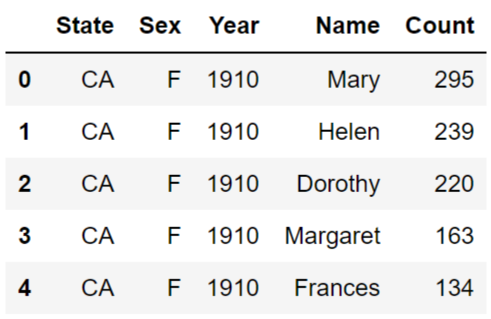
Data 100, Spring 2023

Discussion #3

Data Cleaning and EDA

1. Consider the following *sample* of the baby names table obtained by using `baby_names.sample(5)`.



	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

(a) Consider the name, year, and count variables. What type of variable are each one? (Quantitative discrete, Quantitative continuous, Qualitative nominal, Qualitative ordinal)

(b) Which of the following is a (minimal) primary key for this table (given the sample of the data and reasonable assumptions about what might be in the rest of the data)?
(A *primary key* is the column or minimal set of columns that uniquely determines the values in all the remaining columns. This is a statement about the table structure and should hold for all data that could be put in the table.)

☐ A. State and Sex
☐ B. Year and Name
☐ C. State, Year, and Name
☐ D. State, Sex, Year, and Name
☐ E. State, Sex, Year, Name, and Count

(c) Mark the following statements as True or False and justify why.
A *foreign key* is a set of 1+ columns in a table that refers to the primary key of another table.

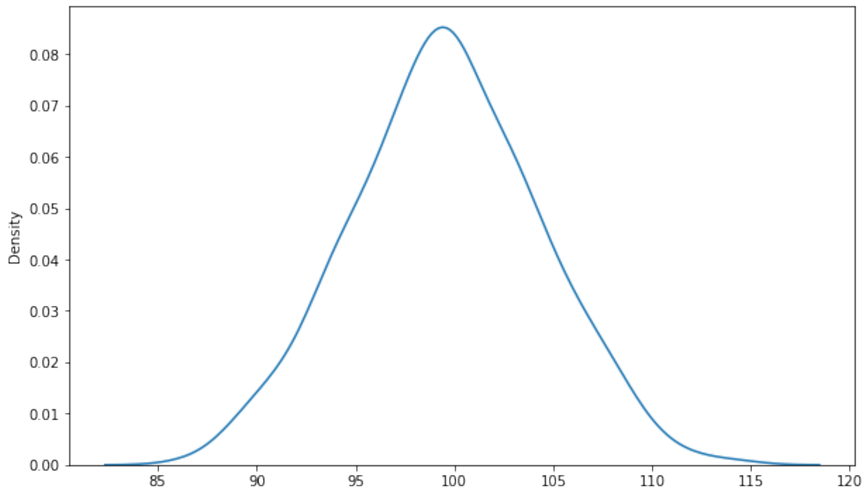
1. (T / F) The foreign key of a table referencing the `babynames` table must always reference the primary key from part (b).

2. (T / F) All tables with a foreign key referencing the `babynames` table must always join or merge on the primary key columns from part (b).
3. (T / F) `babynames` has a primary/foreign key relationship with the `elections` dataset containing the first name and full name of every presidential candidate.
4. (T / F) To join or merge `babynames` with itself, we must join or merge on its primary key columns.

Dealing with Missing Data

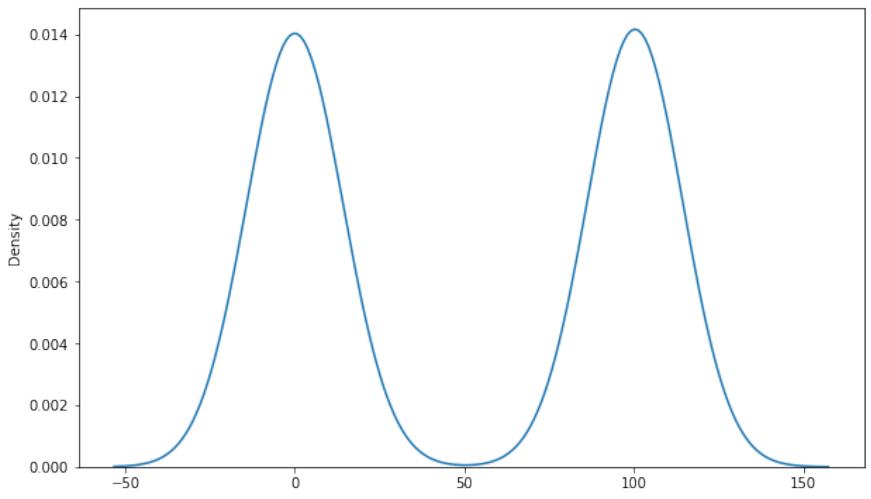
2. While exploring a climate dataset with a million records, you realize that a portion of atmospheric measurements in different fields are NaN values! You decide to impute or drop these missing values before continuing your EDA. Given the empirical distribution of each of the below atmospheric variables, determine how to solve the missing data problem. Variable distributions are constructed from all valid (non-NaN) values for a given field.

   (a) Suppose that you plot the distribution of one atmospheric variable, given below. Which of the following techniques would be **most** effective in solving this issue of missing data? (Note: We visualize our empirical distribution using `df['votes'].plot.kde()`, which a produces a smooth curve. More next week!)



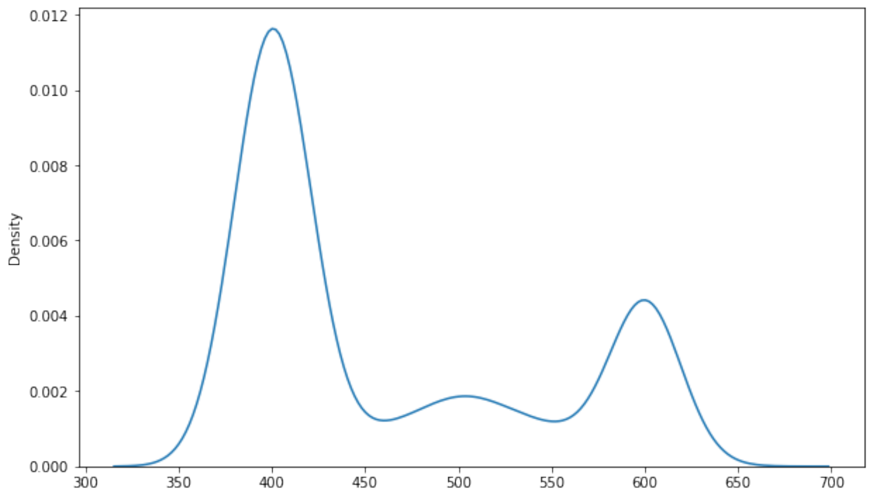
   ☐ A. Using the mean to impute the missing values
   ☐ B. Using the mode to impute the missing values
   ☐ C. Using the median to impute the missing values
   ☐ D. Dropping any rows with missing values
   ☐ E. Imputing missing values with zero

   (b) Suppose that the distribution of a second atmospheric variable is given by the following figure. Which of the following techniques would be **most** effective in solving this issue of missing data?



☐ A. Using the mean to impute the missing values
☐ B. Using the mode to impute the missing values
☐ C. Using the median to impute the missing values
☐ D. Dropping any rows with missing values
☐ E. Imputing missing values with zero

(c) Suppose that the distribution of a third atmospheric variable is given by the following figure. Which of the following techniques would be **reasonably** effective in solving this issue of missing data?



☐ A. Using the mean to impute the missing values
☐ B. Using the mode to impute the missing values
☐ C. Using the median to impute the missing values
☐ D. Dropping any rows with missing values
☐ E. Imputing missing values with zero

Regular Expressions

Here's a complete list of metacharacters:

```
. ^ $ * + ? { } [ ] \ | ( )
```

Some reminders on what each can do (this is not exhaustive):

"^" matches the position at the beginning of string (unless used for negation `"[^]"`)

"$" matches the position at the end of string character.

"?" match preceding literal or sub-expression 0 or 1 times.

"+" match preceding literal or sub-expression *one* or more times.

"*" match preceding literal or sub-expression *zero* or more times

"." match any character except new line.

"[ ]" match any one of the characters inside, accepts a range, e.g., `"[a-c]"`.

"( )" used to create a sub-expression

"\d" match any *digit* character. `"\D"` is the complement.

"\w" match any *word* character (letters, digits, underscore). `"\W"` is the complement.

"\s" match any *whitespace* character including tabs and newlines. `\S` is the complement.

"*?" Non-greedy version of *. Not fully discussed in class.

"\b" match boundary between words. Not discussed in class.

"+?" Non-greedy version of +. Not discussed in class.

"{m,n}" The preceding element or subexpression must occur between m and n times, inclusive.

Some useful `re` package functions:

`re.split(pattern, string)` split the `string` at substrings that match the `pattern`. Returns a list.

`re.sub(pattern, replace, string)` apply the `pattern` to `string` replacing matching substrings with `replace`. Returns a string.

`re.findall(pattern, string)` Returns a list of all matches for the given `pattern` in the `string`.

Regular Expressions

3. Which strings contain a match for the following regular expression, `"1+1$"`? The character `"␣"` represents a single space.

   ◯ A. `What␣is␣1+1` ◯ B. `Make␣a␣wish␣at␣11:11` ◯ C. `111␣Ways␣to␣Succeed`

4. Write a regular expression that matches a string which contains only one word containing only lowercase letters and numbers (including the empty string).

5. Given `sometext = "I've␣got␣10␣eggs,␣20␣gooses,␣and␣30␣giants."`, use `re.findall` to extract all the items and quantities from the string. The result should look like **`['10 eggs', '20 gooses', '30 giants']`**. You may assume that a space separates quantity and type, and that each item ends in s.

6. For each pattern specify the starting and ending position of the first match in the string. The index starts at zero and we are using closed intervals (both endpoints are included).

	`abcdefg`	`abcs!`	`ab␣abc`	`abc,␣123`
`abc*`	$[0, 2]$	______	______	______
`[^\s]+`	______	______	______	______
`ab.*c`	______	______	______	______
`[a-z1,9]+`	______	______	______	______

7. *(Bonus)* Given the following text in a variable `log`:

```
169.237.46.168 - - [26/Jan/2014:10:47:58 -0800]
"GET␣/stat141/Winter04/␣HTTP/1.1" 200 2585
"http://anson.ucdavis.edu/courses/"
```

   Fill in the regular expression in the variable `pattern` below so that after it executes, `day` is 26, `month` is Jan, and `year` is 2014.

```
pattern = ...
matches = re.findall(pattern, log)
day, month, year = matches[0]
```

8. *(Bonus)* Given that `sometext` is a string, use `re.sub` to replace all clusters of non-vowel characters with a single period. For example `"a␣big␣moon,␣between␣us..."` would be changed to `"a.i.oo.e.ee.u."`.

9. *(Bonus)* Given the text:

```
"<record>␣Josh␣Hug␣<hug@cs.berkeley.edu>␣Faculty␣</record>"
"<record>␣Lisa␣Yan␣<lisa.yan@berkeley.edu>␣Instructor␣</record>"
```

   Which of the following matches exactly to the email addresses (including angle brackets)?
   ◯ A. `<.*@.*>` ◯ B. `<[^>]*@[^>]*>` ◯ C. `<.*@\w+\..*>`