Intermediate Data Science

Data Cleaning and Preparation

Joanna Bieri DATA201

Important Information

- Email: joanna_bieri@redlands.edu
- Office Hours take place in Duke 209 – Office Hours Schedule
- Class Website
- Syllabus

Data Cleaning

Often in data science a huge portion of your time will be spent loading, cleaning, transforming, and rearranging data. Here are the main topics we will cover:

1. Handling Missing Data
2. Transforming Data
3. String Manipulation
4. Categorical Data

Handling Missing Data

Missing data can happen for a wide variety of reasons:

- The data really does not exist for a certain observation: In population data babies would not have a date of marriage or a list of children.
- The data was improperly entered: Errors are easy to make.
- The data set was damaged: Reading or writing issues happen.
- The missing data (None, NaN, or NA) means something important: Maybe a student did not take a test and that is important in your analysis.

Handling Missing Data

Pandas uses floating point NaN (Not a Number) to represent missing data.

`np.nan`

this object has type float.

Methods to handle missing data:

`isna()` / `isnull()`

Detect missing values.

```
df.isna()
df['column'].isnull()
```

Methods to handle missing data:

`notna()` / `notnull()`

Detect non-missing values.

```
df.notna()
df['column'].notnull()
```

Methods to handle missing data:

`dropna()`

Remove missing values.

```
df.dropna()  # Drop rows with any NA values
df.dropna(axis=1)  # Drop columns with any NA values
```

Methods to handle missing data:

`fillna()`

Fill NA values with a specified value or method.

```
df.fillna(0)  # Replace NA with 0
```

Methods to handle missing data:

`replace()`

Replace specified values including NA.

```
df.replace(to_replace=np.nan, value=0)
```

BEWARE OF PYTHON mutable!

When you are saving data from one list to the next you should be very careful about how you do that! Python lists are **mutable** this means that when you set one list equal to another it does not make a new copy in memory, instead it copies a reference. Here is an example:

BEWARE OF PYTHON mutable!

```
list1 = [5,4,3,2,1]
list2 = list1

print('Here is list2, it looks like a copy!')
print(list2)

print('Now we will change something in list2')
list2[0] = 10
print(list2)

print('Now look at list1')
print(list1)
```

BEWARE OF PYTHON mutable!

```
Here is list2, it looks like a copy!
[5, 4, 3, 2, 1]
Now we will change something in list2
[10, 4, 3, 2, 1]
Now look at list1
[10, 4, 3, 2, 1]
BUT WE DIDN"T CHANGE LIST1 !!!!! WHY DID IT CHANGE????
```

BEWARE OF PYTHON mutable!

Use `.copy()`

```
list2 = list1.copy() ## THIS IS OUR ONLY CHANGE
```

```
Here is list2, it looks like a copy!
[5, 4, 3, 2, 1]
Now we will change something in list2
[10, 4, 3, 2, 1]
Now look at list1
[5, 4, 3, 2, 1]
List1 did not change
```

Moral of the mutable story

If you are creating a new variable by setting it equal to another list and you want to make changes to one without changing the other you should use .copy(). This is true of all mutable python types:

- list
- dict
- set
- pd.DataFrame
- pd.Series
- np.array

Filtering out Missing Data

You want to be careful and intentional when filtering out missing data. We will explore the process with a DataFrame that contains lots of missing data.

```
data = pd.DataFrame([[1., 6.5, 3.], [ np.nan, np.nan, 1.],
                     [np.nan, np.nan, np.nan], [np.nan, 6.5, 3
data
```

	0	1	2
0	1.0	6.5	3.0
1	NaN	NaN	1.0
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Filtering out Missing Data

```
data.dropna()
```

	0	1	2
0	1.0	6.5	3.0

```
data.dropna(how='all')
```

	0	1	2
0	1.0	6.5	3.0
1	NaN	NaN	1.0
3	NaN	6.5	3.0

Filtering out Missing Data

```
data.dropna(thresh=2)
```

	0	1	2
0	1.0	6.5	3.0
3	NaN	6.5	3.0

Filtering out Missing Data

Notice that each of these decisions creates a very different result! You should also notice that some optional commands are pretty common:

- `axis=0` do the calculation to the rows - usually default
- `axis=1` do the calculation to the columns

Filling in Missing Data

Most of the time you will use `.fillna()` but there are some nice optional arguments that let you customize the command. Remember, to make changes in memory you need to add `inplace=True`.

	0	1	2
0	1.0	6.5	3.0
1	NaN	NaN	1.0
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Filling in Missing Data

```
data.fillna(0)
data.fillna({0:np.nan,1:'Hello',2:0})
data.fillna(data.mean())
```

Data Transformation

Next we will talk a bit more about cleaning data:

1. Removing Duplicate Data
2. Replacing Data
3. Renaming
4. Discretizing and Binning
5. Outliers
6. Sampling
7. Dummy Variables

Duplicate Data

Sometimes data sets will have duplicate variables, maybe they are not identical but the represent the same thing. Here column k1 has the words and k2 has the numerical values:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

Duplicate Data

Notice that observations 5 and 6 are identical! We can check for this using the `.duplicated()` and `drop_duplicates()` command.

```
data.duplicated()
```

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Duplicate Data

```
data.drop_duplicates()
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

Duplicate Data

NOTE: You can also drop duplicates using just a subset of the columns:

```
data.drop_duplicates(subset=['k2','v2'])
```

Transforming Data and Mapping

We will imagine that you have some data that tells you about the amount of different types of fruit. Say you want to add a new column to this data that says what kind of fruit each one is: citrus, berry, tropical, or pome. You can use a dictionary and the `.map()` function to add this information.

Transforming Data and Mapping

	food	ounces
0	orange	4.0
1	blueberry	-999.0
2	orange	12.0
3	banana	6.0
4	strawberry	7.5
5	orange	8.0
6	banana	-999.0
7	apple	5.0
8	blackberry	6.0

Transforming Data and Mapping

```
# Here is a dictionary mapping fruits to categories
food_to_category = {
    "orange": "citrus",
    "blueberry": "berry",
    "strawberry": "berry",
    "blackberry": "berry",
    "banana": "tropical",
    "apple": "pome"
}
```

Transforming Data and Mapping

```
# Now we will add the column
data['category'] = data['food'].map(food_to_category)
```

	food	ounces	category
0	orange	4.0	citrus
1	blueberry	-999.0	berry
2	orange	12.0	citrus
3	banana	6.0	tropical
4	strawberry	7.5	berry
5	orange	8.0	citrus
6	banana	-999.0	tropical
7	apple	5.0	pome
8	blackberry	6.0	berry

Replacing Values

We have seen above how to replace NaN values, but what if there were other types of things you wanted to replace in the dataset? The `.replace()` function can replace any data you want with replacement data.

```
data.replace('tropical','berry',inplace=True)
```

	food	ounces	category
0	orange	4.0	citrus
1	blueberry	-999.0	berry
2	orange	12.0	citrus
3	banana	6.0	berry
4	strawberry	7.5	berry
5	orange	8.0	citrus
6	banana	-999.0	berry
7	apple	5.0	pome

Renaming

There are lots of ways to rename things on both the column labels and the index labels in a data frame. Here are a few examples of doing this. Lets start with this data set:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

Renaming

We could rename the columns directly:

```
data.rename(columns = {'three':'THREE', 'four':'FOUR'}, inplac
```

We could rename the indexes directly:

```
data.rename(index = {'Ohio':'California'}, inplace=True)
```

	one	two	THREE	FOUR
California	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

Renaming

We could also define a function that updates the names and the map() it to the index or columns.

```
def new_names(x):
    return x[:4].upper()

data.index = data.index.map(new_names)
```

	one	two	THREE	FOUR
CALI	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

Discretization and Binning

Sometimes you will want to take continuous data and represent it as bins.

For example, maybe you want high, medium, and low income classes. This is where binning will help.

Imagine you have a list of ages and you want to create age categories:

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
bins = [18, 25, 35, 60, 100]
age_categories = pd.cut(ages, bins)
```

Discretization and Binning

	age	range
0	20	(18, 25]
1	22	(18, 25]
2	25	(18, 25]
3	27	(25, 35]
4	21	(18, 25]
5	23	(18, 25]
6	37	(35, 60]
7	31	(25, 35]
8	61	(60, 100]
9	45	(35, 60]
10	41	(35, 60]
11	32	(25, 35]

Discretization and Binning

You can also get the cagegory codes - a numerical category

```
data['code'] = age_categories.codes
```

	age	range	code
0	20	(18, 25]	0
1	22	(18, 25]	0
2	25	(18, 25]	0
3	27	(25, 35]	1
4	21	(18, 25]	0
5	23	(18, 25]	0
6	37	(35, 60]	2
7	31	(25, 35]	1
8	61	(60, 100]	3
9	45	(35, 60]	2
10	41	(35, 60]	2

Discretization and Binning

In the range column the notation that you see is:

- (means inclusive
- [means exclusive

so you would read the range $(18, 25]$ to be ages 18 but less than 25.

Detecting Outliers

Sometimes you want to be able to detect outliers in a dataset, however this process can take a variety of operations and is highly dependent on how you define outliers in your data. There are two examples in the class notes.

Permutation and Random Sampling

Often when doing a data science project you will want to take random samples of your data. This might be to help you avoid bias in the ordering of your data. It might be to create model training and testing data sets. Or maybe your data set is too big and you want to start with a smaller subset of the data. There are lots of ways to do this:

Permutation and Random Sampling

- NUMPY - has `random.permutation()` which will give you a list of integers in a range that are permuted (rearranged) randomly.
- PANDAS - has a function `.sample()` that can take a sample from a DataFrame or series.
- Other Packages - later this semester we will see other packages like sklearn that can create test-train splits of your data.

Permutation and Random Sampling

Here is an example data set

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

Permutation and Random Sampling

Now lets get a permutation of the rows - mix them up!

```
num_rows = 5
sample = np.random.permutation(num_rows)
df.take(sample)
```

```
[3 4 0 2 1]
```

	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
0	0	1	2	3	4	5	6
2	14	15	16	17	18	19	20
1	7	8	9	10	11	12	13

Permutation and Random Sampling

We could also permute the columns

```
num_cols = df.shape[1]
sample = np.random.permutation(num_cols)
df.take(sample, axis=1)
```

```
[1 2 3 6 5 4 0]
```

	1	2	3	6	5	4	0
0	1	2	3	6	5	4	0
1	8	9	10	13	12	11	7
2	15	16	17	20	19	18	14
3	22	23	24	27	26	25	21
4	29	30	31	34	33	32	28

Permutation and Random Sampling

Pandas has the ability to sample the data. You can choose the number of samples n and decide if you want to allow for rows to be sampled again replace=True.

```
df.sample(n=3)
df.sample(n=30, replace=True)
```

Dummy Variables

Dummy variables are variables that take the place of something in your data set. They are especially useful for classifying categorical data. In these cases you replace a column with categories with several columns of 0 or 1 to represent whether or not (True/False) the observation belongs to the category.

Dummy Variables

Lets say we have some categorical data that we want to interact with numerically. Below you will see the key column that contains a,b,c. Lets generate dummy variables for this data.

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

Dummy Variables

```
dummies = pd.get_dummies(df["key"])
```

	a	b	c
0	False	True	False
1	False	True	False
2	True	False	False
3	False	False	True
4	True	False	False
5	False	True	False

Dummy Variables

```
dummies = pd.get_dummies(df["key"],dtype=float)
```

	a	b	c
0	0.0	1.0	0.0
1	0.0	1.0	0.0
2	1.0	0.0	0.0
3	0.0	0.0	1.0
4	1.0	0.0	0.0
5	0.0	1.0	0.0

Dummy Variables

```
dummies = pd.get_dummies(df["key"],dtype=int)
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

Dummy Variables

we can use the `.join()` function to add the dummies to our data frame:

```
df_new = df.join(dummies)
```

	key	data1	a	b	c
0	b	0	0	1	0
1	b	1	0	1	0
2	a	2	1	0	0
3	c	3	0	0	1
4	a	4	1	0	0
5	b	5	0	1	0

Dummy Variables

Here is a slightly more complicated example.

These files contain 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000. Thanks to Shyong Lam and Jon Herlocker for cleaning up and generating the data set. See README for more information

Dummy Variables

	movie_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Com
1	2	Jumanji (1995)	Adventure|Children's|Fant
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children's
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

Dummy Variables

```
dummies = movies['genres'].str.get_dummies("|")
```

	Action	Adventure	Animation	Children's	Comedy	Crime	Doc
0	0	0	1	1	1	0	0
1	0	1	0	1	0	0	0
2	0	0	0	0	1	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	1	0	0
...	...	...	...	...	...	...	...
3878	0	0	0	0	1	0	0
3879	0	0	0	0	0	0	0
3880	0	0	0	0	0	0	0
3881	0	0	0	0	0	0	0
3882	0	0	0	0	0	0	0

Dummy Variables

Notice that this looks at the 'genres' element, breaks up the string by the | character, and then assigns it a 1 in any category that it belongs to. If we look at the first row, which represents Toy Story, we can see this movie belongs to the categories: Animation, Children's, Comedy.

String Manipulation

One of the most common things you will have to do is interact with data that is formatted as strings (words). This can happen because of the way you saved your data, maybe everything got turned into strings, or because of the way the data was originally entered.

1. Simple strings to numbers
2. Object methods

String Manipulation

Anything that is entered into python with quotes or read in as a string will be assumed to be a string. Even though the number below is clearly, to our human minds, a number, Python sees it as a word.

```
number = '3'
int(number)
float(number)
str(3.0)
```

String Manipulation

If we start with a more complicated string there are lots of things we can do to alter it. The `.split()` function can split up a string how ever you want! It turns the string into a list of substrings.

```
val = "a,b,  guido"
val.split(",")
```

```
['a', 'b', '  guido']
```

String Manipulation

We can strip off the white space that remains:

```
string_list = val.split(',')
new_string_list = [x.strip() for x in string_list]
```

```
['a', 'b', '  guido']
['a', 'b', 'guido']
```

We can join the pieces together with a common character:

```
'.'.join(new_string_list)
```

```
'a.b.guido'
```

String Manipulation

There are so many different string methods! Here are some of my favorites:

- `.replace(old text, new text)` replace text
- `.rstrip()` strip from the right end of the string
- `.lstrip()` strip from the left end of the string
- `.lower()` make the string all lower case
- `.upper()` make the string all upper case
- `.title()` capitalize each first letter

String Manipulation

For more advanced string manipulation you can use regex.

```
import re
```

look in the book or online for more information.

Many of the string methods are also implemented directly in pandas and can be applied directly to Series data.

Categorical Data

Pandas has a build in data type called Categorical. This helps encode certain columns or parts of your data as being categorical, rather than just an assortment of strings. Categorical has the advantage of being better for memory and for sorting and organizing data.

Our data will have two city columns one that is regular and one categorical

```
df['city_cat'] = df['city'].astype('category')
```

	city	city_cat
0	Chicago	Chicago
1	Los Angeles	Los Angeles
2	Chicago	Chicago
3	New York	New York
4	New York	New York
...	...	...

Categorical Data

Lets look at the memory usage of each column:

```
print(df['city'].memory_usage(deep=True))
print(df['city_cat'].memory_usage(deep=True))
```

```
57667564
1000413
```

Categorical Data

We can also assign levels to categorical data. By default if you compare two strings with a great or less than the comparison will be alphabetical. What what if you want reverse alphabetical or you want to compare “high”,“medium”,“low”?

```
# Lets get the list of cities and
# order them reverse alphabetically
levels =
list(df['city'].value_counts().keys().sort_values(ascending=Fa

# Tell pands what the levels are
df['city_cat_levels'] =
pd.Categorical(df['city'], categories=levels, ordered=True)

# Now compare
print(df['city_cat_levels'] > 'Los Angeles')
# True for 'Chicago'
```