* Terminal- interface to shell.
* Echo – to print data written on shell by user(put ‘’ around data 2 be printed as data may be seen in weird manner)
* $ symbol indicates that it’s a shell variable.
* $COLUMN \* $ROWS prints out d size of shell.
* Ls- to list down contents of a directory

Options: -l=>prints detailed list of files

* Pwd(print working directory)-to list content to current working directory
* ..-parent directory/folder
* . current directory
* ~ tilde- home directory
* Mkdir-make directory

Syntax=> mkdir directory location/folder name

* Mv- moving the files

Syntax=> mv [where to move from location] [where to move to location]

* Curl- download files from web.

Syntax=> curl [download link]

Options=>- L: follow redirects

-o: to get to write in file rather than on terminal

Syntax => curl -o [folder name want to save under] [link name]

It shows the source code to page

* Cat- reads the file and o/p the contents.
* Less – shows less of the content at first.In this ‘/’ is used to search the content ‘B’ to goback and ‘Q’ to quit the command.
* Rm- to delete files

Options=> -i: used as interactive ie prompts user b4 actually removing it.

* Rmdir- to delete directory.
* Grep- used to search a text file for lines with particular content.

Options => -c : to count the no. of tyms

* Wc(word count)
* Pipe- represented by ‘|’
* Types of variables:

1. Shell variables-> just internal to shell programs
2. Environment variables-> shared with programs that are run from within the shell.the most variable is $PATH which tells where ur program files are.

To add a directory to end of ur path : PATH=$PATH:/new/dir/directory name

* a Version Control System is just software that helps you control (or manage) the different versions...of something (typically source code).popular ones are git,subversion,mercurial.

a version control system is to help you maintain a detailed history of the project

Has 2 types of models:

1. Centralised Model->all users connect to a central, master repository
2. Distributed Model->each user has the entire repository on their computer

**(Git is version control tool and Github is service that hosts git projects)**

* **Version Control System / Source Code Manager**
* A **version control system** (abbreviated as **VCS**) is a tool that manages different versions of source code. A **source code manager** (abbreviated as **SCM**) is another name for a version control system.
* Git is an SCM (and therefore a VCS!). The URL for the Git website is [**https://git-scm.com/**](https://git-scm.com/) (see how it has "SCM" directly in its domain!).
* **Commit**
* Git thinks of its data like a set of snapshots of a mini filesystem. Every time you **commit** (save the state of your project in Git), it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. You can think of it as a save point in a game - it saves your project's files and any information about them.
* Everything you do in Git is to help you make commits, so a commit is *the* fundamental unit in Git.
* **Repository / repo**
* A **repository** is a directory which contains your project work, as well as a few files (hidden by default on Mac OS X) which are used to communicate with Git. Repositories can exist either locally on your computer or as a remote copy on another computer. A repository is made up of commits.
* **Working Directory**
* The **Working Directory** is the files that you see in your computer's file system. When you open your project files up on a code editor, you're working with files in the Working Directory.
* This is in contrast to the files that have been saved (in commits!) in the repository.
* When working with Git, the Working Directory is also different from the command line's concept of the *current working directory* which is the directory that your shell is "looking at" right now.
* **Checkout**
* A **checkout** is when content in the repository has been copied to the Working Directory.

## Staging Area / Staging Index / Index

* A file in the Git directory that stores information about what will go into your next commit. You can think of the **staging area** as a prep table where Git will take the next commit. Files on the Staging Index are poised to be added to the repository.

## SHA

* A **SHA** is basically an ID number for each commit. Here's what a commit's SHA might look like: e2adf8ae3e2e4ed40add75cc44cf9d0a869afeb6.
* It is a 40-character string composed of characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. "SHA" is shorthand for "Secure Hash Algorithm". If you're interested in learning about hashes, check out our [**Intro to Computer Science course**](https://www.udacity.com/course/intro-to-computer-science--cs101).

## Branch

* A **branch** is when a new line of development is created that diverges from the main line of development. This alternative line of development can continue without altering the main line.
* Going back to the example of save point in a game, you can think of a branch as where you make a save point in your game and then decide to try out a risky move in the game. If the risky move doesn't pan out, then you can just go back to the save point. The key thing that makes branches incredibly powerful is that you can make save points on one branch, and then switch to a different branch and make save points there, too.
* With this terminology in mind, let's take a high-level look at how we'll be using Git by looking at the typical workflow when working with version control.
* . before any file means it will be hidden directory.
* **config file** - where all project specific configuration settings are stored.  
  From the [**Git Book**](https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration):

Git looks for configuration values in the configuration file in the Git directory (.git/config) of whatever repository you’re currently using. These values are specific to that single repository.

For example, let's say you set that the global configuration for Git uses your personal email address. If you want your work email to be used for a specific project rather than your personal email, that change would be added to this file.

* **description file** - this file is only used by the GitWeb program, so we can ignore it
* **hooks directory** - this is where we could place client-side or server-side scripts that we can use to hook into Git's different lifecycle events
* **info directory** - contains the global excludes file
* **objects directory** - this directory will store all of the commits we make
* **refs directory** - this directory holds pointers to commits (basically the "branches" and "tags")
* the git init command to create a new, empty repository in the current directory. Running this command creates a hidden .git directory. This .git directory is the brain/storage center for the repository. It holds all of the configuration files and directories and is where all of the commits are stored.

## Git Clone Output Explanation

Let's look briefly at the output that git clone displays.

The first line says "Cloning into 'course-git-blog-project'...". Git is creating a directory (with the same name of the project we're cloning) and putting the repository in it...that's pretty cool!

The rest of the output is basically validation - it's counting the remote repository's number of objects, then it compresses and receives them, then it unpacks them.

* The git status command will display the current status of the repository.

This command will:

* tell us about new files that have been created in the Working Directory that Git hasn't started tracking, yet
* files that Git *is* tracking that have been modified
* a whole bunch of other things that we'll be learning about throughout the rest of the course ;-)
* Git Log- displays info abt existing commits.
* Git Show-displays info abt given commit.

 The git log command is used to display all of the commits of a repository. By *default*, this command displays:

* the SHA
* the author
* the date
* and the message

the --oneline flag is used to alter how git log displays information: This command:

* lists one commit per line
* shows the first 7 characters of the commit's SHA
* shows the commit's message

## git log –stat:

the --stat flag is used to alter how git log displays information.

This command:

* displays the file(s) that have been modified
* displays the number of lines that have been added/removed
* displays a summary line with the total number of modified files and lines that have been added/removed

## Annotated git log -p Output

Using the image above, let's do a quick recap of the git log -p output:

* 🔵 - the file that is being displayed
* 🔶 - the hash of the first version of the file and the hash of the second version of the file
  + not usually important, so it's safe to ignore
* ❤️ - the old version and current version of the file
* 🔍 - the lines where the file is added and how many lines there are
  + -15,83 indicates that the old version (represented by the -) started at line 15 and that the file had 83 lines
  + +15,85 indicates that the current version (represented by the +) starts at line 15 and that there are now 85 lines...these 85 lines are shown in the patch below
* ✏️ - the actual changes made in the commit
  + lines that are red and start with a minus (-) were in the original version of the file but have been removed by the commit
  + lines that are green and start with a plus (+) are new lines that have been added in the commit

The git show command will show only one commit.

Globbing lets you use special characters to match patterns/characters. In the .gitignore file,

**Git Tag Recap**

To recap, the git tag command is used to add a marker on a specific commit. The tag does not move around as new commits are added.

$ git tag -a beta

This command will:

* add a tag to the most recent commit
* add a tag to a specific commit *if a SHA is passed*

**The git branch command**

The git branch command is used to interact with Git's branches:

$ git branch

It can be used to:

* list all branch names in the repository
* create new branches
* delete branches

**he git checkout Command**

Remember that when a commit is made that it will be added to the current branch. So even though we created the new sidebar, no new commits will be added to it since we haven't *switched to it*, yet. If we made a commit right now, that commit would be added to the master branch, *not* the sidebar branch. We've already seen this in the demo, but to switch between branches, we need to use Git's checkout command.

$ git checkout sidebar

It's important to understand how this command works. Running this command will:

* remove all files and directories from the Working Directory that Git is tracking
  + (files that Git tracks are stored in the repository, so nothing is lost)
* go into the repository and pull out all of the files and directories of the commit that the branch points to
* *to delete the "footer-fix" branch*
* $ git branch -d footer-fix

$ git log --oneline --decorate --graph --all

The --graph flag adds the bullets and lines to the leftmost part of the output. This shows the actual *branching* that's happening. The --all flag is what displays *all* of the branches in the repository.

he git merge command is used to combine branches in Git:

$ git merge <other-branch>

There are two types of merges:

* Fast-forward merge – the branch being merged in must be *ahead* of the checked out branch. The checked out branch's pointer will just be moved forward to point to the same commit as the other branch.
* the regular type of merge
  + two divergent branches are combined
  + a merge commit is created

**Revert Recap**

To recap, the git revert command is used to reverse a previously made commit:

$ git revert <SHA-of-commit-to-revert>

This command:

* will undo the changes that were made by the provided commit
* creates a new commit to record the change

SERVERS AND BROWSERS

* Web servers using http.server are made of two parts: the HTTPServer class, and a request handler class.
* a variable called self.wfile, which is used to send the response. The name wfile stands for writeable file.
* self.wfile represents the connection from the server to the client
* I call serve\_forever on the HTTPServer, telling it to start handling HTTP requests. And that starts the web server running.
* The encodemethod on strings translates the string into a bytes object, which is suitable for sending over the network
* The **query** part of the URI is the part after the ? mark. Conventionally, query parameters are written as key=value and separated by & signs. (isch stands for Image Search.
* There is a Python library called urllib.parse that knows how to unpack query parameters and other parts of an HTTP
* urllib.parse.**parse\_qs**(*qs*[, *keep\_blank\_values*[, *strict\_parsing*]])
* Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a dictionary.
* **"Quoting"** in this sense doesn't have to do with quotation marks, the kind you find around Python strings. It means translating a string into a form that doesn't have any special characters in it, but in a way that can be reversed (unquoted) later.
* (And if that isn't confusing enough, it's sometimes also referred to as **URL-encoding** or **URL-escaping**).
* An action is idempotent if doing it twice (or more) produces the same result as doing it once. POST requests are not idempotent.
* Both, but they're different. r.content is a bytes object representing the literal binary data that the server sent. r.text is the same data but interpreted as a str object, a Unicode string.
* Being able to handle two ongoing tasks at the same time is called concurrency, and the basic **http.server.HTTPServer** doesn't have it. It's pretty straightforward to plug concurrency support into an **HTTPServer**, though. The Python standard library supports doing this by adding a mixin to the **HTTPServer**class. A mixin is a sort of helper class, one that adds extra behavior the original class did not have.
* Specialized web server programs — like [**Apache**](https://httpd.apache.org/), [**Nginx**](https://www.nginx.com/resources/wiki/), or [**IIS**](https://www.iis.net/) — can serve static content from disk storage very quickly and efficiently. They can also provide access control, allowing only authenticated users to download particular static content.
* Splitting requests up among several servers is called *load balancing*.
* Load balancing also helps handle conditions where one server becomes unavailable, allowing other servers to pick up the slack. A reverse proxy can *health check* the backend servers, only sending requests to the ones that are currently up and running. This also makes it possible to do updates to the backend servers without having an outage.
* The first two, the cookie's *name* and *content*, are also called its *key* and *value*. They're analogous to a dictionary key and value in Python — or a variable's name and value for that matter. They will both be sent back to the server. There are some syntactic rules for which characters are allowed in a cookie name; for instance, they can't have spaces in them. The value of the cookie is where the "real data" of the cookie goes — for instance, a unique token representing a logged-in user's session.
* The next two fields, *Domain* and *Path*, describe the scope of the cookie — that is to say, which queries will include it. By default, the domain of a cookie is the hostname from the URI of the response that set the cookie. But a server can also set a cookie on a broader domain, within limits. For instance, a response from www.udacity.com can set a cookie for udacity.com, but not for com.
* The fields that Chrome describes as *"Send for"* and *"Accessible to script"* are internally called *Secure* and *HttpOnly*, and they are boolean flags (true or false values). The internal names are a little bit misleading. If the *Secure* flag is set, then the cookie will only be sent over HTTPS (encrypted) connections, not plain HTTP. If the *HttpOnly* flag is set, then the cookie will not be accessible to JavaScript code running on the page.
* Finally, the last two fields deal with the lifetime of the cookie — how long it should last. The creation time is just the time of the response that set the cookie. The expiration time is when the server wants the browser to stop saving the cookie. There are two different ways a server can set this: it can set an *Expires* field with a specific date and time, or a *Max-Age* field with a number of seconds. If no expiration field is set, then a cookie is expired when the browser closes.

When a browser and a server speak HTTPS, they're just speaking HTTP, but over an encrypted connection. The encryption follows a standard protocol called [**Transport Layer Security**](https://en.wikipedia.org/wiki/Transport_Layer_Security), or **TLS** for short. TLS provides some important guarantees for web security:

* It keeps the connection **private** by encrypting everything sent over it. Only the server and browser should be able to read what's being sent.
* It lets the browser **authenticate** the server. For instance, when a user accesses [**https://www.udacity.com/**](https://www.udacity.com/), they can be sure that the response they're seeing is really from Udacity's servers and not from an impostor.
* It helps protect the **integrity** of the data sent over that connection — checking that it has not been (accidentally or deliberately) modified or replaced.

## How does TLS assure privacy?

The data in the TLS certificate and the server's private key are mathematically related to each other through a system called [**public-key cryptography**](https://en.wikipedia.org/wiki/Public-key_cryptography). The details of how this works are way beyond the scope of this course. The important part is that the two endpoints (the browser and server) can securely agree on a shared secret which allows them to scramble the data sent between them so that only the other endpoint — and not any eavesdropper — can unscramble it.

## How does TLS assure authentication?

A server certificate indicates that an encryption key belongs to a particular organization responsible for that service. It's the job of a certificate authority to make sure that they don't issue a cert for (say) **udacity.com**to someone other than the company who actually runs that domain.

But the cert also contains metadata that says what DNS domain the certificate is good for. The cert in the picture above is only good for sites in the **.herokuapp.com** domain. When the browser connects to a particular server, if the TLS domain metadata doesn't match the DNS domain, the browser will reject the certificate and put up a big scary warning to tell the user that something fishy is going on.

## How does TLS assure integrity?

Every request and response sent over a TLS connection is sent with a [**message authentication code**](https://en.wikipedia.org/wiki/Message_authentication_code) (MAC) that the other end of the connection can verify to make sure that the message hasn't been altered or damaged in transit.

* HTTPS only protects data in-transit

CRUD Create:

1. Create
2. Add
3. Commit
4. Query to check if its added to db

CRUD Update:

* 1. Find using .filter\_by(condn) always rerturns an array of data
  2. Reset the value
  3. Add to session
  4. Session commit

CRUD Delete:

1.find

2. sessdion.delete

3.session.commit()

##### Port forwarding allows us to open pages in our browser from the web server from our virtual machine as if they were being run locally.

Dynamic Paths in URL=>

“path/<type:variable name>/path” where type={int,string,path}

Inorder to avoid writing html code again and again in python use RENDER\_TEMPLATE(name,variables want to pass)

HTML escaping=> writing special charcters n html code

{logical code want to executr%}

{{print the result}}

Use {%endfor/endif%} in order to stop the loop

URL\_FOR(function name,arguments to be passed)=>used to provide hyperlinks within the page

API(Application Programming Interface): that allow external application to use the public info our app want to share

Authentication=> is the process of verifying who u are who u say u r

Authorization=> deals with the issue of do you have the right to acess

Standard used for authorization is OAUTH

PROS of using 3rd party authorization and authentication:

1.outsource auth handling to OAUTH providers(no need to think of how to encrypt or store passwords)

2.easier to register as use

AJAX=Asychronous Javascript AND XML request

Two methods to call AJAX request:

* + 1. Using .ajax method
    2. 2 using .getJson method

TO SPEED UP THE CONTENT OF WEBPAGE

1.request generic(common) html

2. request unique html

3.render generic html

4. render unique html

Finger is used to tell no. of current logged in users.

Info displayed by finger is sired in a directory called etc/passwd:

The format includes=>username:password:userid:groupid:description of user:home directory:user default shell

To create new user=> sudo adduser username

Info about sudo users=>etc/sudoers

To make a user as sudo user=> sudo (copy)cp (vagrant file)/etc/sudoers.d/vagrant (as) (stuent filoe) /etc/sudoers.d/student

Sudo nano etc/sudoers.d/st5udent(in order to edit sumthong)

In order to make user reset the password when next tym they log in=>sudo passwd -e(to expire the current password) username

Connectiong as a new user to server=> ssh username@ ip address -p port number

Generate key pair on local machine=> ssh-keygen

Key types=>DSA,ECDSA,RSA(default),ED25519

Hashing algo=>MD5,SHA256

To make firewall fubction=> pre-instaalled in ubuntu as ufw but is initially inactive

Inorder to set instructions using firewall=>sudo ufw default deny incoming OR sudo ufw deault allow outgoing

Steps for configuring ports in order to make firewall working or turning it to active state=>

* 1. Sudo ufw allow ssh
  2. Sudo ufw allow 2222/tcp
  3. Sudo ufw allow www(to make http request)
  4. Sudo ufw enable (setting firewall status to active)

[**WSGI**](http://wsgi.readthedocs.org/en/latest/) is a specification that describes how a web server communicates with web applications.

install PostgreSQL to server your data using the command sudo apt-get install postgresql.

Library is a bunch of javascript code that someone else wrote,packaged up and distributed and can be used in our application.eg: jquery is a lib for ajax,dom manipulation.

Library focus on application organisation.

Frameworks are collection of libraries.

KNOCKOUT JS=>

**ViewModel:** Knockout's ViewModel is similar to the Octopus. It separates the Model and the View

**Declarative Bindings:** Bindings allow you to connect the View and Model in a direct and simple way.

**Automatic UI Refresh:** Knockout's will update the View when the Model changes. And with the right declarative bindings, Knockout can update the Model when elements in the View change (such as input elements, checkboxes, etc).

**Dependency Tracking:** Knockout allows you to create a relationship between parts of the Model, and will automatically update Model data that depends on *other* Model data when that other Model data changes.

GGOOGLE MAP TYPES=>

* 1. Road mps
  2. Sateliites maps
  3. Hybrid(roads map over satellite maps)
  4. Terrain maps

# Nearby Search

Formerly know as simply Place Search, the nearby search is the simplest way to find places within a specified area. You are able to specify a center, and a radius to search within. This search will return a list of 20 places by default, and 60 with a Premium Plan license.

# Text Search

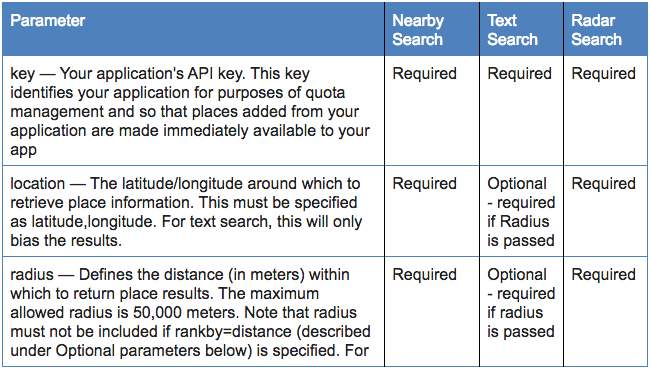
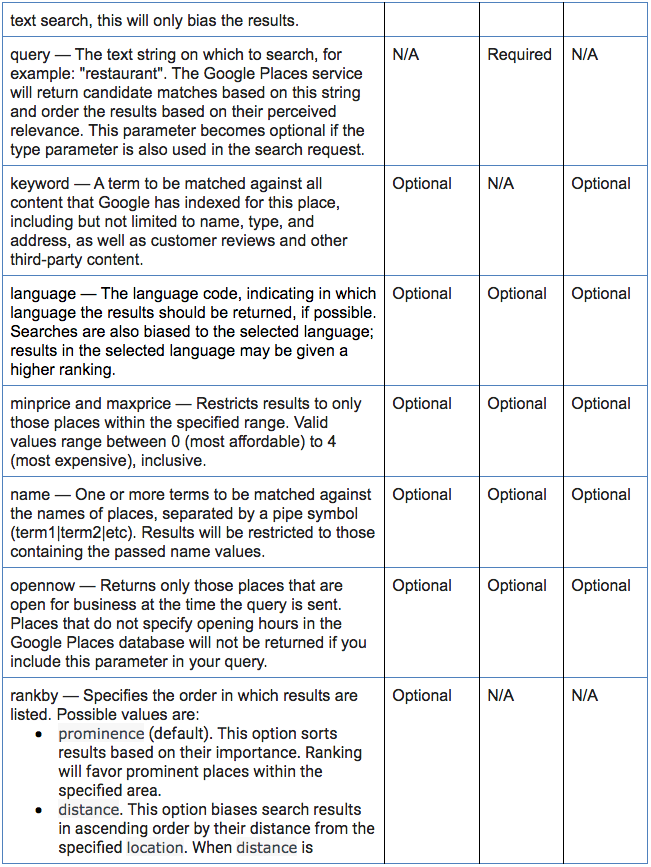
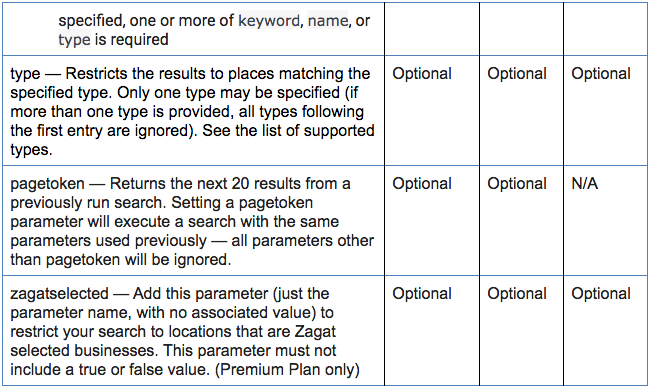
The Text search allows a user or system to execute a places search without a specified location, by using a text query. The search can also be biased by passing location information.

# Radar Search

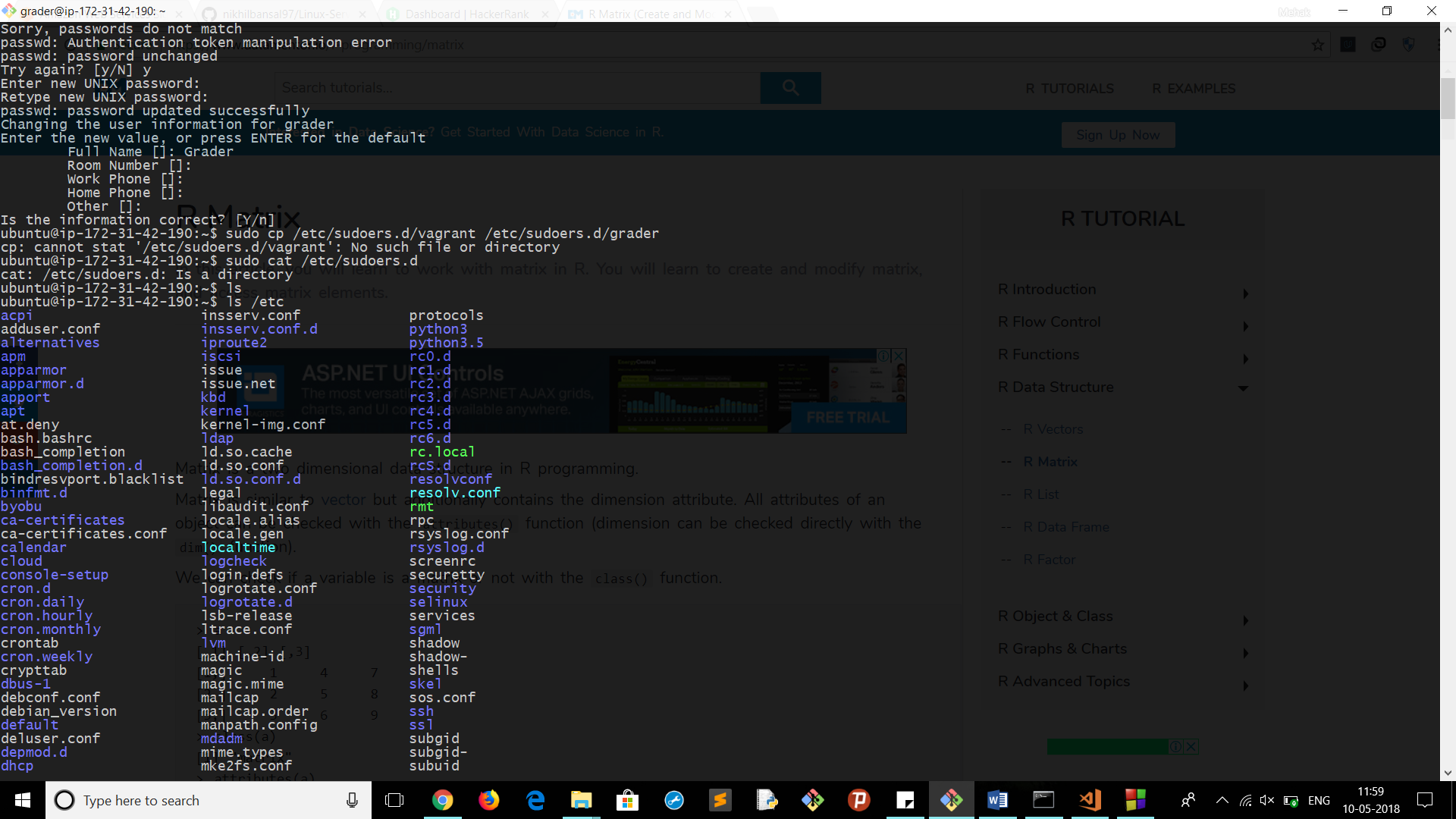
Radar Search allows the user to specify the same parameters as the Nearby Search, and will return 200 results instead of 20, but with limited returned data.

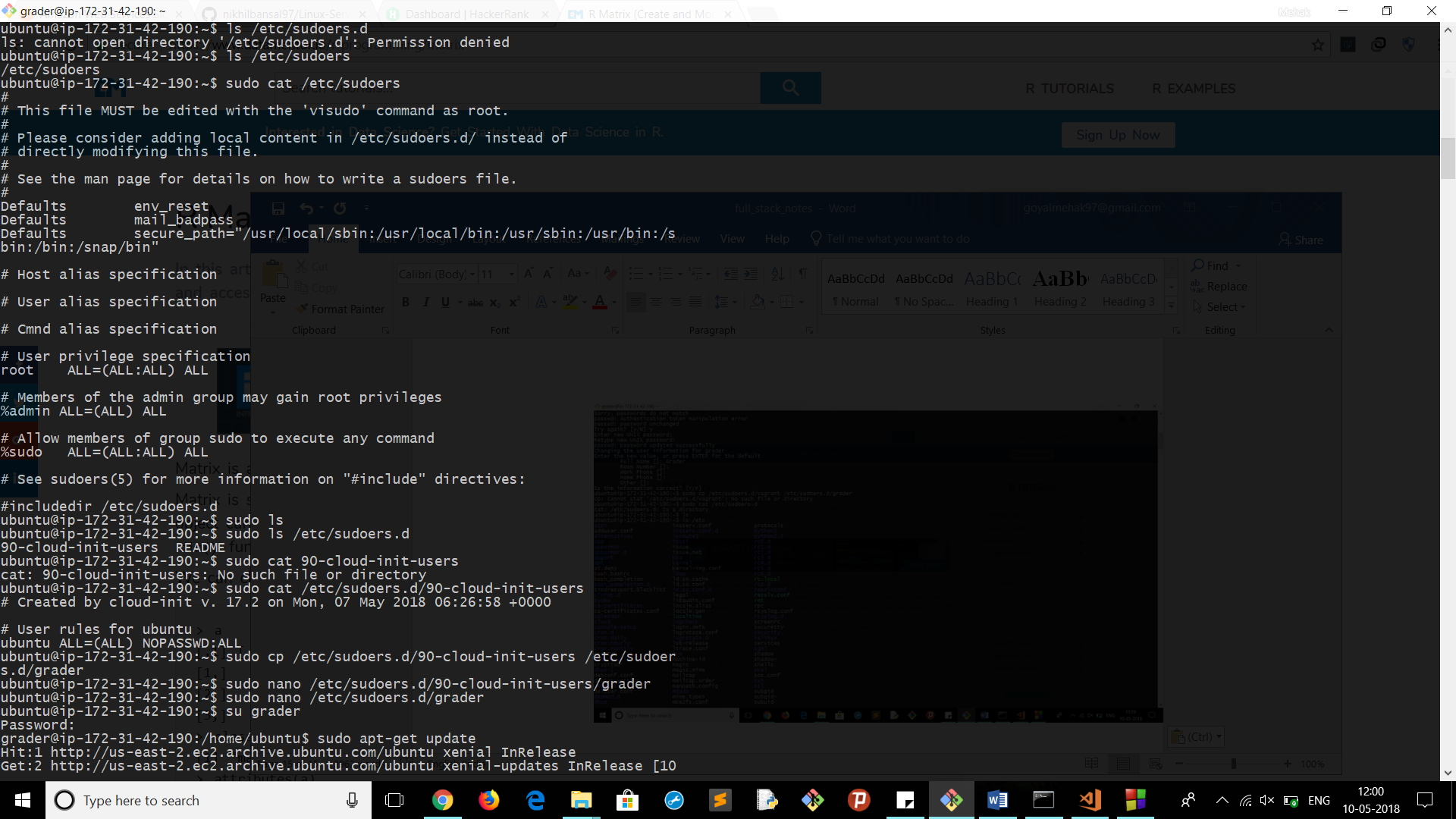
For all the searches, you can pass the value of the next\_page\_token to the pagetoken parameter of a new search to see the next set of results. If the next\_page\_token is null, or is not returned, then there are no further results.

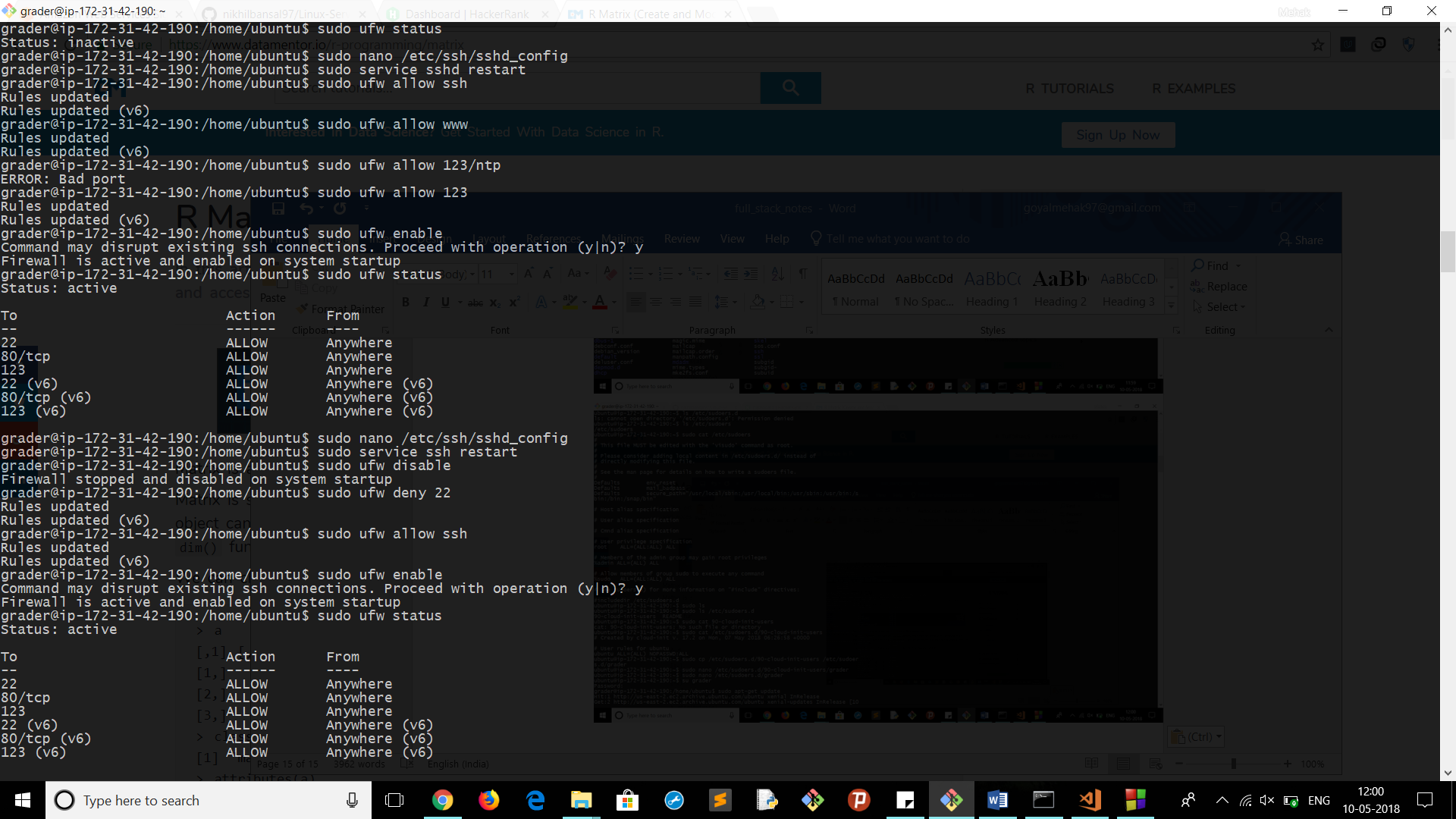
The following is a list of the required and optional parameters, and which search methods are able to be used with them.

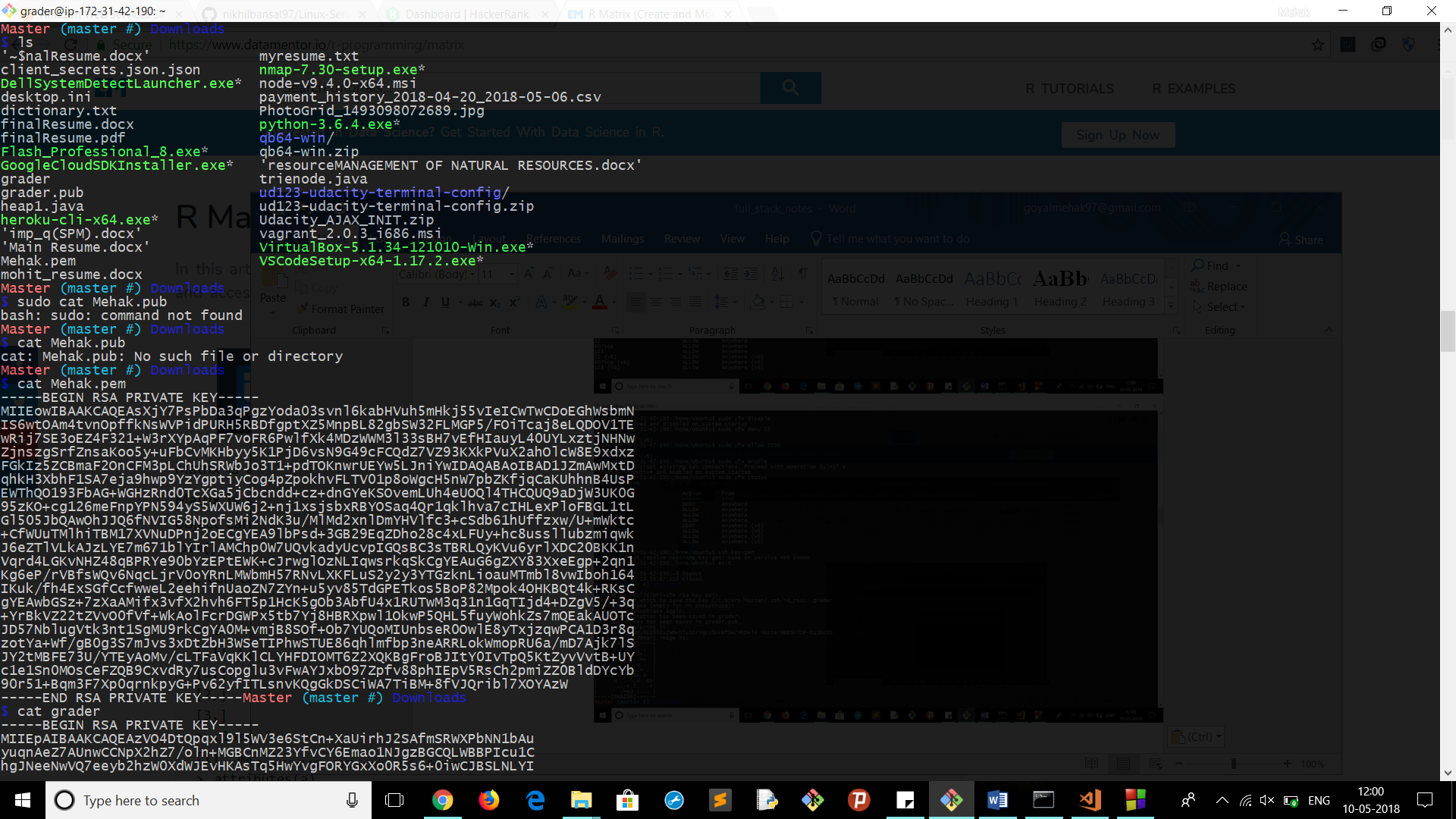


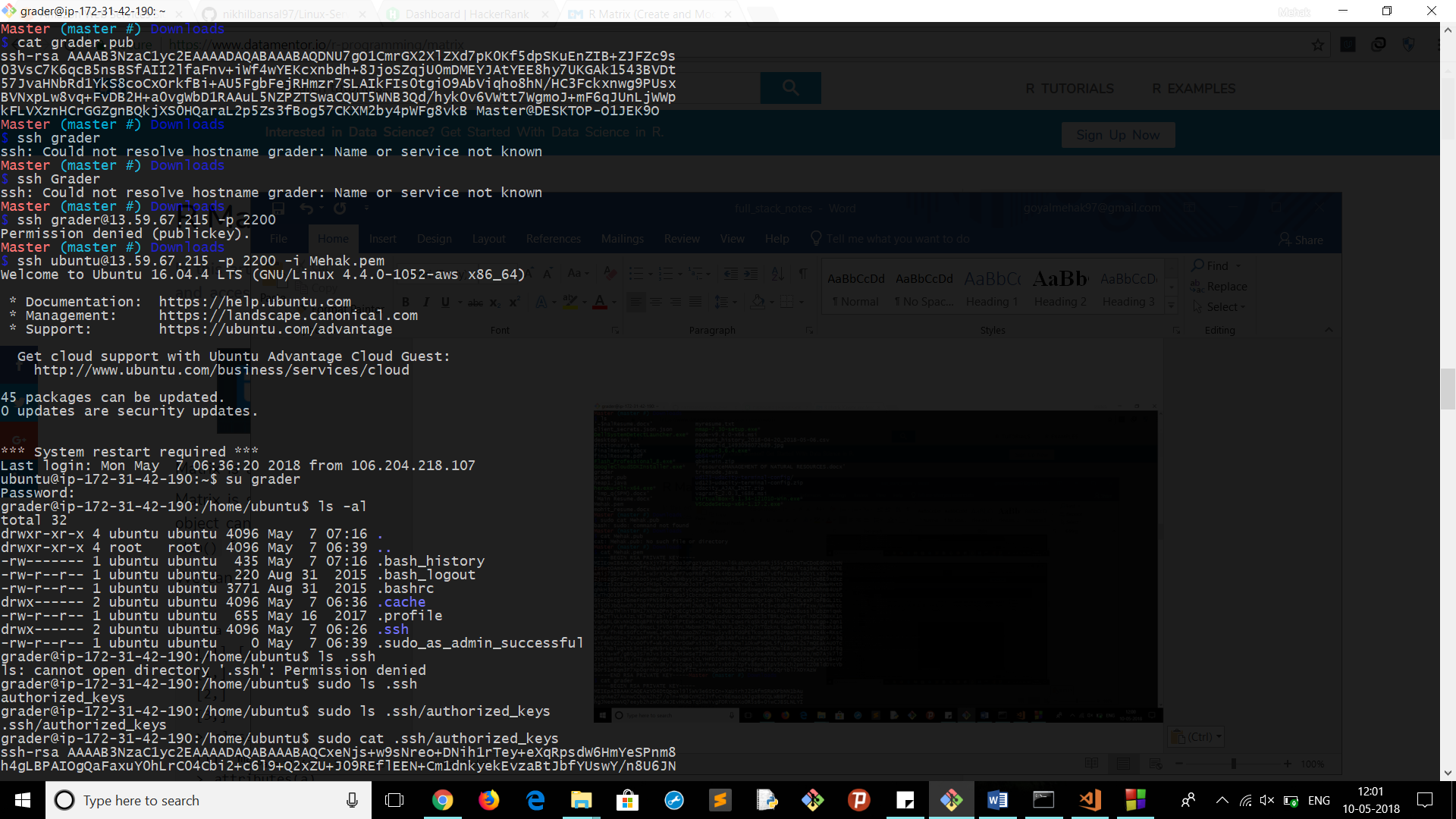


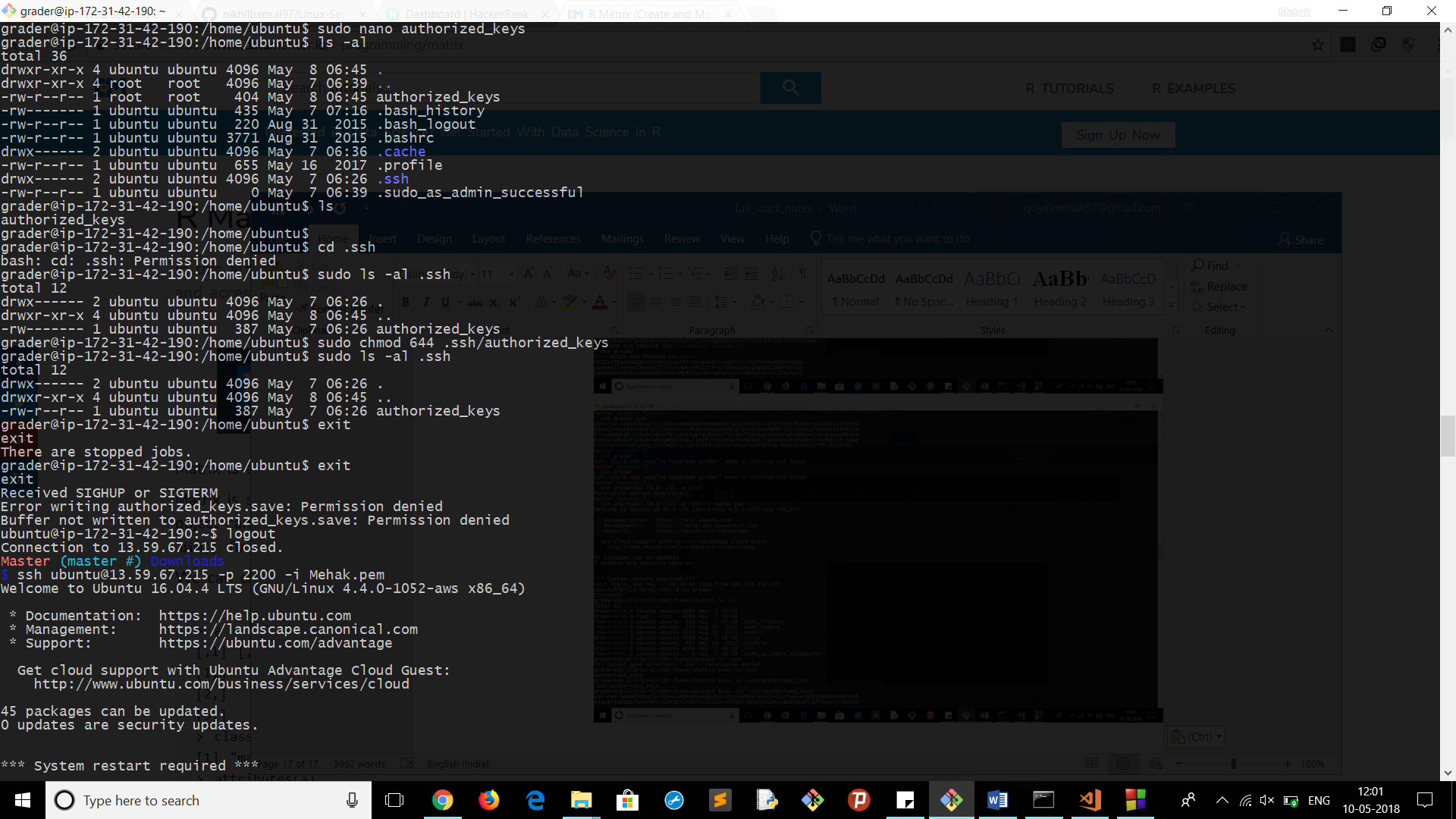


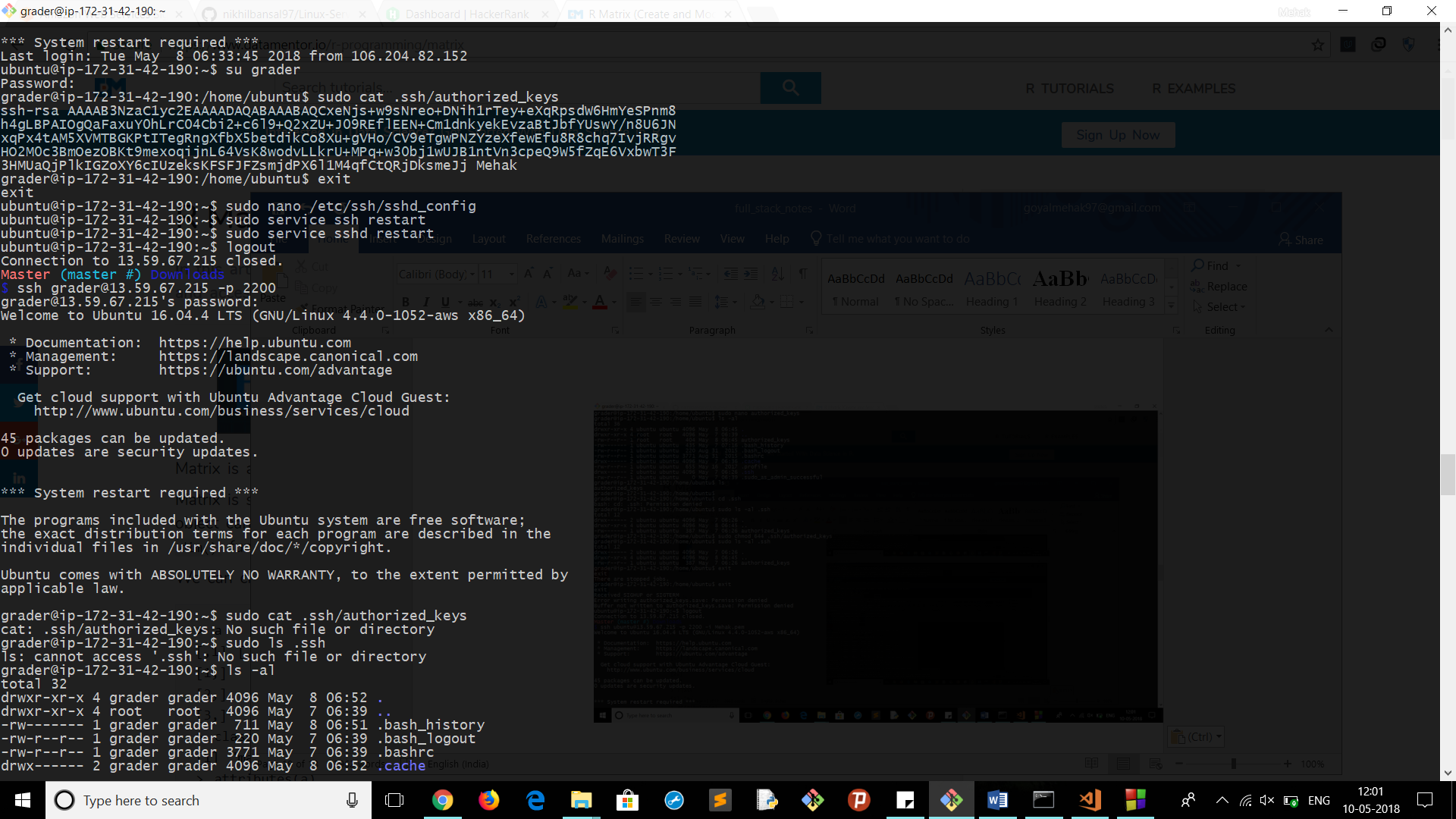


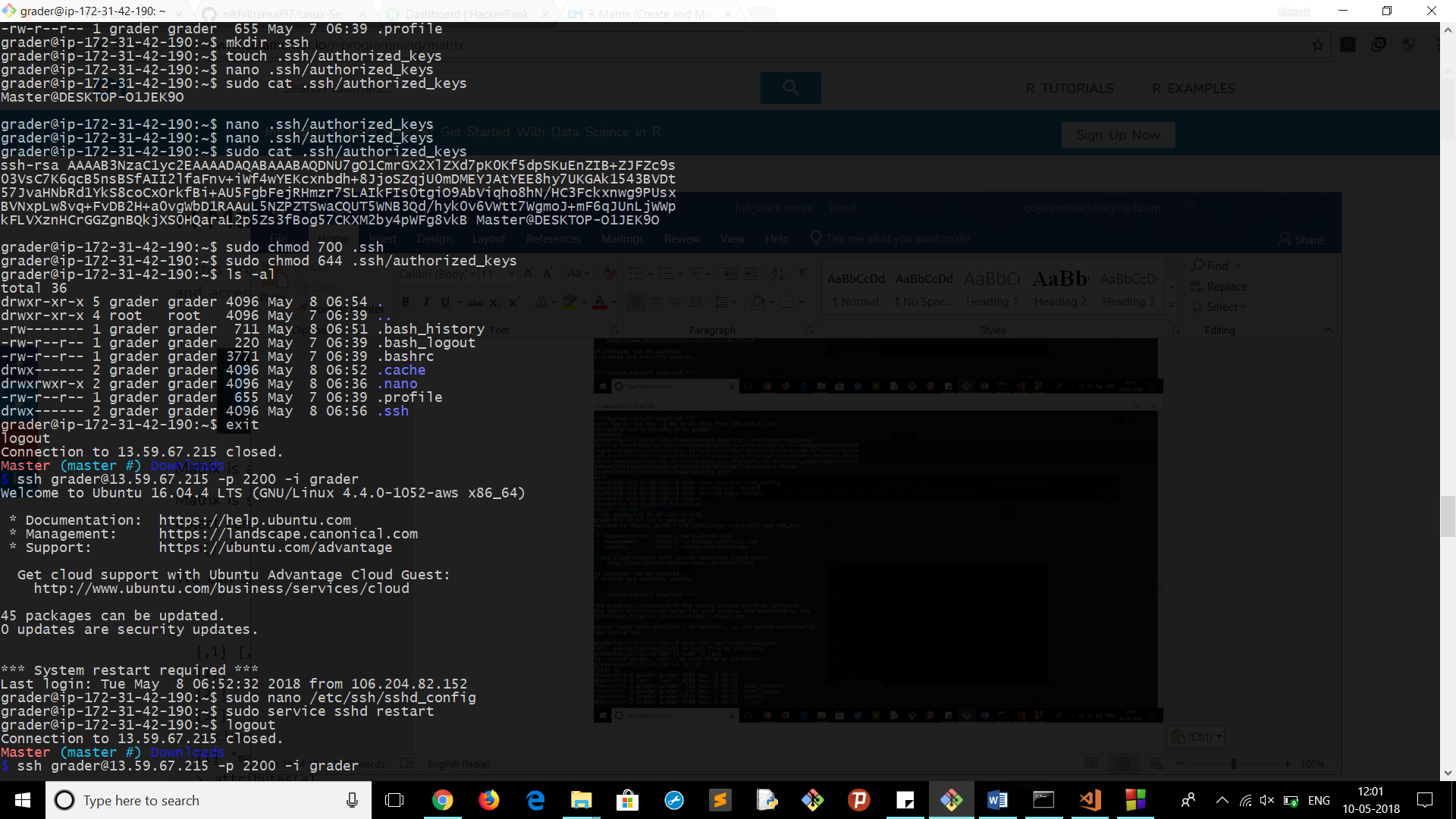


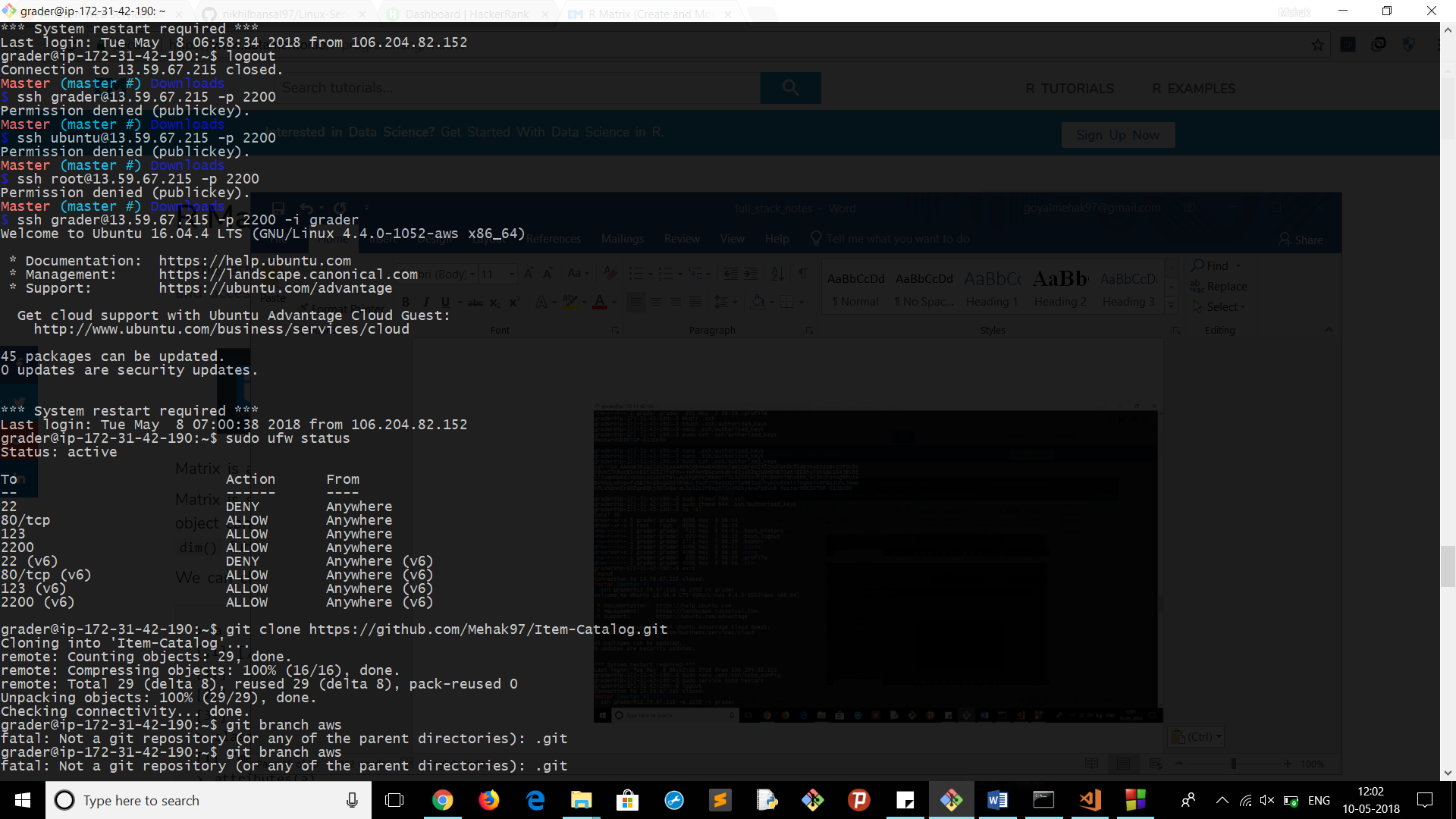




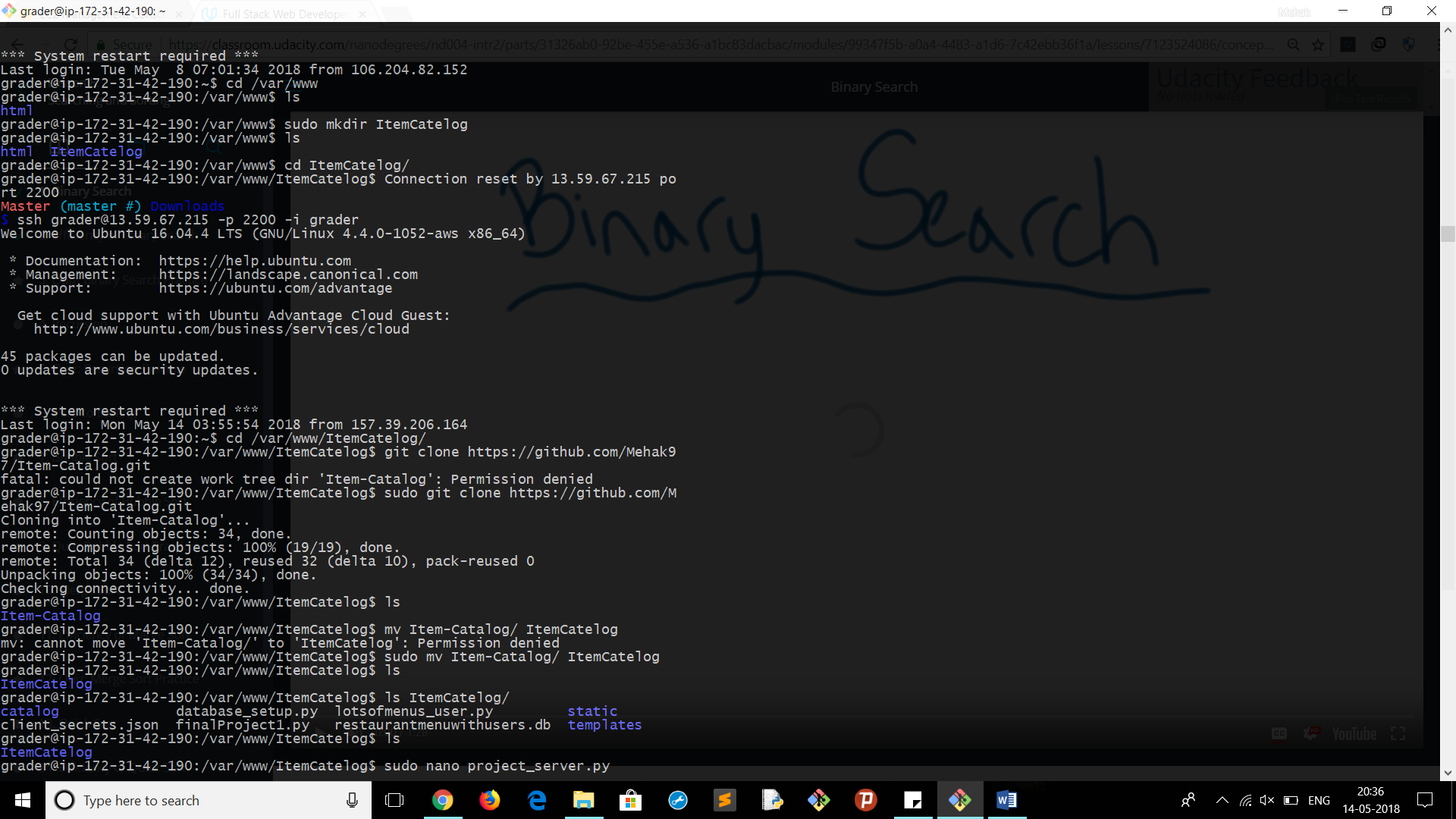


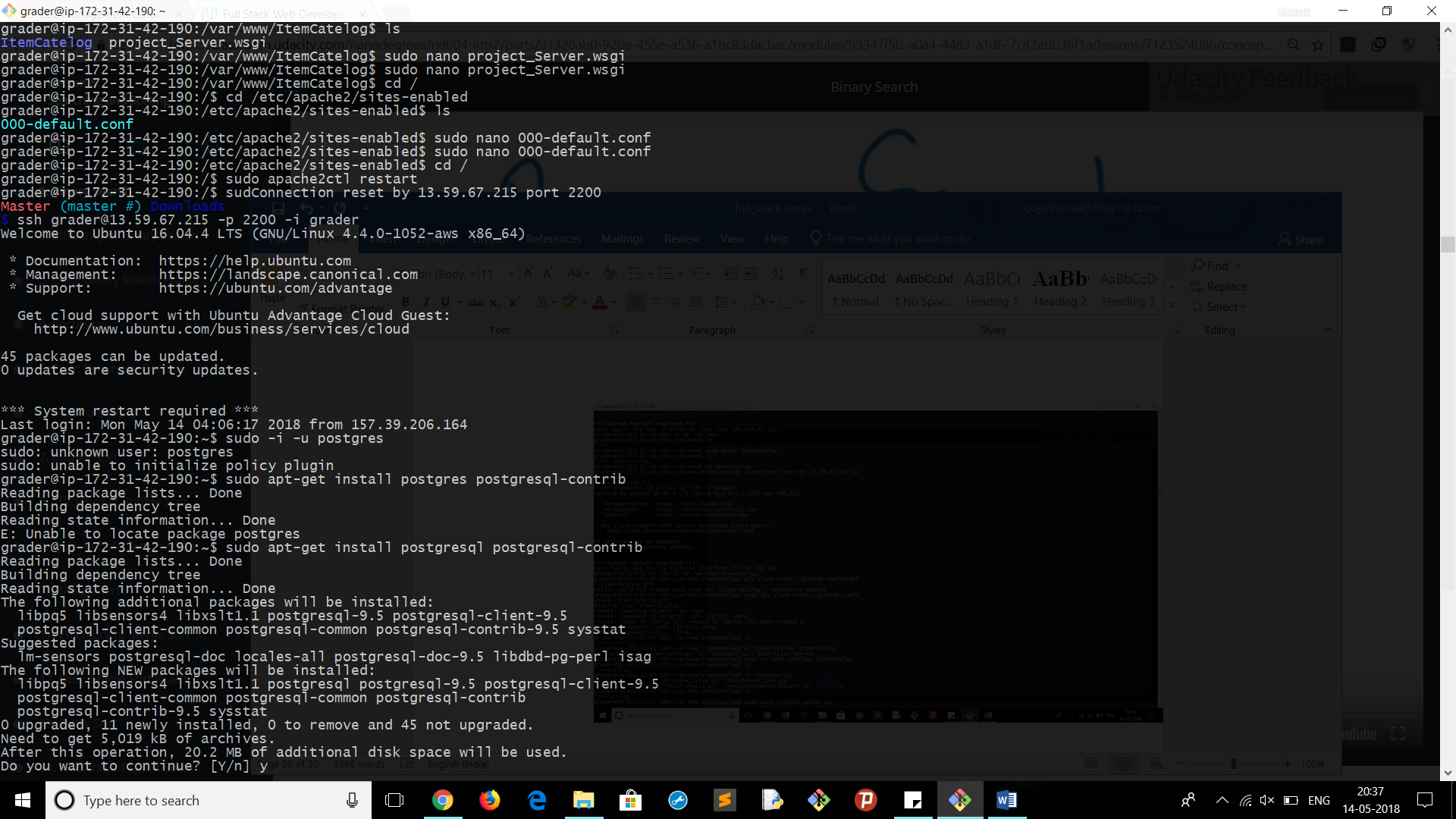




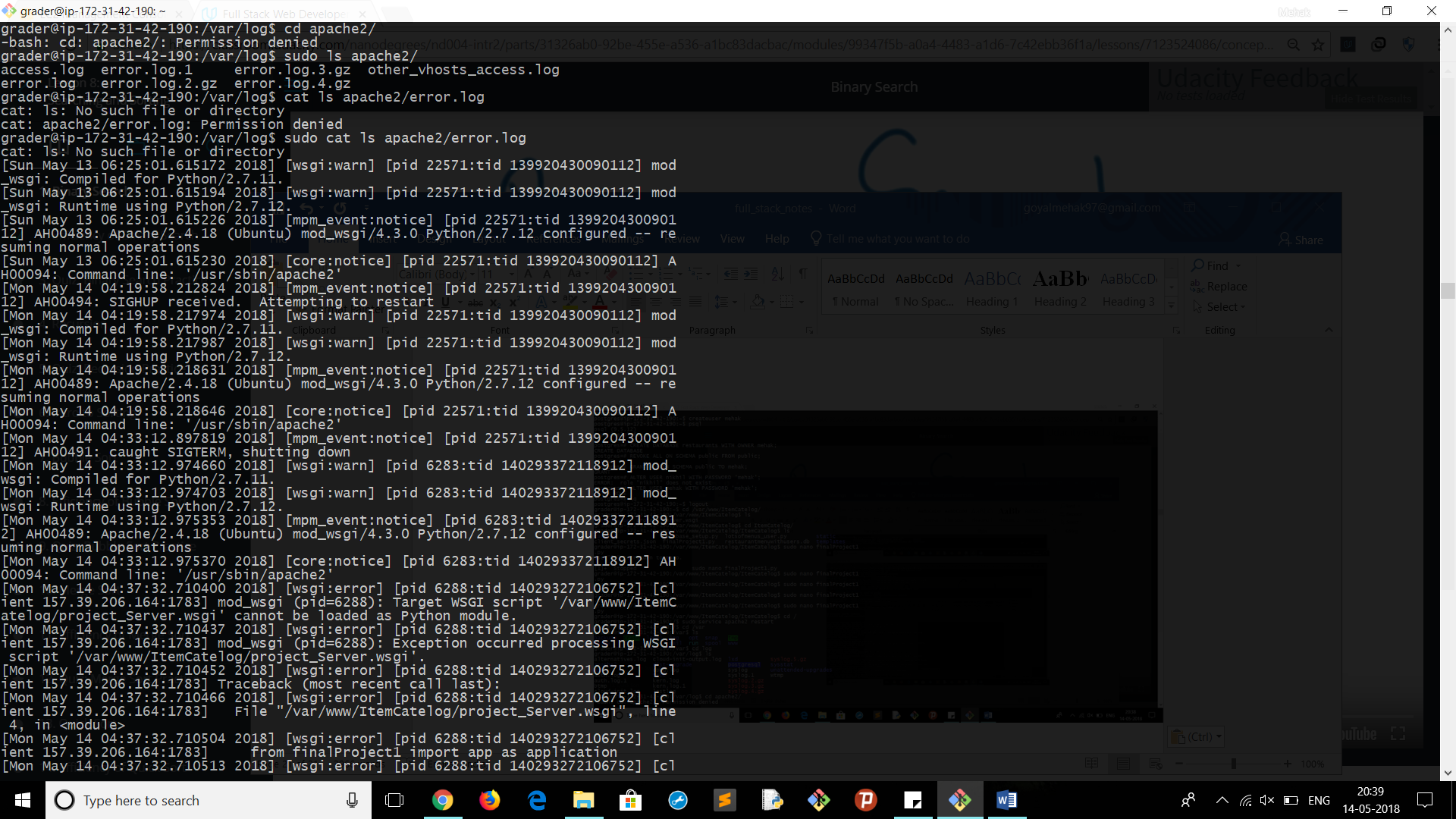


Sudo apt-get install apache2









Install packages required to run the project