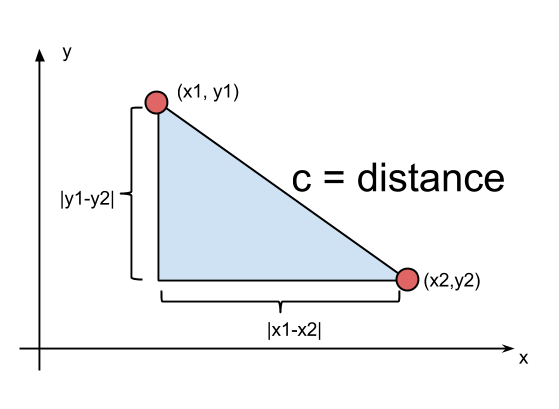
R Notebook sandbox: Playing with Distance

Table of Contents

# Distance

Sys.setenv(DATABASE\_WSU\_SANDBOX = “wsu\_sandbox\_db”); Sys.setenv(USER\_WSU\_SANDBOX = “wsu\_sandox\_user”); Sys.setenv(PASSWORD\_WSU\_SANDBOX = “!WSUCougars”);



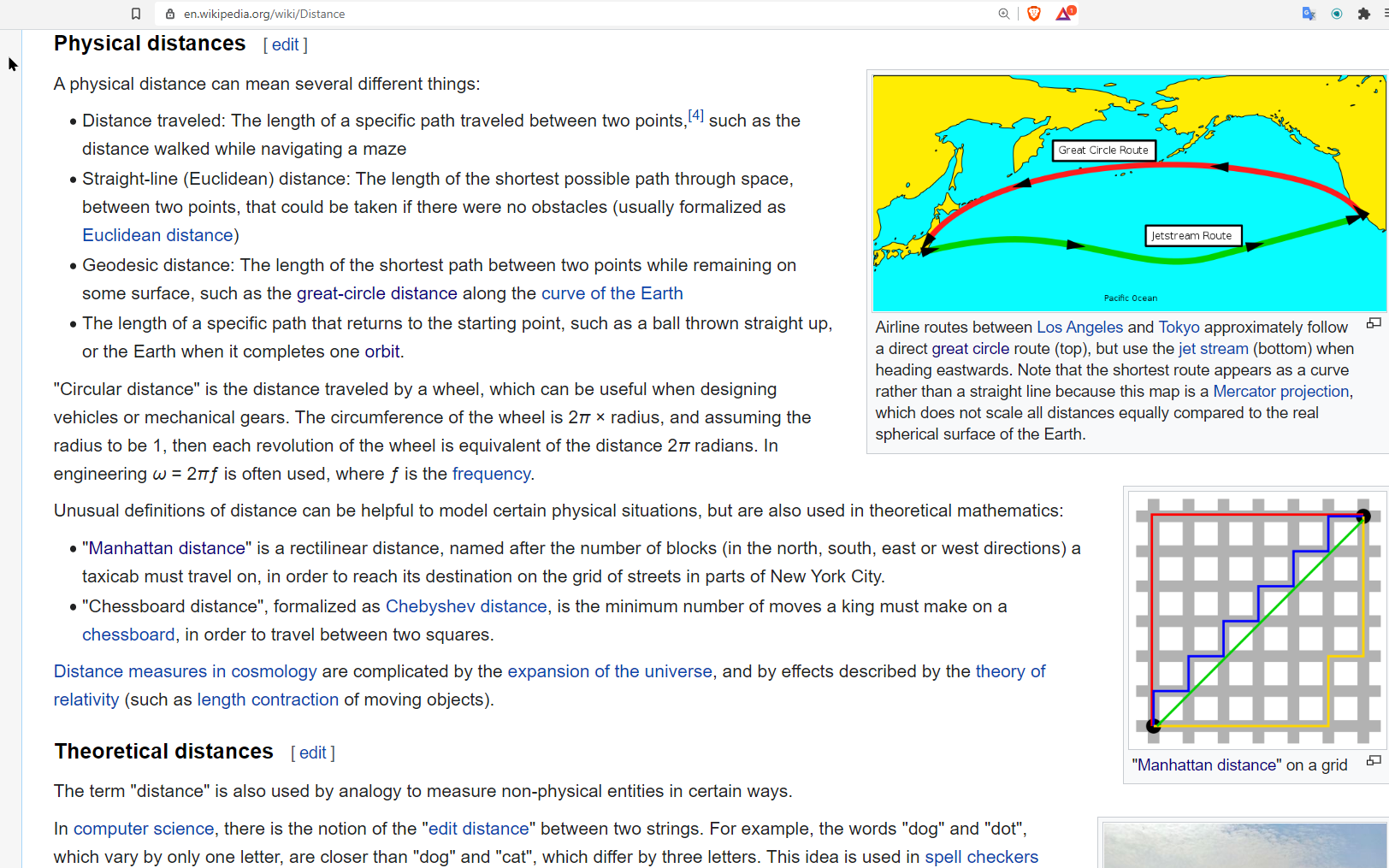
**Source:** [**https://i.stack.imgur.com/Vrq80.png**](https://i.stack.imgur.com/Vrq80.png)

*vs* pure HTML:

**Source:** [**https://i.stack.imgur.com/Vrq80.png**](https://i.stack.imgur.com/Vrq80.png)

* note: It seems currently, you can’t nest a IMG inside a div, the renderer (Knitter) breaks.
* note: It seems you can’t nest bold and italics, and where is the ‘underline’?

To compare two or more things, the concept of distance is essential. So let’s make certain we understand what it’s all about.



Source: <https://en.wikipedia.org/wiki/Distance>

## Manhattan Distance

This metric assume you are constrainted to a grid of city streets or blocks. You can’t walk on the diagonal (the adage: **shortest distance** between two points, …) because a large building is in the way.

## Euclidean Distance

This metric is based on the Pythagorean Theorem and attributed to Euclid. Some evidence suggests that the Babylonians and Chinese had this knowledge much earlier. Regardless, this is the 2-D “Flatland” variation of distance and can be applied to an n-D space.

## Mahalanobis Distance

This distance can be used to account for the density of the points to offset the Euclidean distance. It is like “adding gravity to the equation”. If several points are clustered together, their interdependence can be considered when computing distance.

See:

## Crow-flies Distances

We live on a spherical ellipsoid, so Euclidean Geometry is a bit limited. The earth bulges at the equator as it spins, so there are several formulas to calculate distances. An airplane flying from New York to Paris flies close to the North Pole to save distance on a Spherical Ellipsoid.

Note: The WIKIPEDIA screenshot above explains that shortest time-distance may not be shortest distance due to prevailing winds.

# Chicago

So let’s start in the city of Chicago:

chicago.willis.latlong = c(41.8791064,-87.6353986);  
chicago.cloud.gate.latlong = c(41.8826213,-87.6234554);  
chicago.lincoln.zoo.latlong = c(41.9217771,-87.6355701);  
chicago.marriott.latlong = c(41.8920961,-87.6244929);  
chicago.searle.latlong = c(41.8966098,-87.6175966);  
  
chicago = as.data.frame( matrix( c( chicago.willis.latlong,  
 chicago.cloud.gate.latlong,  
 chicago.lincoln.zoo.latlong,  
 chicago.marriott.latlong,  
 chicago.searle.latlong  
 )  
 ,ncol=2,byrow=TRUE) );  
 rownames(chicago) = c("Willis Tower", "Cloud Gate", "Lincoln Zoo", "Marriott", "Searle NW");  
 colnames(chicago) = c("latitude","longitude");  
  
chicago;

## latitude longitude  
## Willis Tower 41.87911 -87.63540  
## Cloud Gate 41.88262 -87.62346  
## Lincoln Zoo 41.92178 -87.63557  
## Marriott 41.89210 -87.62449  
## Searle NW 41.89661 -87.61760

dist(chicago, method="manhattan", diag=FALSE, upper=TRUE);

## Willis Tower Cloud Gate Lincoln Zoo Marriott Searle NW  
## Willis Tower 0.0154581 0.0428422 0.0238954 0.0353054  
## Cloud Gate 0.0154581 0.0512705 0.0105123 0.0198473  
## Lincoln Zoo 0.0428422 0.0512705 0.0407582 0.0431408  
## Marriott 0.0238954 0.0105123 0.0407582 0.0114100  
## Searle NW 0.0353054 0.0198473 0.0431408 0.0114100

# what does latitude, longitude mean?   
# If the earth were a perfect sphere with radius 4000 miles,  
# what would the factors be for latitude, longitude?  
  
# how many miles is 1 degree of latitude  
latitude.factor = 69; # rough mile estimate # 68.703 ?  
  
chicago$x.lat = chicago$latitude\*latitude.factor;  
  
# how many miles is 1 degree of longitude  
longitude.factor = 54.6; # rough mile estimate   
  
chicago$y.lat = chicago$longitude\*longitude.factor;  
  
chicago;

## latitude longitude x.lat y.lat  
## Willis Tower 41.87911 -87.63540 2889.658 -4784.893  
## Cloud Gate 41.88262 -87.62346 2889.901 -4784.241  
## Lincoln Zoo 41.92178 -87.63557 2892.603 -4784.902  
## Marriott 41.89210 -87.62449 2890.555 -4784.297  
## Searle NW 41.89661 -87.61760 2890.866 -4783.921

dist(chicago[,3:4], method="manhattan", diag=FALSE, upper=TRUE);

## Willis Tower Cloud Gate Lincoln Zoo Marriott Searle NW  
## Willis Tower 0.8946268 2.9536422 1.4917405 2.1797238  
## Cloud Gate 0.8946268 3.3632128 0.7104087 1.2850970  
## Lincoln Zoo 2.9536422 3.3632128 2.6528041 2.7178968  
## Marriott 1.4917405 0.7104087 2.6528041 0.6879833  
## Searle NW 2.1797238 1.2850970 2.7178968 0.6879833

# so let's go to Google maps and check out walking distance   
# It says about 0.9 miles  
  
# Other distances  
dist(chicago[,3:4], method="euclidean", diag=FALSE, upper=TRUE);

## Willis Tower Cloud Gate Lincoln Zoo Marriott Searle NW  
## Willis Tower 0.6957389 2.9442932 1.0760561 1.5502857  
## Cloud Gate 0.6957389 2.7815440 0.6562108 1.0168350  
## Lincoln Zoo 2.9442932 2.7815440 2.1354298 1.9946523  
## Marriott 1.0760561 0.6562108 2.1354298 0.4886502  
## Searle NW 1.5502857 1.0168350 1.9946523 0.4886502

dist(chicago[,3:4], method="maximum", diag=FALSE, upper=TRUE);

## Willis Tower Cloud Gate Lincoln Zoo Marriott Searle NW  
## Willis Tower 0.6520987 2.9442783 0.8962893 1.2077346  
## Cloud Gate 0.6520987 2.7017502 0.6537612 0.9652065  
## Lincoln Zoo 2.9442783 2.7017502 2.0479890 1.7365437  
## Marriott 0.8962893 0.6537612 2.0479890 0.3765380  
## Searle NW 1.2077346 0.9652065 1.7365437 0.3765380

dist(chicago[,3:4], method="minkowski", diag=FALSE, upper=TRUE);

## Willis Tower Cloud Gate Lincoln Zoo Marriott Searle NW  
## Willis Tower 0.6957389 2.9442932 1.0760561 1.5502857  
## Cloud Gate 0.6957389 2.7815440 0.6562108 1.0168350  
## Lincoln Zoo 2.9442932 2.7815440 2.1354298 1.9946523  
## Marriott 1.0760561 0.6562108 2.1354298 0.4886502  
## Searle NW 1.5502857 1.0168350 1.9946523 0.4886502

# same result, different package with more distance features  
library(philentropy); # install.packages("philentropy", dependencies=TRUE);

## Warning: package 'philentropy' was built under R version 3.6.3

distance(chicago[,3:4], method="euclidean", diag=FALSE, upper=TRUE);

## Metric: 'euclidean'; comparing: 5 vectors.

## v1 v2 v3 v4 v5  
## v1 0.0000000 0.6957389 2.944293 1.0760561 1.5502857  
## v2 0.6957389 0.0000000 2.781544 0.6562108 1.0168350  
## v3 2.9442932 2.7815440 0.000000 2.1354298 1.9946523  
## v4 1.0760561 0.6562108 2.135430 0.0000000 0.4886502  
## v5 1.5502857 1.0168350 1.994652 0.4886502 0.0000000

distance(chicago[,3:4], method="canberra", diag=FALSE, upper=TRUE);

## Metric: 'canberra'; comparing: 5 vectors.

## v1 v2 v3 v4 v5  
## v1 0.000000e+00 -2.618298e-05 0.0005082130 9.283577e-05 1.073528e-04  
## v2 -2.618298e-05 0.000000e+00 0.0003981039 1.071784e-04 1.335358e-04  
## v3 5.082130e-04 3.981039e-04 0.0000000000 2.909256e-04 1.977025e-04  
## v4 9.283577e-05 1.071784e-04 0.0002909256 0.000000e+00 1.451704e-05  
## v5 1.073528e-04 1.335358e-04 0.0001977025 1.451704e-05 0.000000e+00

#distance(chicago[,3:4], method="minkowski", diag=FALSE, upper=TRUE);  
  
getDistMethods(); # lot's of methods, some with their own parameters ..

## [1] "euclidean" "manhattan" "minkowski"   
## [4] "chebyshev" "sorensen" "gower"   
## [7] "soergel" "kulczynski\_d" "canberra"   
## [10] "lorentzian" "intersection" "non-intersection"   
## [13] "wavehedges" "czekanowski" "motyka"   
## [16] "kulczynski\_s" "tanimoto" "ruzicka"   
## [19] "inner\_product" "harmonic\_mean" "cosine"   
## [22] "hassebrook" "jaccard" "dice"   
## [25] "fidelity" "bhattacharyya" "hellinger"   
## [28] "matusita" "squared\_chord" "squared\_euclidean"  
## [31] "pearson" "neyman" "squared\_chi"   
## [34] "prob\_symm" "divergence" "clark"   
## [37] "additive\_symm" "kullback-leibler" "jeffreys"   
## [40] "k\_divergence" "topsoe" "jensen-shannon"   
## [43] "jensen\_difference" "taneja" "kumar-johnson"   
## [46] "avg"

##################################  
library(geosphere); # install.packages("geosphere", dependencies=TRUE);

## Warning: package 'geosphere' was built under R version 3.6.3

# Haversine formula is robust "crow-flies"  
distm( chicago[,2:1], fun=distHaversine); # form is "long,lat" so reverse

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0.000 1064.394 4750.102 1705.2299 2443.9749  
## [2,] 1064.394 0.000 4472.882 1058.2286 1631.1274  
## [3,] 4750.102 4472.882 0.000 3429.1556 3172.7211  
## [4,] 1705.230 1058.229 3429.156 0.0000 760.9385  
## [5,] 2443.975 1631.127 3172.721 760.9385 0.0000

distm( chicago[,2:1], fun=distMeeus); # form is "long,lat" so reverse

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0.000 1065.450 4739.559 1703.2212 2441.8548  
## [2,] 1065.450 0.000 4463.793 1055.9035 1628.0412  
## [3,] 4739.559 4463.793 0.000 3422.4607 3168.2795  
## [4,] 1703.221 1055.903 3422.461 0.0000 760.8433  
## [5,] 2441.855 1628.041 3168.279 760.8433 0.0000

distm( chicago[,2:1], fun=distGeo); # form is "long,lat" so reverse

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0.000 1065.450 4739.514 1703.2122 2441.8446  
## [2,] 1065.450 0.000 4463.754 1055.8934 1628.0279  
## [3,] 4739.514 4463.754 0.000 3422.4320 3168.2601  
## [4,] 1703.212 1055.893 3422.432 0.0000 760.8418  
## [5,] 2441.845 1628.028 3168.260 760.8418 0.0000

# default unit is meters, so let's convert  
library(measurements); # install.packages("measurements", dependencies=TRUE);  
conv\_unit(2.54, "cm", "inch");

## [1] 1

conv\_unit( distm( chicago[,2:1], fun=distHaversine), "m", "mi"); # meters to miles

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0.0000000 0.6613836 2.951576 1.0595808 1.5186156  
## [2,] 0.6613836 0.0000000 2.779320 0.6575528 1.0135356  
## [3,] 2.9515764 2.7793201 0.000000 2.1307785 1.9714375  
## [4,] 1.0595808 0.6575528 2.130779 0.0000000 0.4728253  
## [5,] 1.5186156 1.0135356 1.971438 0.4728253 0.0000000

conv\_unit( distm( chicago[,2:1], fun=distMeeus), "m", "mi"); # meters to miles

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0.0000000 0.6620396 2.945026 1.0583326 1.5172982  
## [2,] 0.6620396 0.0000000 2.773672 0.6561080 1.0116179  
## [3,] 2.9450256 2.7736723 0.000000 2.1266185 1.9686776  
## [4,] 1.0583326 0.6561080 2.126619 0.0000000 0.4727661  
## [5,] 1.5172982 1.0116179 1.968678 0.4727661 0.0000000

conv\_unit( distm( chicago[,2:1], fun=distGeo), "m", "mi"); # meters to miles

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0.0000000 0.6620398 2.944997 1.0583270 1.5172919  
## [2,] 0.6620398 0.0000000 2.773648 0.6561018 1.0116097  
## [3,] 2.9449972 2.7736480 0.000000 2.1266006 1.9686655  
## [4,] 1.0583270 0.6561018 2.126601 0.0000000 0.4727652  
## [5,] 1.5172919 1.0116097 1.968666 0.4727652 0.0000000

## that's cool, but this is the end-all "crow-flies" distance formula ... not manhattan  
### [+5 Easter] Can you get the accuracy of Haversine working with manhattan  
### See https://stackoverflow.com/questions/32923363/manhattan-distance-for-two-geolocations  
  
# actually, longitude is a function of latitude  
# https://gis.stackexchange.com/questions/142326/calculating-longitude-length-in-miles  
#   
# deg2rad = function(degrees)  
# {  
# degrees \* (pi/180);  
# }  
# rad2deg = function(radians )  
# {  
# radians \* (180/pi);  
# }  
#   
# computeLongitudeFromLatitude = function(latitude) # in decimal degrees  
# {  
# 1 / ( 69.172 \* cos(deg2rad(latitude)) );   
# }  
#   
# chicago$y.lat2 = computeLongitudeFromLatitude(chicago$latitude);  
#   
# chicago;  
#   
# dist(chicago[,3,5], method="manhattan", diag=FALSE, upper=TRUE);

# New York City area (e.g., Manhattan)

So you do Manhattan:

nyc.timesquare.latlong = c(40.7578705,-73.9854185);  
nyc.bull.wallstreet.latlong = c(40.705575,-74.0134097);  
nyc.lincoln.center.latlong = c(40.772, -73.9847);  
nyc.macys.latlong = c(40.7510547,-73.9904135);  
nyc.broadway.latlong = c(40.7593527,-73.9870634);  
nyc.stpatricks.latlong = c(40.758611, -73.976389);  
nyc.best.pizza.latlong = c(40.6250931,-73.9616134);  
nyc.best.cupcakes.latlong = c(40.7301048,-74.0026878);  
nyc.saks.latlong = c(40.7582027,-73.9772205);  
  
nyc = as.data.frame( matrix( c( nyc.timesquare.latlong,  
 nyc.bull.wallstreet.latlong,  
 nyc.lincoln.center.latlong,  
 nyc.macys.latlong,  
 nyc.broadway.latlong,  
 nyc.stpatricks.latlong,  
 nyc.best.pizza.latlong,  
 nyc.best.cupcakes.latlong,  
 nyc.saks.latlong  
 )  
 ,ncol=2,byrow=TRUE) );  
 rownames(nyc) = c("Times Square", "The Bull on WallStreet", "The Lincoln Center", "Macy's", "Broadway (Les Miserable)", "St. Patrick's", "Di Fara Pizza", "Molly's Cupcakes", "Saks 5th Avenue");  
 colnames(nyc) = c("latitude","longitude");  
  
nyc;

## latitude longitude  
## Times Square 40.75787 -73.98542  
## The Bull on WallStreet 40.70558 -74.01341  
## The Lincoln Center 40.77200 -73.98470  
## Macy's 40.75105 -73.99041  
## Broadway (Les Miserable) 40.75935 -73.98706  
## St. Patrick's 40.75861 -73.97639  
## Di Fara Pizza 40.62509 -73.96161  
## Molly's Cupcakes 40.73010 -74.00269  
## Saks 5th Avenue 40.75820 -73.97722

In this RNotebook, write brief responses to the questions:

#### *Question 1:* When would the “angle of rotation” for NYC matter to compute the Manhattan Distance?

#### *Question 2:* How does Di Fara Pizza complicate things? How would you really have to compute distance in this scenario (thing Google Maps “walking” or “driving directions”)?

#### *Question 3:* Which distance metric seems to be most conservative (overstating distance: Manhattan, Euclidean, Haversine)?

#### *Question 4:* Which Spherical Ellipsoid Distance is most accurate (Haversine, Meeus, or Geo)? How can you verify that?

# Store Locator

I did some work for a company called ‘organicgirl’ a few years back. We built a store locator as part of their brand presence online. It was in production for about 8 years. You can check out their “updated version” that doesn’t full operate.

The idea is that you input a ZIP code, we have a database that loosely maps that ZIP code to a latitude and longitude. Then you query a database to find other entities (e.g., stores) within a given radius.

Searching for a radius in a database is expensive. So it is easier to search on a square. After SQL gives you the result for a square, you manually compute the distances of each to the query input and reduce the result said to the circle (the inscribed circle in the square).

So I created a SQL sandbox and have populated it with a table of ZIPCODES for the USA and CANADA. It has 864,000 records (most are from CANADA oddly enough). I removed CANADA, and I believe the US-data is intack. About 42,000 records.

To keep the database connection information “PRIVATE”, please see the \_SECRET\_.txt file in the DROPBOX. Run that code from the “console” of RStudio below, but DO NOT store in the RNotebook. \_student\_access\_\_\unit\_01\_exploratory\_data\_analysis\week\_05

# this is something you would never want public normally  
# it is a sandbox, so let's give it a whirl ...  
  
Sys.setenv(DATABASE\_WSU\_SANDBOX = "wsu\_sandbox\_db");  
  
db.host = 'md5.mshaffer.com';  
db.name = Sys.getenv("DATABASE\_WSU\_SANDBOX");  
db.user = Sys.getenv("USER\_WSU\_SANDBOX");  
db.passwd = Sys.getenv("PASSWORD\_WSU\_SANDBOX");  
  
#install.packages("RMySQL"); library(RMySQL)  
  
# tidyverse has a SQL syntax structure, but RMySQL follows SQL syntax a bit.  
  
  
## This is set from the command console ... the one line of code is in the dropbox called "db  
library(RMySQL); # install.packages("RMySQL", dependencies=TRUE);

## Warning: package 'RMySQL' was built under R version 3.6.3

## Loading required package: DBI

mysql.connection = dbConnect(RMySQL::MySQL(),  
 user = db.user,  
 password = db.passwd,  
 dbname = db.name ,  
 host = db.host);  
  
  
db.table.zipcodes = "zipcodes";  
  
zipcode = '99163'; # CANADA allows strings for zipcodes, I removed, so only U.S.  
  
mysql.query.template = "SELECT \* FROM {tablename} WHERE zipcode = '{zipcode}';";  
mysql.query = gsub("{tablename}",db.table.zipcodes, mysql.query.template, fixed=TRUE);  
mysql.query = gsub("{zipcode}",zipcode, mysql.query, fixed=TRUE);  
  
mysql.query;

## [1] "SELECT \* FROM zipcodes WHERE zipcode = '99163';"

#result = dbSendQuery(mysql.connection, mysql.query);  
result = dbGetQuery(mysql.connection, mysql.query);  
  
result;

## zipcode latitude longitude city state\_long state  
## 1 99163 46.7655 -117.192 PULLMAN WASHINGTON WA

# these functions don't exist in R?  
deg2rad = function(degrees)  
 {  
 degrees \* (pi/180);  
 }  
rad2deg = function(radians )  
 {  
 radians \* (180/pi);  
 }  
  
  
radius.miles = 10;  
# let's build a box  
my.latitude = result$latitude[1];  
my.longitude = result$longitude[1];  
  
delta.latitude = radius.miles / 68.703 ;  
delta.longitude = radius.miles / (69.172 \* cos(deg2rad(my.longitude)));   
  
# 4 sides of the square ... CREATE A BOUNDING BOX  
latitude.lower = my.latitude - delta.latitude;  
latitude.upper = my.latitude + delta.latitude;  
  
longitude.lower = my.longitude - delta.longitude;  
longitude.upper = my.longitude + delta.longitude;  
  
## longitude signs are opposite of latitude, would that be different outside US?  
mysql.query.template = "SELECT \* FROM {tablename} WHERE latitude > {latitude.lower} AND latitude < {latitude.upper} AND longitude < {longitude.lower} AND longitude > {longitude.upper} ORDER BY zipcode ASC;";  
mysql.query = gsub("{tablename}",db.table.zipcodes, mysql.query.template, fixed=TRUE);  
mysql.query = gsub("{zipcode}",zipcode, mysql.query, fixed=TRUE);  
mysql.query = gsub("{latitude.lower}",latitude.lower, mysql.query, fixed=TRUE);  
mysql.query = gsub("{latitude.upper}",latitude.upper, mysql.query, fixed=TRUE);  
mysql.query = gsub("{longitude.lower}",longitude.lower, mysql.query, fixed=TRUE);  
mysql.query = gsub("{longitude.upper}",longitude.upper, mysql.query, fixed=TRUE);  
  
mysql.query;

## [1] "SELECT \* FROM zipcodes WHERE latitude > 46.619945948503 AND latitude < 46.911054051497 AND longitude < -116.875642287706 AND longitude > -117.508357712294 ORDER BY zipcode ASC;"

## database went away, so I need to connect again ... this should be a function  
## this "remote database connection" is always going to be slow ...   
## maybe consider HeidiSQL on your workstation, and connect via "localhost"  
## http://md5.mshaffer.com/WSU\_STATS419/zipcodes.sql  
  
mysql.connection = dbConnect(RMySQL::MySQL(),  
 user = db.user,  
 password = db.passwd,  
 dbname = db.name ,  
 host = db.host);  
  
result.neighbors = dbGetQuery(mysql.connection, mysql.query);  
  
result.neighbors;

## zipcode latitude longitude city state\_long state  
## 1 83843 46.7284 -116.968 MOSCOW IDAHO ID  
## 2 83844 46.7303 -116.997 MOSCOW IDAHO ID  
## 3 83872 46.8617 -116.976 VIOLA IDAHO ID  
## 4 99102 46.7963 -117.249 ALBION WASHINGTON WA  
## 5 99111 46.8789 -117.357 COLFAX WASHINGTON WA  
## 6 99163 46.7655 -117.192 PULLMAN WASHINGTON WA  
## 7 99164 46.7289 -117.156 PULLMAN WASHINGTON WA  
## 8 99165 46.7194 -117.184 PULLMAN WASHINGTON WA

# note: we have our "seed" (99163) in our result set.

#### TODO: Take this information and you compute the pair-wise distances using the best Spherical Ellipsoid method. Recall the focal zipcode (99163) are the distances we care about. Add as column to the dataframe result.neighbors$distance with the ppropriate values in miles. Add another column result.neighbors$incircle which is TRUE/FALSE depending on the distance provided as the input. Remove the specific row for the focal zipcode (99163) at the very end.

#### TODO: Choose a zipcode of your choice (not Pullman), and repeat everything above for that zipcode.

# String distances

Distance can also be applied to strings, based on various methods. For example, if I am typing the word the, I may accidently transpose some of hte characters.

library(RecordLinkage); # install.packages("RecordLinkage", dependencies=TRUE);

## Warning: package 'RecordLinkage' was built under R version 3.6.3

## Loading required package: RSQLite

## Warning: package 'RSQLite' was built under R version 3.6.3

##   
## Attaching package: 'RSQLite'

## The following object is masked from 'package:RMySQL':  
##   
## isIdCurrent

## Loading required package: ff

## Warning: package 'ff' was built under R version 3.6.3

## Loading required package: bit

## Warning: package 'bit' was built under R version 3.6.3

##   
## Attaching package: 'bit'

## The following object is masked from 'package:base':  
##   
## xor

## Attaching package ff

## - getOption("fftempdir")=="C:/Users/Gamer/AppData/Local/Temp/RtmpETfmux/ff"

## - getOption("ffextension")=="ff"

## - getOption("ffdrop")==TRUE

## - getOption("fffinonexit")==TRUE

## - getOption("ffpagesize")==65536

## - getOption("ffcaching")=="mmnoflush" -- consider "ffeachflush" if your system stalls on large writes

## - getOption("ffbatchbytes")==171316346.88 -- consider a different value for tuning your system

## - getOption("ffmaxbytes")==8565817344 -- consider a different value for tuning your system

##   
## Attaching package: 'ff'

## The following objects are masked from 'package:utils':  
##   
## write.csv, write.csv2

## The following objects are masked from 'package:base':  
##   
## is.factor, is.ordered

## RecordLinkage library

## [c] IMBEI Mainz

##   
## Attaching package: 'RecordLinkage'

## The following object is masked from 'package:bit':  
##   
## clone

## The following object is masked from 'package:base':  
##   
## isFALSE

w1.singular = "TRIANGLE"; sort( unlist(strsplit(w1.singular,"",fixed=TRUE)) );

## [1] "A" "E" "G" "I" "L" "N" "R" "T"

w1 = "TRIANGLES"; sort( unlist(strsplit(w1,"",fixed=TRUE)) );

## [1] "A" "E" "G" "I" "L" "N" "R" "S" "T"

w2 = "GNARLIEST"; sort( unlist(strsplit(w2,"",fixed=TRUE)) );

## [1] "A" "E" "G" "I" "L" "N" "R" "S" "T"

w3 = "RESLATING"; sort( unlist(strsplit(w3,"",fixed=TRUE)) );

## [1] "A" "E" "G" "I" "L" "N" "R" "S" "T"

# the number returned is bound between [0,1]  
jarowinkler(w1.singular, w1);

## [1] 0.9777778

jarowinkler(w1.singular, w2);

## [1] 0.6944444

jarowinkler(w1.singular, w3);

## [1] 0.5935185

jarowinkler(w1, w2);

## [1] 0.7566138

jarowinkler(w1, w3);

## [1] 0.5703704

#############  
  
levenshteinSim(w1.singular, w1);

## [1] 0.8888889

levenshteinSim(w1.singular, w2);

## [1] 0.1111111

levenshteinSim(w1.singular, w3);

## [1] 0.1111111

levenshteinSim(w1, w2);

## [1] 0.2222222

levenshteinSim(w1, w3);

## [1] 0

levenshteinDist(w1.singular, w1);

## [1] 1

levenshteinDist(w1.singular, w2);

## [1] 8

levenshteinDist(w1.singular, w3);

## [1] 8

levenshteinDist(w1, w2);

## [1] 7

levenshteinDist(w1, w3);

## [1] 9

#### Question: What do you notice about w1, w2, w3 when we sort the characters that compose each string? What is the term for this type of equality (e.g., ant and tan mixing up the same characters to form a new string that has its own meaning)? Is there a string function that would find such examples?

We will revisit string distances again. This was just a brief introduction to the idea.