Lab: Week 14

36-350 – Statistical Computing

Week 14 – Fall 2020

Name: Kimberly Zhang

Andrew ID: kyz

You must submit **your own** lab as a knitted PDF file on Gradescope.

This week's lab is like last week's lab: you will do your work "remotely" and cut and paste your answers into plain code blocks below:

```
This is a plain code block. Note the lack of a {r} above.
```

Question 1

(6 points)

Notes 12B (3)

Create a table dubbed `rdata` that has five columns: `id` (type `serial primary key`), `a` and `b` (both `text`), `moment` (`date`), and `x` (`real`). You need not add any constraints.

```
postgres=# create table rdata (
id serial primary key,
a text,
b text,
moment date,
x real
);
CREATE TABLE
```

Question 2

(6 points)

Notes 14A (2-3)

Use a `select` command with the `generate_series()` function to display the sequence 1 to 100. Have the column name in your final outputted table be `id`. (Again, you just need to display the first few lines of output. You can force this by having, e.g., `limit 5` as the last part of your command.)

```
postgres=# select id
from generate_series(1,100) as id
limit 5;
 id
----
  1
  2
  3
  4
```

```
  5
(5 rows)
```

Question 3

(6 points)

Notes 14A (2-3)

Use a `select` command to create (and display!) a random text string. One approach is to use the `md5()` function that has as its lone argument a random number that is cast to text. (Sounds weird, but it works.)

```
postgres=# select md5('10');
               md5
----------------------------------
 d3d9446802a44259755d38e6d163e820
(1 row)
```

Question 4

(6 points)

Notes 14A (2-3)

Use a `select` command to choose a random element from a fixed array of strings. One can obtain a fixed text array by writing out a strong for a literal `postgres` array (like `'{X,Y,Z}'`) and then telling `postgres` to cast this to text: `'{X,Y,Z}'::text[]`. One can randomly select an index in the resulting text array by combining the `ceil()` (i.e., ceiling) amd `random()` functions to make a selection. (Multiply the output of `random()` by the number of elements in the text array, then determine the ceiling: this yields, in this case, 1, 2, or 3.) (Note: `('{X,Y,Z}'::text[])[1]` would return `'X'`: like `R`, `SQL` is 1-indexed.)

```
postgres=# select ('{A, B, C}'::text[])[(random()*3)::integer];
 text
------
 C
(1 row)
```

Question 5

(6 points)

Notes 14A (2-3)

`select` a random date in 2020. You can do this by adding a random integer to the `date '2020-01-01'`. For instance, the expression `select '2020-01-01'::date + 7 as random_date;` will output January 8th. Here, replace the 7 with an expression that gives a random integer, converting a non-integer numeric type to an integer by appending `::integer`.

```
postgres=# select '2020-01-01'::date + (random()*365)::integer as random_date;
 random_date
-------------
 2020-08-04
(1 row)
```

Question 6

(10 points)

Notes 14A (6)

Now let's put this all together. Use `insert` to populate the `rdata` table with 100 rows, where the `id` goes from 1 to 100, `a` is random text, `b` is a random choice from a set of strings (at least three in size), `moment` contains random days in 2020, and `x` contains random real numbers in some range. Show the first five rows of your table.

```
postgres=# insert into rdata (a, b, moment, x)
(select md5(id::varchar(3)) as a,
('{X, Y, Z}'::text[])[(random()*3)::integer] as b, '2020-01-01'::date + (random()*365)::integer as momer
from generate_series(1,100) as id);
INSERT 0 100
postgres=# select * from rdata limit 5;
 id |                a                 | b |   moment   |     x
----+----------------------------------+---+------------+-----------
  1 | c4ca4238a0b923820dcc509a6f75849b | X | 2020-06-20 | 56.709507
  2 | c81e728d9d4c2f636f067f89cc14862c | Y | 2020-11-04 |  79.19716
  3 | eccbc87e4b5ce2fe28308fd9f2a7baf3 | Y | 2020-11-12 |  16.84765
  4 | a87ff679a2f3e71d9181a67b7542122c | Y | 2020-01-05 | 59.078167
  5 | e4da3b7fbbce2345d7772b0674a318d5 | Z | 2020-06-02 | 22.210258
(5 rows)
```

Question 7

(6 points)

Notes 14A (2-3)

The `~*` or `ilike` operators in `postgres` are used for pattern matching in text. The two operators use slightly different ways to specify the patterns. Use `select` with either to display the rows for which `a` matches a specific pattern that is at least four characters long and contains a mixture of numbers and letters. Note that you can use regexes here!

```
postgres=# select *
from rdata
where a ilike '%e4da%';
 id |                a                 | b |   moment   |     x
----+----------------------------------+---+------------+-----------
  5 | e4da3b7fbbce2345d7772b0674a318d5 | Z | 2020-06-02 | 22.210258
(1 row)
```

Question 8

(6 points)

Notes 14A (2-3)

Use `select` with the `overlaps` operator to find all rows whose `moment` lies in the month of November. The `overlaps` operator looks like this

```
(date1,date2) overlaps (date3,date4)
```

where the pairs of dates represent intervals of time. (The two dates in a pair can be equal. For instance, `date1` and `date2` should both be `moment`. Also, remember that one can use `::date` to cast a string to a `date` variable.)

```
postgres=# select *
from rdata
where (moment, moment) overlaps ('2020-11-01'::date, '2020-11-30'::date);
 id  |                a                 | b |   moment   |     x
-----+----------------------------------+---+------------+-----------
```

```
   2 | c81e728d9d4c2f636f067f89cc14862c | Y | 2020-11-04 |  79.19716
   3 | eccbc87e4b5ce2fe28308fd9f2a7baf3 | Y | 2020-11-12 |  16.84765
  12 | c20ad4d76fe97759aa27a0c99bff6710 | X | 2020-11-12 |  65.35675
  50 | c0c7c76d30bd3dcaefc96f40275bdc0a | X | 2020-11-19 | 17.172565
  60 | 072b030ba126b2f4b2374f342be9ed44 | X | 2020-11-11 | 78.150154
  67 | 735b90b4568125ed6c3f678819b6e058 | Y | 2020-11-07 | 91.936935
  76 | fbd7939d674997cdb4692d34de8633c4 | Y | 2020-11-06 |  75.97037
  84 | 68d30a9594728bc39aa24be94b319d21 | Y | 2020-11-05 | 72.466324
 100 | f899139df5e1059396431415e770c6dd | Z | 2020-11-15 |  45.62769
(9 rows)
```

Question 9

(6 points)

Notes 12B (6) + Notes 14A (2-3) + Notes 14B (3)

Use `update` to set the value of `b` to a fixed character in all rows that are divisible by 3 and by 5. Then select those rows to display them. (Hint: the modulo operator!)

```
postgres=# update rdata
set b = 'K'
where mod(id, 3) = 0
and mod(id, 5) = 0;
UPDATE 6
postgres=# select *
from rdata
where mod(id, 3) = 0
and mod(id, 5) = 0;
 id |                a                 | b |   moment   |     x
----+----------------------------------+---+------------+-----------
 15 | 9bf31c7ff062936a96d3c8bd1f8f2ff3 | K | 2020-10-09 |  6.488498
 30 | 34173cb38f07f89ddbebc2ac9128303f | K | 2020-03-26 |   81.3635
 45 | 6c8349cc7260ae62e3b1396831a8398f | K | 2020-09-28 |  24.48165
 60 | 072b030ba126b2f4b2374f342be9ed44 | K | 2020-11-11 | 78.150154
 75 | d09bf41544a3365a46c9077ebb5e35c3 | K | 2020-09-02 |  77.26934
 90 | 8613985ec49eb8f757ae6439e879bb2a | K | 2020-03-22 | 82.686104
(6 rows)
```

Question 10

(6 points)

Notes 12B (8)

Use `delete` to remove all rows for which `id` is even and greater than 2. Show the output that indicates that 49 rows were deleted.

```
postgres=# delete from rdata
where mod(id, 2) = 0
and id > 2;
DELETE 49
```

Issue the following commands in your `postgres` session:

```
drop table rdata;
```

```
\cd [PATH TO DIRECTORY WITH GalaxyStatistics.txt FILE FROM WEEK 12]

create table galaxies (
  field varchar(6),
  gini numeric(4,3),
  conc numeric(4,3)
);

\copy galaxies from 'GalaxyStatistics.txt' with ( format csv, header, delimiter ' ' );
```

Why are we using `numeric(4,3)` here? Because there is no pertinent scientific information beyond the thousandths place for either `gini` or `conc`. Note that if we didn't use this data type, but say `real` instead, we could always use the `round()` function to round off values in tables to, e.g., three decimal places. We will still have to round output from functions, however!

Question 11

(6 points)

Notes 14B (3,5)

Show the range of observed values for both the Gini coefficient and the concentration statistic. Give the output columns the names `gini_range` and `conc_range` respectively. We don't need to round the output from `min()` and `max()` as the output is of the same data type as the input.

```
postgres=# select max(gini) - min(gini) as gini_range,
max(conc) - min(conc) as conc_range
from galaxies;
 gini_range | conc_range
------------+------------
      0.910 |      5.086
(1 row)
```

Question 12

(6 points)

Notes 14B (3,7)

One rule-of-thumb for estimating the sample standard deviation for a set of data is to take the range and divide it by four. Try that here: output both the sample standard deviation and the "rule of four" estimate for the concentration column. You should observe that the "rule of four" doesn't work very well here. Round your answers to three decimal places.

```
postgres=# select round((max(gini) - min(gini))/4,3) as gini_rule4,
round(stddev(gini),3) as gini_stddev,
round((max(conc) - min(conc))/4,3) as conc_rule4,
round(stddev(conc),3) as conc_stddev
from galaxies;
 gini_rule4 | gini_stddev | conc_rule4 | conc_stddev
------------+-------------+------------+-------------
      0.228 |       0.082 |      1.272 |       0.560
(1 row)
```

Question 13

(6 points)

Notes 14B (7)

Review, off-line, the relationship between the covariance of two random variables and the correlation coefficient for those same two variables. Then, below, show both the correlation coefficient between the Gini coefficient and the concentration statistic as computed directly with a `postgres` function, and as computed using the covariance, etc. As usual, round your numbers to three decimal places. (Note the following: `round()` requires its input to be of type `numeric`, while the output of `corr()`, for example, is of type `double precision`. So you'll sometimes you'll have to cast the output of one function to `numeric` in order to get rounding to work.)

```
postgres=# select round(corr(gini, conc)::numeric, 3) as direct_corr,
round((covar_samp(gini,conc)/(stddev(gini)*stddev(conc)))::numeric, 3)
as covar_corr
from galaxies;
 direct_corr | covar_corr
-------------+------------
       0.431 |      0.431
(1 row)
```

Question 14

(6 points)

Use the `mode()` function, not covered in the notes, to determine which of the sky fields is most represented in the `galaxies` table. You'll need to look up the documentation for `mode()` to determine actually how to use this.

```
postgres=# select mode() within group (order by field) as most_rep_field
from galaxies;
 most_rep_field
----------------
 COSMOS
(1 row)
```

Question 15

(6 points)

Notes 14B (6)

Compute the interquartile range for the Gini coefficient. To be clear: the output from your `postgres` code should be one number that represents the difference between the 25^{th} and 75^{th} percentiles of the data. (Don't output two numbers and compute the difference by hand.) Call your output data column `iqr`. Round your answer to four decimal places.

```
postgres=# select round((percentile_cont(0.75) within group (order by gini) -
percentile_cont(0.25) within group (order by gini))::numeric,4) as iqr
from galaxies;
  iqr
--------
 0.0770
(1 row)
```

Question 16

(6 points)

Notes 14B (6-7)

Compute the median value of the concentration statistic. Then regress the Gini coefficient upon the concentration statistics for all values of concentration less than the median, and output the regression model slope. Note: we are expecting that you compute the median first, then hardwire that number into the subsequent calculation. Round output to three decimal places. You should get 0.041.

```
postgres=# select round((regr_slope(gini, conc))::numeric,3) as slope
from galaxies
where conc <
(select percentile_cont(0.5) within group (order by conc) from galaxies);
 slope
-------
 0.041
(1 row)
```