often you will find archives of stuff with names like stuff.tar, or stuff.tar.gz. This is stuff in a tar archive, and stuff in a tar archive which has been compressed using the gzip compression program respectively. Chances are that if someone gives you a tape written on a UNIX system, it will be in tar format, and you will use tar (and your tape drive) to read it. Likewise, if you want to write a tape to give to someone else, you should probably use tar as well.

Example	Description
`tar cvf arc.tar file`	Create an archive file.
`tar xvf arc.tar`	Extract from the archive file.
`tar cvfz arc.tar.gz dname`	Create a gzip compressed tar archive containing everything in the directory `dname`.

Example	Description
`tar xvfz arc.tar.gz`	Extract a gzip compressed tar archive.

Shell Programming

Unlike the shells available on other operating systems, the Linux shell is a fully operational scripting language that allows you to define variables, use control structures such as if statements and loops and program functions.

Variables

You would create a variable in shell as follows:

```
var=foobar
echo $var
echo ${var}
```

By default, if no specifications are given, a variable can hold any type of data. To check the type of the variable, you can use the command `declare` with `-p` option.

```
declare -p var
```

If you want the variable to be of certain type, you need to declare it with the same command. For example, to declare a variable to be of integer type you would use `-i` option.

```
declare -i var=1
declare -p var
```

There should be no spaces between variable names, equal sign and statement. To see the variables set in the current bash session, use `set`. To remove the variable, you have to use command `unset`.

```
set | grep foobar
unset var
echo $var
```

There are three types of variables:

1. *Local Variables* - This is the type we have just created above, it is present within the current instance of the shell and it is not available to programs that are started by the shell.

2. *Environment Variables* - A variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly.
3. *Shell Variables* - A special variable that is set by the shell and is required by the shell in order to function correctly.

To create environment variables, use `export` command. To list the environment variables use `printenv` or `env` command. One environment variable that you might find necessary to modify from time to time is `PATH` which lists the directories where the files for executing the programs can be found. Another practical usage is to store some long path in an environment variable for a quick access to the folder.

```
printenv | grep foobar
export var2=foobar2
env | grep foobar
```

Exporting a variable will last for a single terminal session, to make the changes permanent you will need to look for a OS specific solutions. For example, various Linux distros have have system-wide config files for shells in `/etc/`, and you could place there a shell script that defines all sorts of environment variables that will be loaded when a terminal session is opened. Googling "make environment variable permanent" will give you more details.

Control Structures

If-else statements. You can use several types of formats, from double parenthesis to square brackets, depending on which one you use the equality operators might change. For example, with square brackets you need to use `-eq` to check for a numeric equality.

```
var=1
if (( $var==1 ))
then
    echo "first condition satisfied"
elif [ $var -eq 2 ]
then
    echo "second condition satisfied"
else
    echo "third condition satisfied"
fi
```

For loop:

```
list="jordi matt frank giovanna"
for name in $list;
do
    echo "remember to send e-mail to $name";
done
```

Functions

An example of a function without arguments.

```
hello () {
    echo "Hello world!"
}
hello
```

Examine a popular function called fork bomb, try to figure out what it does.[5] Do not try to run it on your computer!

[5] Fork bomb

```
:(){ :|: & };:
```

When put in more user friendly terms.

```
bomb(){
    bomb | bomb &
    };
bomb
```

Shell Scripting

It is of course possible to write shell scripts that can then be used as programs.

The following example is a real world example of a script that automatically downloads data from the Internet. The script is in the code folder of the brush-up. The scripts use some commands that have not been covered.

Create a text file called `download-and-clean-data.sh`

```
touch download-and-clean-data.sh
```

Open the script to edit it directly in the terminal with `vi`

```
vi download-and-clean-data.sh
```

Enter the text below (you have to press `i` to enter the editing mode and insert any text).

```
#!/bin/bash

echo "This scripts downloads data from finance yahoo"

list='IBM BAC'

site=http://ichart.finance.yahoo.com/table.csv

for series in $list
do
    echo $series

    wget "$site?s=$series&a=00&b=1&c=2000&d=11&e=31&f=2013&g=d&ignore=.csv"

    mv table* $series.csv

    tail -n +2 $series.csv | tac > tmp

    mv tmp $series.csv
done
```

Save and close the document with the following command `:wq`.
Make the script executable with

```
chmod +x download-and-clean-data.sh
```

You can run it in two ways

```
./download-and-clean-data.sh
bash download-and-clean-data.sh
```

The script is going to connect to the website `http://finance.yahoo.com`, download stock prices for the stocks BAC and IBM, clean the data files and save them in the current directory.

If you want to supply your bash script an argument when you run it from the terminal, you use `$1` inside a shell script.

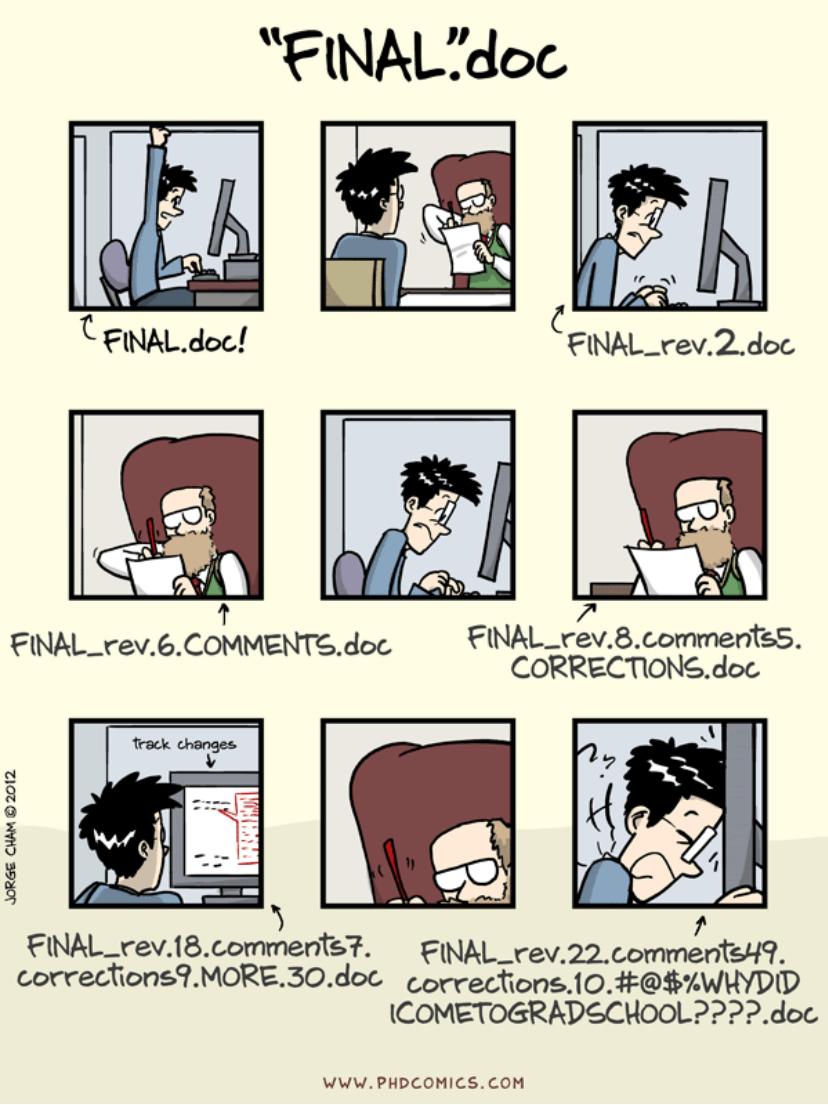
```
echo 'head -30 "$1" | tail -5' > piping_arguments.sh
```

This essentially means "the first filename (or other parameter) on the command line". Note that this command will place a text in the echo command in a file called `piping_arguments.sh`. We can now run this script as before, but with additional argument the script name, in the example below, the argument being `download-and-clean-data.sh`.

```
chmod +x piping_arguments.sh
bash piping_arguments.sh download-and-clean-data.sh
```

Version control with Git

Have you ever had situation like this?



Version control to the rescue! There are several version control softwares, but we will focus on arguably the most popular one, called `git`. It has been developed for collaborative effort of developing the Linux kernel - it allows multiple users to work simultaneously on the code of the same software project. `git` is also the software underlying

the GitHub, a popular public software repository.

At the moment we will ignore the main aspect of `git` - collaborative software development, and focus on local usage by a single developer - you. There are some benefits to reap from version control system even if it is used by a single developer:

1. You avoid situations from the comic, no more clutter.
2. Facilitates experimentation with the code.
3. Easy to share the code publicly through GitHub (e.g. R package).

Collaborative development brings in further complexities that go beyond this course and you will learn about these with practice. The aim is here to get started with version control and incorporate it into your own code development.

Here you will get bare basics and there are many good online tutorials to learn more about it. Check the official website for many good case usage examples and more detailed explanations[6].

[6] git-scm.com

Using git to work with a local repository

`git` is a command line tool. In Ubuntu for example, you can install it easily with

```
sudo apt install git
```

Once it is installed, as for other shell commands you can get an overview by typing

```
git --help
man git
```

Similarly, the basic structure of the command is

```
git <command> <arguments>
```

We will illustrate git by creating a local repository for developing a small `bash` project. In folder `cool_bash_project` we will initialize git with the command `git init`

```
mkdir cool_bash_project && cd cool_bash_project/
git init
ls -hla
```

You will note that git creates a hidden directory called `.git` where git data is stored. You should be careful not to modify the content of this folder unless you are absolutely certain you know what you are doing. The command `git status` tells you the current status of the repository.

```
git status
```

You should see this kind of output:

```
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

At the moment nothing has been put under version control, i.e. in git terminology, nothing has been "committed". So let's add some empty files to the repository and commit them to version control.

```
touch myscript.sh
touch Readme.md
git add Readme.md myscript.sh
git commit . -m "Initial commit"
git status
```

The procedure is always first adding the files with the `git add` command and then committing them with `git commit` which makes the modifications permanent. Note that commit command requires you to specify a brief comment explaining what changes you have done.

From now on you start editing your files and when you think you have done big enough modification (or finished a logical, closed modification), you commit your changes to the repository.

```
echo "Cool Bash Project" >> Readme.md
git status
git add Readme.md
git commit -m "Readme - written up project description"
```

Important thing to note is the mechanism by which changes in the code are saved. Git does not save a complete new file for each commit, instead it saves a difference between the versions. This makes git much more efficient in terms of data storage. Also, it means it needs to be able to easily identify differences. Hence, git works great for text files, but not for binary files and you should avoid committing such files (including images, MS doc's, pdf's etc). This is why you don't need to manually create versions - version control system is doing that for you in an efficient manner.

Of course, you need to be able to go back to previous versions. With `git log` command you can see a list of your commits. Among

other details about each commit (description you have put, date etc) there is a unique identifier associated to each commit, for example it would look like this

```
commit 63e60c919b7f8ca6fe2f5bb7388b9841f21335cb
```

You can use this identifier to revert the whole repository to the state in which it was during that commit with `git checkout` command

```
git checkout 63e60c919b7f8ca6fe2f5bb7388b9841f21335cb

# you can also arbitrarily shorten the identifier,
# it will probably catch it
git checkout 63e60c919b
```

With this you can work on your code and safely experiment with some modifications and potentially breaking the code, knowing that you can always go back to a version where everything has worked fine.

A better way to do this kind of experimentation is actually to work on a separate "branch". By default when you start a repository you are working on a "master" branch. When you want to implement a new feature it's better to create a new branch at certain point (think of it as a copy of the whole repo with its own history and commits) and when you are satisfied with changes and convinced everything works you "merge" it into the master branch.

When creating a new branch you automatically switch to it.

```
git checkout -b coolNewFeature  # creating a new branch
git branch  # list branches
git status
```

You can now create new files and commit them to the new branch.

```
echo "some code" >> feature.sh
git status
git add feature.sh
git commit -m "Adding feature x, draft"
```

But in the same time you can separately work on the main master branch, without changes and commits from "coolNewFeature" branch interfering with it.

```
git checkout master
echo "some further comments" >> Readme.md
git add Readme.md
git commit -m "Adding some further comments"

# nice way of showing branches and commits in them
git log --graph --all --oneline --decorate
```

Merging can often be done automatically, but in more complex projects it can be a bit more involved and you should investigate this topic further on your own. In case below merging will be done automatically as there are no conflicts that need to be resolved. You will automatically be asked to commit a message associated with the merge.

```
git merge coolNewFeature
git log --graph --all --oneline --decorate
```

You are often interested in comparing your current file with previous or some other version. because files are based on text this is easy to do and there are many tools to do that. In git a command that comes by default is `git diff`. Meld is a nice and simple graphical tool to visualize differences and make changes.[7].

[7] You'll have to first install it, check the website

```
git diff Readme.md
meld Readme.md
```

Two other essential command are those used to rename files, `git mv`, and to remove files, `git rm`.

```
# renaming
git mv Readme.md README.md
git status
git commit . -m "renaming files"

# removing
touch wrong.txt
git add wrong.txt
git commit -m "adding new files"
git rm wrong.txt
git commit -m "removing wrong files"
git status
```

Sometimes, your repository contains files that you are not interested to store in the repository, e.g. `.dropbox` file or `.Rdata`. You can

instruct git to ignore such files by creating a `.gitignore` file where you list individual files or whole directories.

There are many other features of git that you should get a handle on, but already with these few commands you are equipped for getting the individual development benefits of version control.

Collaborative development

This is where version control systems shine. It is a complex topic and there are couple of work flows. We will not go into details here but I will explain some basics.

Collaborative development version works as follows. A central server contains the main repository of code (say at GitHub or Bitbucket). Individual users download a copy of the code on their local machine, this is what you do with `git clone` command that you can also find at a GitHub repo. For example, you can clone the repo of this course from github.com/hstojic/BGSE_IntroToComputing and start working on the files

```
git clone https://github.com/hstojic/BGSE_IntroToComputing.git
cd BGSE_IntroToComputing
git status
git log
```

Each time you work on your local copy, it is important to make sure you are up to date with the latest changes in the main repository with `git pull` command.

```
git pull
```

In a basic workflow, I would assign you as a collaborator on the project and then you would be allowed to submit changes to the main repository, using the `git push` command.

```
# committing your own changes in the project
echo "some new code" >> feature2.sh
git status
git add feature2.sh
git commit -m "Adding feature y, draft"

# pushing your commit to the repo at the server (origin)
# from your master branch
git push origin master
```

Servers

For any serious industrial computations personal computers will not be enough, you will have to use servers, clusters and various others technologies. Hence, it is important that you get used to working with servers. We will do only the first steps here, launching a server on a cloud computing platform, connecting to it, transferring files and doing simple computation task.

The cornerstone of modern computing is cloud computing. We will make use of Amazon's platform called Amazon Web Services (AWS).[8] AWS is a collection of various cloud-based services, from computing to data storage solutions. With cloud computing it is far easier to maintain and scale computing resources – you do not have to invest any money into actual hardware, you simply pay for usage and you can easily scale up the resources when necessary. On a side note, Amazon is a pioneer in what they call human intelligence tasks, that you might find useful in your future jobs. It is a service called Amazon Turk, which you can use for a different type of machine learning task - acquiring labels for the data. Getting hands on big high quality data set where you could train your fancy supervised learning algorithms might be a difficult task - someone has to provide the ground truth for the observation. For example, in image recognition usually humans have to code what exactly is on the image - car, cow etc. Amazon Turk is often used for exactly such jobs - small tasks where human intelligence is needed. You would pay as little as couple of cents for classifying each observation and could easily increase the size of the training datasets.

[8] aws.amazon.com

Brief intro to AWS (focused on EC2)

Disclaimer. AWS has an extensive set of instructions for all of their services, please do not rely only on my brief instructions, read the AWS documentation and Google whatever is not clear.

First you need to register for the account. You will need a valid bank account for that and telephone number where they will call you to confirm the activation of the account.

They have various services. We will focus on Elastic computing service (EC2), a service where you are provided with virtual servers optimized for various purposes - for example, computing, storing data, or network services. You will be interested mostly in computing optimized hardware. Some other services worth mentioning are S3 which is a storage solution when downloads/uploads are frequent, RDS is their database specialized service and EMR is their map-reduce service specialized for distributed computing (Hadoop, Spark

etc). Another useful bit is a CloudWatch which you can use to shut down the servers once certain conditions are met (e.g. very low CPU utilization).

After signing in your account, or as they call it - AWS console, you will get to the main dashboard with the list of all AWS services. Before going to EC2, an important detail to know is that AWS has its resources dispersed across the world, in what they call zones. This is important because not all services might be available in some zones, and more importantly, some services cannot be used across the zones. For example, you cannot run computations in one zone while storing data in a zone on the other side of the world. This is of course to prevent inefficient data transfers. You should choose a zone closer to you to reduce the latency a bit.

By clicking on EC2 services you arrive to the EC2 dashboard. To start a virtual server, or instance, to use the AWS terminology:

1. Click on Instances tab
2. Click the button Launch instance.
3. This will open a wizard for launching a new instance. There you first have an option of selecting one of the vanilla instances - various types of Linux and Windows servers (Quick start) or you can choose your own image if you have it (My AMIs tab), or some of the public images (Community AMI tab).
4. In the next step you choose an instance type, essentially the hardware for your server. This should be chosen according to your computing needs.
5. Then you can already skip the rest of details and click Review and launch button.[9]
6. This will lead you to an overview of your instance, if you are satisfied you can click Launch button.
7. Final thing before the instance is really launched is to choose SSH key-pair that you will use to connect to the instance. SSH is a way to securely connect to your instance. Without the key you cannot access the instance, store it to a local folder that you can easily access.[10]
8. That's it! It may take a minute or two to start the instance. Once the instance is up and running you can connect to it in several ways and we will get to it in a moment.

[9] Important detail that will be set to default values are security rules - only access over SSH on port 22 is open. You can always return to these details later on.

[10] I suggest creating a special folder for SSH keys. Later on we will see how to create environment variables in Linux to access the folder easily.

When you mark an instance that is running on your instances overview, below the list of instances a window will open showing many relevant details about the instance, such as its public IP address, zone, security groups and so on.

Once you are done with using the instance you can either **stop** it or **terminate** it. If you only stop it, it will remain in your instance

tab and you can quickly start it again with all the data in it preserved. Stopped instances do not consume any resources (unless large volumes are attached to it). However, if you terminate it, it will be deleted completely and the data in it will be lost.

If you select a vanilla instance in step 3 you will get a basic operating system. Hence, setting it up so you can actually use it for some computational tasks might involve a bit of work. For example, if you run your computations in R you would need to install R and required packages. After you have set it up in a way you wanted you create an image with the server specifications, together with all the programs you have installed. You can make it publicly accessible as well. What this means is that usually there are some useful AMIs that you can use and do not have to bother with installing all the necessary programs by ourselves. For example, a kind soul provided everyone with a list of public AMIs that have a nice set of R related programs already installed.[11]

[11] R images

IMPORTANT. Be careful with starting virtual servers. AWS will charge your account for each service, and if you leave servers running for a long time it could be significant amount of money, especially if its a whole cluster. Always make sure that you stop the instances once you finish with computations.[12]

[12] Check the prices at aws.amazon.com/ec2/pricing/

Connecting to an EC2 instance

We will use SSH to connect to the servers.[13] SSH stands for "Secure Shell", a cryptographic network protocol for secure data communication between an SSH client (your computer) and SSH server (a remote computer, including EC2 instances) over an insecure network. Using SSH, you get secure access to the command-line interface of the EC2 instance and can do anything you are authorized to do as if you were sitting in front of that remote computer. In order to use SSH, you don't really need to know anything else about it.[14]

[13] Browser access to the instance is a bit buggy, but you are welcome to try it out later on.

[14] SSH is secure because it encrypts the communication, you should know some basics of encryption, check, for example RSA

Linux. First make sure that the instance is running and that the SSH port 22 in your security group is opened for your IP. Once an instance is running, you will be able to click on Connect tab. Since we will use SSH to connect to the server, you should make sure that SSH is the selected option, not browser. First make sure that you set the rights for your key before connecting to the instance. You will find a command that look like this:

```
chmod 400 your-key.pem
```

SSH comes by default on all Linux distros. Open a terminal in a folder where you keep the key, copy paste the command and press enter.[15] This has to be done only once for each key.

[15] Some Linux distros have an option in their file manager to open terminal in the current folder, some also have a shortcut `F4` that opens terminal directly. Otherwise you will have to open terminal and navigate to the folder, or simply add the whole path to `your-key.pem`

After that we can continue with connecting to the instance. In the connect tab you will find another command that should be introduced in the terminal:

```
ssh -i "your-key.pem" ubuntu@public-ip-of-the-instance
```

of course, you should substitute `your-key` with the name you have given to your key, and `public-ip-of-the-instance` with the IP of your instance[16]. Copy paste the command in the terminal, again in the same folder where the key is, and press enter. The terminal will ask you if you are sure that you want to continue connecting, type Yes, and after that you should be logged in your virtual server. With this your terminal effectively becomes the terminal of the Amazon EC2 instance until you log out[17].

[16] Sometimes in the connect tab you will see root instead of ubuntu, but you will not be able to log in the instance with root and you should use ubuntu nevertheless.

[17] You can close the connection with a shortcut Ctrl+D

Executing a computing task on a server

While computing in parallel on clusters of servers can be more involved, computing on a single server is straightforward. After you log into a server, executing any computation works pretty much the same as on your local computer.

We will run the shell script we have created previously as an example of a computation task. Before connecting to the instance we need to transfer the script to it. We will use `scp` (secure copy) command

```
scp -i "path/to/your-key.pem" path/download-and-clean-data.sh \
ubuntu@public-ip-of-the-instance:
```

Colon at the end is important, it marks the beginning of the path on the server, as it is now, you will copy it to the home folder of the server. To copy the whole folder you have to use option `-r`

```
scp -i "path/to/your-key.pem" -r path/to/the/folder/ \
ubuntu@public-ip-of-the-instance:
```

Next, same as before, you'll first have to make the script executable and then run it.

To transfer the files back we will again make use of `scp`

```
scp -i "path/to/your-key.pem" \
ubuntu@public-ip-of-the-instance:IBM.csv .

scp -i "path/to/your-key.pem" \
ubuntu@public-ip-of-the-instance:BAC.csv \
/home/user/Desktop/
```

Note that we have just reversed the order, now the server path comes first (copying from) and our personal computer is a destination (copying to). When copying `IBM.csv` we used a dot `.` as a destination, which means copy to the current folder (in which the terminal is placed at the moment), while when copying `BAC.csv` we instructed `scp` to copy it to a folder with a specified path (note that you have to substitute `user` with the username on your own computer).

Other server related things

Using servers without SSH connection

If the connection to the remote server is lost, the process you have started will be stopped. For long computational tasks chances are your local computer will get disconnected.

You can use `screen` application that comes on every Linux to start some processes and detach from it. Then we can log out from the server and the processes will continue to run. After starting `screen`, we can use again:

```
bash download-and-clean-data.sh
```

to start the computations. Then we press `CTRL+A` and write `detach` to safely detach from this screen process.

To see the list of all `screen` processes use `screen -ls` and to resume one of them, use `screen -R process_number`. For more details on using `screen` see e.g. here. There are alternatives, see for example tmux.

Transferring files to and from EC via a GUI

Using scp for transferring files can sometimes be a headache, especially if you do transfer files often and do not do it programatically (we'll come to this bit in a moment).

Filezilla. There are FTP applications with (somewhat attractive) graphical user interface that you can use to transfer folders, files, use drag & drop type of operations etc. Filezilla[18] is my personal favourite that works on all operating systems[19] Of course, you can use any of the similar applications for transferring files.

When you start FileZila, normally you will immediately encounter the "Host", "Username", "Password" box. To upload to or download data from a normal FTP server you simply enter the host, user and password in these boxes. Since Amazon uses key-pair system for added security, we have to additionally configure Filezilla.

1. Open "Settings" and click "SFTP"
2. Click "Add keyfile..." and then select the ".pem" file.
3. A dialog box will pop up and ask you if you want to convert the ".pem" file into a supported format. Click Yes.

[18] filezilla-project.org

[19] A windows specific alternative is Winscp that relies on PuTTY beneath the hood, winscp.net.

4. Name it with extension ".ppk" and save it.
5. Go back to the main interface and click button to open the site manager.
6. Click "New Site" and enter the host and user info. Set "Protocol" as "SFTP", "Logon Type" as "Normal", and leave "Password" as blank.
7. Click "Connect" and upon connection you can drag and drop files or folders.

Advanced - controlling your AWS account programatically

Manually connecting to an instance, sending files and starting jobs can be tiresome, especially if you do it often or on larger scale.

For that purpose Amazon has a collection of command line tools for controlling your AWS resources programatically.[20] In an essence, you can create a small shell script that will launch one or more instances of particular type, send files, start some computation jobs, retrieve files and finally, shutdown the instances.

AWS has API's for popular languages. For example, Python has a module called Boto that does the same thing, but from Python console instead.[21]

[20] Check the full documentation here.

[21] Check the AWS docs here and Python docs here.

References

1. Software carpentry has a amazingly well made tutorial on shell, more focused on applications to research and data analysis purposes. [link]
2. Learn Python the hard way, a famous book for learning Python, has a similar analogous resource for learning the shell. [link]
3. More specifically, there are many useful command line tools for data scientists. There is a book on the topic: "Data Science at the Command Line", check one of the blog post of the author.
4. Bash manual. [link]