Department of Data Science - Data and Visual Analytics Lab

Lab6. Pandas Data Cleaning

Objectives

In this lab, you will learn how to

- Clean Column Names
- Converte String Columns to Numeric
- Remove Non-Digit Characters
- Convert Columns to Numeric Dtypes
- Rename Columns
- Extract Values from Strings
- Drop Missing Values
- Fill Missing Values

Data Cleaning Steps

Verify the contents with .head() method

```
import pandas as pd
df = pd.read_csv('path_to_data')
df.head(10)
```

See the names and types of the columns. Most of the time you're going get data that is not quite what you expected, such as dates which are actually strings

```
#Get column names
column_names = df.columns
print(column_names)

#Get column data types
df.dtypes

#Also check if the column is unique
for i in column_names:
   print('{} is unique: {}'.format(i, df[i].is_unique))
```

Now let's see if the dataframe has an index associated with it, by calling .index on the df.

```
# Check the index values
df.index.values

# Check if a certain index exists
'foo' in df.index.values

# If index does not exist
df.set_index('column_name_to_use', inplace=True)
```

Now, let's figure out which columns you want to keep or remove. We want to remove the columns in indexes 1, 3, and 5

```
# Create list comprehension of the columns you want to lose
columns_to_drop = [column_names[i] for i in [1, 3, 5]]
```

```
#Drop unwanted columns
df.drop(columns_to_drop, inplace=True, axis=1)
```

The **inplace=True** has been added so you don't need to save over the original df by assigning the result of .drop() to df.

What To Do With NaN

If you need to fill in errors or blanks, use the fillna() and dropna() methods. It seems quick, but all manipulations of the data should be documented so you can explain them to someone at a later time.

You could fill the NaNs with strings, or if they are numbers you could use the mean or the median value. There is a lot of debate on what do with missing or malformed data, and the correct answer is ... it depends.

You'll have to use your best judgement and input from the people you're working with on why removing or filling the data is the best approach.

```
# Fill NaN with ' '
df['col'] = df['col'].fillna(' ')

#Fill NaN with 99
df['col'] = df['col'].fillna(99)

#Fill NaN with the mean of the column
df['col'] = df['col'].fillna(df['col'].mean())
```

You can also propagate non-null values forward or backwards by putting method='pad' as the method argument. It will fill the next value in the dataframe with the previous non-NaN value. Maybe you just want to fill one value (limit=1)or you want to fill all the values. Whatever it is make sure it is consistent with the rest of your data cleaning

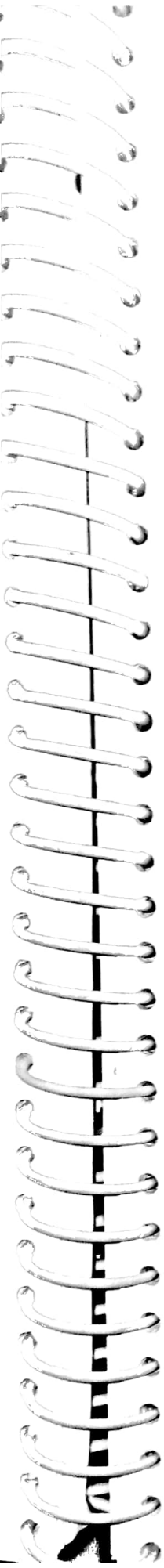
```
df = pd.DataFrame(data={'col1':[np.nan, np.nan, 2,3,4, np.nan,
np.nan]})

     col1
0    NaN
1    NaN
2    2.0
3    3.0
4    4.0   # This is the value to fill forward
5    NaN
6    NaN

df.fillna(method='pad', limit=1)

     col1
0    NaN
1    NaN
2    2.0
3    3.0
4    4.0
5    4.0 # Filled forward
6    NaN
```

Notice how only index 5 was filled? If I had not filled limited the pad , it would have filled the entire dataframe. We are not limited to forward filling, but also backfilling with bfill.



Fill the first two NaN values with the first available value

```
df.fillna(method='bfill')
```

```
     col1
0    2.0 # Filled
1    2.0 # Filled
2    2.0
3    3.0
4    4.0
5    NaN
6    NaN
```

You could just drop them from the dataframe entirely, either by the row or by the column.

```
# Drop any rows which have any nans
df.dropna()

# Drop columns that have any nans
df.dropna(axis=1)

# Only drop columns which have at least 90% non-NaNs
df.dropna(thresh=int(df.shape[0] * .9), axis=1)
```

np.where

Consider if you're evaluating a column, and you want to know if the values are strictly greater than 10. If they are you want the result to be 'foo' and if not you want the result to be 'bar'

```
# Follow this syntax
  np.where(if_this_condition_is_true, do_this, else_this)

  # Example
  df['new_column'] = np.where(df[i] > 10, 'foo', 'bar)
```

You're able to do more complex operations like the one below. Here we are checking if the column record starts with foo and does not end with bar. If this checks out we will return True else we'll return the current value in the column.

```
df['new_column'] = np.where(df['col'].str.startswith('foo') and
                            not df['col'].str.endswith('bar'),
                            True, df['col'])
```

And even more effective, you can start to nest your np.where so they stack on each other. Similar to how you would stack ternary operations, make sure they are readable as you can get into a mess quickly with heavily nested statements.

```
# Three level nesting with np.where
  np.where(if_this_condition_is_true_one, do_this,
    np.where(if_this_condition_is_true_two, do_that,
      np.where(if_this_condition_is_true_three, do_foo, do_bar)))

#A trivial example
df['foo'] = np.where(df['bar'] == 0, 'Zero',
                np.where(df['bar'] == 1, 'One',
                    np.where(df['bar'] == 2, 'Two', 'Three')))
```

Assert and Test What You Have

Just because you have your data in a nice dataframe, no duplicates, no missing values, you still might have some issues with the underlying data. And, with a dataframe of 10M+ rows or new API, how can you make sure the values are exactly what you expect them to be?

Truth is, you never really know if your data is correct until you test it. Best practices in software engineering rely heavily on testing their work, but for data science it is still a work in progress. Better to start now and teach yourself good work principles, rather than having to retrain yourself at a later date.

Let's make a simple dataframe to test.

```
df = pd.DataFrame(data={'col1':np.random.randint(0, 10, 10),
      'col2':np.random.randint(-10, 10, 10)})
>>
   col1  col2
0     0     6
1     6    -1
2     8     4
3     0     5
4     3    -7
5     4    -5
6     3   -10
7     9    -8
8     0     4
9     7    -4
```

Let's test if all the values in col1 are >= 0 by using the built in method assert which comes with the standard library in python. What you're asking python if is True all the items in df['col1'] are greater than zero. If this is True then continue on your way, if not throw an error

```
assert(df['col1'] >= 0 ).all() # Should return nothing
```

Humm looks like we have some options when we're testing our dataframes. Let's test if any of the values are strings.

```
assert(df['col1'] != str).any() # Should return nothing
```

The best practice with asserts is to be used to test conditions within your data that should never happen. This is so when you're running your code, everything stops should one of these assertions fail.

The .all() method will check if all the elements in the objects pass the assert, while .any() will check if any of the elements in the objects pass the assert test.

This can be helpful when you want to:

- Check if any negative values have been introduced into the data;
- Make sure two columns are exactly the same;
- Determine the results of a transformation, or;
- Check if unique id count is accurate.

Pandas Testing Package

Not only do we get an error thrown, but pandas will tell us what is wrong

```
import pandas.util.testing as tm
tm.assert_series_equal(df['col1'], df['col2'])
>>
AssertionError: Series are different. Series values are different
(100.0 %)
[left]:  [0, 6, 8, 0, 3, 4, 3, 9, 0, 7]
[right]: [6, -1, 4, 5, -7, -5, -10, -8, 4, -4]
```

Additionally, if you want to start building yourself a testing suite — and you might want to think about doing this — get familiar with the unittest package built into the Python library

beautifier

Instead of having to write your own regex — which is a pain at the best of times — sometimes it's been done for you. The beautifier package is able to help you clean up some commonly used patterns for emails or URLs. It's nothing fancy but can quickly help with clean up.

Install beautifier first using: pip3 install beautifier

```
from beautifier import Email, Url
email_string = 'foo@bar.com'
email = Email(email_string)
print(email.domain)
print(email.username)
print(email.is_free_email)

Output:
bar.com
foo
False
url_string =
'https://github.com/labtocat/beautifier/blob/master/beautifier/__init__.py
'
url = Url(url_string)
print(url.param)
print(url.username)
print(url.domain)

Output:
None
{'msg': 'feature is currently available only with linkedin urls'}
  github.com
```

Dealing with Unicode

When doing some NLP, dealing with Unicode can be frustrating at the best of times. I'll be running something in spaCy and suddenly everything will break on me because of some unicode character appearing somewhere in the document body.

It really is the worst.

By using using ftfy (fixed that for you) you're able to fix really broken Unicode. Consider when someone has encoded Unicode with one standard and decoded it with a different one. Now you have to deal with this in between string, as nonsense sequences called "mojibake".

```
# Example of mojibake
¯\\_(ã\x83\x84)_/¯
\ufeffParty
\001\033[36;44mI'm
```

Let's see what our strings above can be converted into, so we can read it. The main method is fix_text(), and you'll use that to perform the decoding.

```
import ftfy

foo = '¯\\_(ã\x83\x84)_/¯'
bar = '\ufeffParty'
baz = '\001\033[36;44mI'm'

print(ftfy.fix_text(foo))
print(ftfy.fix_text(bar))
print(ftfy.fix_text(baz))
```