

Cleaning and Prepping Data with Python for Data Science — Best Practices and Helpful Packages



Robert R.F. DeFilippi [Follow](#)

Oct 15, 2018 · 24 min read

Preface

Cleaning data is just something you're going to have to deal with in analytics. It's not great work, but it has to be done so you can *produce* great work.

I've spent so much time writing and rewriting functions to help me clean data, that I wanted to share some of what I've learned along the way. If you have not gone over this post, on how to better organize data science projects check it out as it will help form some of the concepts I'm going over below.

After starting to organize my code better, I've started keeping a custom package where I keep my 'clean up' code. If anything else, it gives me a baseline for writing custom methods on data that does not quite fit my previous clean up scripts. And, I don't need to write that regex email extractor for the 100th time because I've saved it in an accessible location.

Some companies have entire teams devoted to cleaning code, but most don't. So it's best to understand some of the best practices. If anything, you'll get better at understanding the structure of your data, so to better explain why or why not something has occurred.

Also, in preparing for this post I ran across this repo by [kjam](#), which would have been incredibly helpful when I was first learning how to clean data. If you want to go deeper into code cleaning, I suggest you start there.





Your goal is to clean things up ... or at least try to

Check Your Data ... Quickly

The first thing you want to do when you get a new dataset, is to quickly to verify the contents with the `.head()` method.

```
import pandas as pd

df = pd.read_csv('path_to_data')
df.head(10)

>>
... some output here ...
```

Now let's quickly 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 and other oddities. But to check upfront.

```
# 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`. If there is no index, you'll get an `AttributeError: 'function' object has no attribute 'index'` error displayed.

```
# 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)
```

Good. Our data has been quickly checked, we know the data types, if columns are unique, and we know it has an index so we can do joins and merges later on. Let's figure out which columns you want to keep or remove. In this example, we want to get rid of the columns in indexes 1, 3, and 5, so I've just added the string values to a list, which will be used to drop the columns.

```
# 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`. Many of the methods in pandas support `inplace=True`, so try to use it as much as possible to avoid unnecessary reassignment.

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 to 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={'coll':[np.nan, np.nan, 2,3,4, np.nan,
np.nan]}))
```

```

    coll
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)
```

```

    coll
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')
```

```

    coll
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)
```

The parameter `thresh=N` requires that a column has at least `N` non-NaNs to survive. Think of this as the lower limit for missing data you will find acceptable in your columns.

Consider some logging data which might miss some collection of features. You only want the records that have 90% of the available features before you consider them as candidates for your model.

np.where(if_this_is_true, do_this, else_do_that)

I'm guilty in not using this earlier in my analytics career because it is beyond useful. It saves so much time and frustration when munging through a dataframe. If you want to do some basic cleaning or feature engineering quickly, `np.where` here is how you can do it.

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



Credit to <https://www.programiz.com>

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
```

Great seems to have worked. But what if `.all()` was not included in the assert?

```
assert(df['col1'] >= 0 )

>>
ValueError: The truth value of a Series is ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().
```

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
```

What about testing the two columns to see if they are equal?

```
assert(df['col1'] == df['col2']).all()

>>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Ah, our `assert` failed here!

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.

There are more `assert` methods which I won't go over, but get familiar which you can use here. You'll never know when you need to test for a certain condition, and at the same time, you need to start testing for conditions you don't want in your code.

Don't test for everything, but test for things which would break your models.

E.g. Is a feature with should all be 0s and 1s, actually populated with those values.

Additionally, that wonder package `pandas` also includes a testing package.

```
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]
```

Not only did we get an error thrown, but pandas told us what was wrong.

Nice.

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. You can learn more about that [here](#).

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.

```
$ 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)

>>
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)

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

```

I use this package when I have a slew of URLs I need to work through and don't want to write the regex for the 100th time to extract certain parts of the address.

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

&macr;\_\_(&x83&x84)\_/&macr;
\ufe92Party
\001\033[36;44mI&#x92;m

```

Luckily, `ftfy` uses heuristics to detect and undo mojibake, with a very low rate of false positives. 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 = '&macr;\_\_(&x83&x84)\_/&macr;'
bar = '\ufe92Party'

```

```

baz = '\001\033[36;44mI&#x92;m'

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

```

If you want to see how the decoding is done, try `ftfy.explain_unicode()`. I don't think this will be overly helpful, but it's interesting to see the process.

```

ftfy.explain_unicode(foo)

U+0026 &      [Po] AMPERSAND
U+006D m      [Ll] LATIN SMALL LETTER M
U+0061 a      [Ll] LATIN SMALL LETTER A
U+0063 c      [Ll] LATIN SMALL LETTER C
U+0072 r      [Ll] LATIN SMALL LETTER R
U+003B ;      [Po] SEMICOLON
U+005C \      [Po] REVERSE SOLIDUS
U+005F _      [Pc] LOW LINE
U+0028 (      [Ps] LEFT PARENTHESIS
U+00E3 ã      [Ll] LATIN SMALL LETTER A WITH TILDE
U+0083 \x83   [Cc] <unknown>
U+0084 \x84   [Cc] <unknown>
U+0029 )      [Pe] RIGHT PARENTHESIS
U+005F _      [Pc] LOW LINE
U+002F /      [Po] SOLIDUS
U+0026 &      [Po] AMPERSAND
U+006D m      [Ll] LATIN SMALL LETTER M
U+0061 a      [Ll] LATIN SMALL LETTER A
U+0063 c      [Ll] LATIN SMALL LETTER C
U+0072 r      [Ll] LATIN SMALL LETTER R
U+003B ;      [Po] SEMICOLON
None

```

Dedupe

This is a library that uses machine learning to perform de-duplication and entity resolution quickly on structured data. There is a great post here that goes into much more detail than I will and which I've drawn heavily on.

We'll be going through Download Chicago Early Childhood Location data, which can be found here. It has a bunch of missing values and duplicated values from different data sources, so it's good to learn on.

If you've ever gone through duplicated data before, this will look very familiar.

```

# Columns and the number of missing values in each

Id has 0 na values
Source has 0 na values
Site name has 0 na values
Address has 0 na values
Zip has 1333 na values
Phone has 146 na values

```

```

Fax has 3299 na values
Program Name has 2009 na values
Length of Day has 2009 na values
IDHS Provider ID has 3298 na values
Agency has 3325 na values
Neighborhood has 2754 na values
Funded Enrollment has 2424 na values
Program Option has 2800 na values
Number per Site EHS has 3319 na values
Number per Site HS has 3319 na values
Director has 3337 na values
Head Start Fund has 3337 na values
Early Head Start Fund has 2881 na values
CC fund has 2818 na values
Progmod has 2818 na values
Website has 2815 na values
Executive Director has 3114 na values
Center Director has 2874 na values
ECE Available Programs has 2379 na values
NAEYC Valid Until has 2968 na values
NAEYC Program Id has 3337 na values
Email Address has 3203 na values
Ounce of Prevention Description has 3185 na values
Purple binder service type has 3215 na values
Column has 3337 na values
Column2 has 3018 na values

```

The `preProcess` method provided by `dedupe` is necessary to make sure errors don't occur during the sampling and training phases of the model. Trust me, using this will make using `dedupe` much easier. Save this method in your local 'cleaning package' so you can use it in the future when dealing with duplicated data.

```

import pandas as pd
import numpy
import dedupe
import os
import csv
import re
from unicode import unidecode

def preProcess(column):
    '''
    Used to prevent errors during the dedupe process.
    '''
    try :
        column = column.decode('utf8')
    except AttributeError:
        pass
    column = unidecode(column)
    column = re.sub(' +', ' ', column)
    column = re.sub('\n', ' ', column)
    column = column.strip().strip('"').strip("'").lower().strip()

    if not column:
        column = None
    return column

```

Now start to import the the `.csv` column by column, while processing the data.

```

def readData(filename):

    data_d = {}
    with open(filename) as f:
        reader = csv.DictReader(f)
        for row in reader:
            clean_row = [(k, preProcess(v)) for (k, v) in
row.items()]
            row_id = int(row['Id'])
            data_d[row_id] = dict(clean_row)

    return df

name_of_file = 'data.csv'

print('Cleaning and importing data ... ')
df = readData(name_of_file)

```

Now we need to tell `dedupe` what features we should be looking at to determine duplicate values. Below, each `feature` is denoted by `field`, and assigned a data `type` and if it has any missing values. There is a whole list of different variables types you can use here, but to keep it easy we'll stick with strings for now.

I'm also not going to be using every single column to determine the duplication, but you can if you think that will make identifying the values in your dataframe easier.

```

# Set fields
fields = [
    {'field' : 'Source', 'type': 'Set'},
    {'field' : 'Site name', 'type': 'String'},
    {'field' : 'Address', 'type': 'String'},
    {'field' : 'Zip', 'type': 'Exact', 'has missing' : True},
    {'field' : 'Phone', 'type': 'String', 'has missing' : True},
    {'field' : 'Email Address', 'type': 'String', 'has missing' :
True},
]

```

Now let's start feeding `dedupe` some data.

```

# Pass in our model
deduper = dedupe.Dedupe(fields)

# Check if it is working
deduper
>>
<dedupe.api.Dedupe at 0x11535bbe0>

# Feed some sample data in ... 15000 records
deduper.sample(df, 15000)

```

Now we're on to the labelling part. When you run this method below, you'll be prompted by `dedupe` to do some simple labelling.

```
dedupe.consoleLabel(deduper)
```



What you should see; manually training the deduper

The real 'a ha!' moment is when you get this prompt. This is `deduper` asking you to train it, so it know what to look for. You know what a duplicate value should look like, so just pass that knowledge on.

```
Do these records refer to the same thing?  
(y)es / (n)o / (u)nsure / (f)inished
```

Now you no longer have to search through tons and tons of records to see if duplication has occurred. Instead, a neural net is being trained by you to find duplicates in the dataframe.

Once you've provided it with some labeling, finish the training process and save your progress. You can come back to your neural net later if find you have repeated dataframe objects which need deduping.

```
deduper.train()  
  
# Save training  
with open(training_file, 'w') as tf:  
    deduper.writeTraining(tf)  
  
# Save settings  
with open(settings_file, 'wb') as sf:  
    deduper.writeSettings(sf)
```

We're almost done, as next we need to set a threshold for our data. When `recall_weight` is equal to 1 we are telling `deduper` to value recall just as much as precision. However, if `recall_weight=3`, we would value recall three times as much. You can play with these settings to see what works best for you.

```
threshold = deduper.threshold(df, recall_weight=1)
```

Finally, we can now search through our `df` and see where the duplicates are. It's been a long time getting to this position, but this is much *much* better than doing this by hand.

```
# Cluster the duplicates together
clustered_dupes = deduper.match(data_d, threshold)

print('There are {} duplicate sets'.format(len(clustered_dupes)))
```

So let's take a look at our duplicates.

```
clustered_dupes

>>
[(0, 1, 215, 509, 510, 1225, 1226, 1879, 2758, 3255),
 array([0.88552043, 0.88552043, 0.77351897, 0.88552043, 0.88552043,
        0.88552043, 0.88552043, 0.89765924, 0.75684386,
        0.83023088])),
 (2, 3, 216, 511, 512, 1227, 1228, 2687), ...
```

Hum, that's not telling us much. Actually, what is that showing us? What happened to all our values?

If you look closely the values `(0, 1, 215, 509, 510, 1225, 1226, 1879, 2758, 3255)` are all the id locations of duplicates `deduper` thinks are actually the same value. And, we can look at the original data to verify this.

```
{'Id': '215',
 'Source': 'cps_early_childhood_portal_scrape.csv',
 'Site name': 'salvation army temple',
 'Address': '1 n. ogden',
 ...

{'Id': '509',
 'Source': 'cps_early_childhood_portal_scrape.csv',
 'Site name': 'salvation army - temple / salvation army',
 'Address': '1 n ogden ave',
 'Zip': None,
 ..
```

This look like duplicates to me. Nice.

There are many more advanced uses of deduper, such as `matchBlocks` for sequences of clusters, or `Interaction` fields where the interaction between two fields is not just additive but multiplicative. This has already been a lot to go over, so I'll leave that explanation for the article above.

String Matching with fuzzywuzzy

Try this library. It's really interesting because it gives you a *score* for how close strings are when they are compared.

This has been an immensely great tool, as I've done projects in the past where I've needed to rely on Google Sheet's fuzzymatch addon to diagnose data validation issues — think CRM rules not being applied or acted on correctly — and needed to clean records to do any sort of analysis.

But, for large datasets this approach kinda falls flat.

However, with `fuzzywuzzy` you can start to get into string matching in a more scientific matter. Not to get too technical, but it uses something called Levenshtein distance when comparing. This is a string similarity metric for two sequences, such that the distance between is the number of single character edits required to change one word to the other word.

E.g. if you want to change the string `foo` into `bar`, the minimum number of characters to change would be 3, and this is used to determine the 'distance'.

Let's see how this works in practice.

```
$ pip3 install fuzzywuzzy

# test.py
from fuzzywuzzy import fuzz
from fuzzywuzzy import process

foo = 'is this string'
bar = 'like that string?'

fuzz.ratio(foo, bar)
>>
71

fuzz.WRatio(foo, bar) # Weighted ratio
>>
73

fuzz.UQRatio(foo, bar) # Unicode quick ratio
>> 73
```

The `fuzzywuzzy` package has different ways to evaluate strings (`WRatio`, `UQRatio`, etc.) and I'm just going to stick with the standard implementation for this article.

Next, we can look at a tokenized string, which `returns` a measure of the sequences' similarity between 0 and 100 but sorting the token *before* comparing. This is key as you might just want to see the contents of the strings, rather than their positions.

The strings `foo` and `bar` have the same tokens but are structurally different. Do you want to treat them the same? Now you can easily look and account for this type of difference within your data.

```
foo = 'this is a foo'
bar = 'foo a is this'

fuzz.ratio(foo, bar)
>>
31

fuzz.token_sort_ratio('this is a foo', 'foo a is this')
>>
100
```

Or next, you need to find the closest match of a string from a list of values. In this case, we're going to be looking at Harry Potter titles.

What about that Harry Potter book with the ... something title... it has ... I dunno. I just need to guess and see which one of these books scores closest to my guess.

My guess is `'fire'` and let's see how it scores against the possible list of titles.

```
lst_to_eval = ['Harry Potter and the Philosopher\'s Stone',
               'Harry Potter and the Chamber of Secrets',
               'Harry Potter and the Prisoner of Azkaban',
               'Harry Potter and the Goblet of Fire',
               'Harry Potter and the Order of the Phoenix',
               'Harry Potter and the Half-Blood Prince',
               'Harry Potter and the Deathly Hallows']

# Top two responses based on my guess
process.extract("fire", lst_to_eval, limit=2)
>>
[('Harry Potter and the Goblet of Fire', 60), ('Harry Potter and the Sorcerer's Stone', 30)]

results = process.extract("fire", lst_to_eval, limit=2)
for result in results:
    print('{}: has a score of {}'.format(result[0], result[1]))

>>
Harry Potter and the Goblet of Fire: has a score of 60
Harry Potter and the Sorcerer's Stone: has a score of 30
```


Or if you just want to return one, you can.

```
>>> process.extractOne("stone", lst_to_eval)
("Harry Potter and the Sorcerer's Stone", 90)
```

I know we talked about dedupe'ing earlier, but here is another application of the same process with `fuzzywuzzy`. We can take a list of strings containing duplicates and uses fuzzy matching to identify and remove duplicates.

Not as fancy as a neural net, but it will do the job for small operations.

We'll continue with the Harry Potter theme, and look for duplicate characters from the books within a list.

You'll need to set a threshold between 0 and 100. As the threshold decreases the number of duplications found will increase, so the returned list will be shorted. The default is

70.

```
# List of duplicate character names
contains_dupes = [
    'Harry Potter',
    'H. Potter',
    'Harry James Potter',
    'James Potter',
    'Ronald Bilius \'Ron\' Weasley',
    'Ron Weasley',
    'Ronald Weasley']

# Print the duplicate values
process.dedupe(contains_dupes)
>>
dict_keys(['Harry James Potter', 'Ronald Bilius \'Ron\' Weasley'])

# Print the duplicate values with a higher threshold
process.dedupe(contains_dupes, threshold=90)
>>
dict_keys(['Harry James Potter', 'H. Potter', 'Ronald Bilius \'Ron\'
Weasley'])
```

And, as a quick bonus you can also do some fuzzy matching with the `datetime` package to extract dates from a string of text. This is great when you don't want to (again) write a regex expression.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
>>
2047-01-01 08:21:00
```

```
dt = parse("May 18, 2049 something something", fuzzy=True)
print(dt)
>>
2049-05-18 00:00:00
```

Try some sklearn

Along with cleaning the data, you'll also need to prepare the data so it is in a form you can feed into your model. Most of the examples here are pulled directly from the documentation, which should be checked out as it really does a good job of explaining more of the newness of each package.

We'll be importing the `preprocessing` package first, then getting additional methods from there as we go along. Also, I'm using `sklearn` version `0.20.0`, so if you're having issues with importing some of the packages check your version.

We'll be working with two different types of data, `str` and `int` just to highlight how the different preprocessing techniques work.

```
# At start of project
from sklearn import preprocessing

# And let's create a random array of ints to process
ary_int = np.random.randint(-100, 100, 10)

ary_int
>> [ 5, -41, -67, 23, -53, -57, -36, -25, 10, 17]

# And some str to work with
ary_str = ['foo', 'bar', 'baz', 'x', 'y', 'z']
```

Let's try some quick labelling with `LabelEncoder` on our `ary_str`. This is important because you can't just feed raw strings — well you can but that is beyond the scope of this article — in your models. So, we'll encode labels to each of the strings, with value between 0 and n. In our `ary_str`, we have 6 unique values so our range would be 0 - 5.

```
from sklearn.preprocessing import LabelEncoder

l_encoder = preprocessing.LabelEncoder()
l_encoder.fit(ary_str)
>> LabelEncoder()

# What are our values?
l_encoder.transform(['foo'])
>> array([2])
l_encoder.transform(['baz'])
>> array([1])
l_encoder.transform(['bar'])
>> array([0])
```

You'll notice these are *not* ordered, as even though `foo` came before `bar` in the array, it was encoded with `2` while `bar` was encoded with `1`. We'll use a different encoding method when we need to make sure our values are encoded in the correct order.

If you have a lot of categories to keep track of you might forget which `str` maps to which `int`. For that, we can create a dict.

```
# Check mappings
list(l_encoder.classes_)
>> ['bar', 'baz', 'foo', 'x', 'y', 'z']

# Create dictionary of mappings
dict(zip(l_encoder.classes_,
        l_encoder.transform(l_encoder.classes_)))

>> {'bar': 0, 'baz': 1, 'foo': 2, 'x': 3, 'y': 4, 'z': 5}
```

The process is a little different if you have a dataframe, but actually a little easier. You just need to `.apply()` the `LabelEncoder` object to the `DataFrame`. For each column, you'll get a unique label for the values within that column. Notice how `foo` is encoded to `1`, but so is `y`.

```
# Try LabelEncoder on a dataframe
import pandas as pd
l_encoder = preprocessing.LabelEncoder() # New object

df = pd.DataFrame(data = {'col1': ['foo', 'bar', 'foo', 'bar'],
                          'col2': ['x', 'y', 'x', 'z'],
                          'col3': [1, 2, 3, 4]})

# Now for the easy part
df.apply(l_encoder.fit_transform)

>>

   col1  col2  col3
0     1     0     0
1     0     1     1
2     1     0     2
3     0     2     3
```

Now, we're moving on to ordinal encoding where features are still expressed as integer values, but they have a sense of place and structure. Such that `x` comes before `y`, and `y` comes before `z`.

However, we're going to throw a wrench in here. Not only are the values ordered, but they are going to be paired with each other.

We're going to take an two array of values ['foo', 'bar', 'baz'] and ['x', 'y', 'z']. Next we'll encode 0, 1, and 2 to each set of values in each array, and create an encoded pair for each of the values.

E.g. ['foo', 'z'] would be mapped to [0, 2], and ['baz', 'x'] would be mapped to [2, 0].


This is a good approach to take when you need to take a bunch of categories and make them available for a regression, and especially good when you have interleaving sets of strings — separate categories which still overlap with one another — and need representation in the dataframe.

```
from sklearn.preprocessing import OrdinalEncoder
o_encoder = OrdinalEncoder()
ary_2d = [['foo', 'bar', 'baz'], ['x', 'y', 'z']]

o_encoder.fit(ary_2d) # Fit the values
o_encoder.transform(['foo', 'y'])

>> array([[0., 1.]])
```

The classic one hot or 'dummy' encoding, where single features of categories are then expressed as additional columns of 0s or 1s, depending on if the value appears or not. This process creates a binary column for each category and returns a sparse matrix or dense array.

ID	Gender		ID	Male	Female	Not Specified
1	Male		1	1	0	0
2	Female		2	0	1	0
3	Not Specified		3	0	0	1
4	Not Specified		4	0	0	1
5	Female		5	0	1	0

Credit to <https://blog.myyellowroad.com/>

Why even use this? Because this type of encoding is needed for feeding categorical data to many `scikit` models such as linear regression models and SVMs. So get comfortable with this.

```
from sklearn.preprocessing import OneHotEncoder
hot_encoder = OneHotEncoder(handle_unknown='ignore')

hot_encoder.fit(ary_2d)
hot_encoder.categories_

>>
[array(['foo', 'x'], dtype=object), array(['bar', 'y'],
dtype=object), array(['baz', 'z'], dtype=object)]
```

```

hot_encoder.transform([[ 'foo', 'foo', 'baz'], [ 'y', 'y',
'x']]).toarray()

>>
array([[1., 0., 0., 0., 1., 0.],
       [0., 0., 0., 1., 0., 0.]])

```

What about if we had a dataframe to work with?

Could we still use one hot encoding? It's actually much easier than you think as you just need to use the `.get_dummies()` included in pandas.

```

pd.get_dummies(df)

```

	col3	col1_bar	col1_foo	col2_x	col2_y	col2_z
0	1	0	1	1	0	0
1	2	1	0	0	1	0
2	3	0	1	1	0	0
3	4	1	0	0	0	1

Two of the three columns in `df` have been split up and binary encoded to a dataframe.

E.g. the column `col1_bar` is `col1` from `df`, but has `1` as the record value when `bar` was the value in the original dataframe.

What about when our features need to be transformed within a certain range. By using `MinMaxScaler`, each feature can be individually scaled such that it is in the given range. By default the values are between `0` and `1`, but you're able to change the range.

```

from sklearn.preprocessing import MinMaxScaler
mm_scaler = MinMaxScaler(feature_range=(0, 1)) # Between 0 and 1

mm_scaler.fit([ary_int])
>> MinMaxScaler(copy=True, feature_range=(0, 1))

print(scaler.data_max_)
>> [ 5. -41. -67. 23. -53. -57. -36. -25. 10. 17.]

print(mm_scaler.fit_transform([ary_int]))
>> [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] # Humm something is wrong

```

If you notice the able the output is all zeros ... which is not what we wanted. There is a good explanation [here](#) and [here](#), on why that would have happened, but the short story is, the array is formatted incorrectly.

It is a `(1, n)` matrix and needs to be converted to an `(n, 1)` matrix. The easiest way to do this is make sure your array is a `numpy` array, so you're able to manipulate the shape.

```

# Create numpy array
ary_int = np.array([ 5, -41, -67, 23, -53, -57, -36, -25, 10,
17])

# Transform
mm_scaler.fit_transform(ary_int[:, np.newaxis])
>>
array([[0.8
        ],
       [0.28888889],
       [0.
        ],
       [1.
        ],
       [0.15555556],
       [0.11111111],
       [0.34444444],
       [0.46666667],
       [0.85555556],
       [0.93333333]])

# You can also use
mm_scaler.fit_transform(ary_int.reshape(-1, 1))

# Also try a different scale
mm_scaler = MinMaxScaler(feature_range=(0, 10))
mm_scaler.fit_transform(ary_int.reshape(-1, 1))
>>
array([[ 8.
        ],
       [ 2.88888889],
       [ 0.
        ],
       [10.
        ],
       [ 1.55555556],
       [ 1.11111111],
       [ 3.44444444],
       [ 4.66666667],
       [ 8.55555556],
       [ 9.33333333]])

```

Now that we can quickly scale our data, what about implementing some sort of shape to our transformed data? We're looking at standardizing the data, which is going to give you values that create a gaussian with a mean of 0 and an sd of 1. You might consider this approach when implementing gradient descent, or if you need weighted inputs like regression and neural networks. Also, if you're going to implement a KNN, scale your data first. Note this approach is *different* from normalization, so don't get confused.

Simply use the `scale` from `preprocessing`.

```

preprocessing.scale(foo)
>> array([ 0.86325871, -0.58600774, -1.40515833,  1.43036297,
        -0.96407724, -1.09010041, -0.42847877, -0.08191506,  1.02078767,
        1.24132821])

preprocessing.scale(foo).mean()
>> -4.4408920985006264e-17 # Essentially zero

preprocessing.scale(foo).std()
>> 1.0 # Exactly what we wanted

```

The last `sklearn` package to look at is `Binarizer`, you're still getting 0s and 1s through this but now they are defined on your own terms. This is the process of 'thresholding' numerical features to get boolean values. The values threshold greater than the threshold will map to `1`, while those \leq to will map to `0`. As well, this is a common process when text preprocessing to get the term frequencies within a document or corpus.

Keep in mind, both `fit()` and `transform()` require a 2d array, which is why I've nested `ary_int` in another array. For this example I've put the `threshold` as `-25`, so any numbers strictly above that will be assigned a `1`.

```
from sklearn.preprocessing import Binarizer

# Set -25 as our threshold
tz = Binarizer(threshold=-25.0).fit([ary_int])

tz.transform([ary_int])
>>array([[1, 0, 0, 1, 0, 0, 0, 0, 1, 1]])
```

Now that we have these few different techniques, which one is the best for your algorithm? It's probably best to save a few different intermediate dataframes with scaled data, binned data, etc. so you're able to see the effect on the output of your model(s).

Final Thoughts

Cleaning and prepping data is inevitable and generally a thankless task when it comes to data science. If you're lucky enough to have a data engineering team with you who can help set up ETL pipelines to make your job easier, then you might be in the minority of data scientists.

Life is not just a bunch of Kaggle datasets, where in reality you'll have to make decisions on how to access and clean the data you need everyday. Sometimes you'll have a lot of time to make sure everything is in the right place, but most of the time you'll be pressed for answers. If you have the right tools in place and understanding of what is possible, you'll be able to get to those answers easily.

As always, I hope you've learned something new.

Cheers,

Additional Reading

Pythonic Data Cleaning With NumPy and Pandas - Real Python

Data scientists spend a large amount of their time cleaning datasets and getting them down to a form with which they...

realpython.com

--	--

Handy Python Libraries for Formatting and Cleaning Data

The real world is messy, and so too is its data. So messy, that a recent survey reported data scientists spend 60% of...

blog.modeanalytics.com

Cleaning Data in Python DataCamp Data scientists spend 80% of their time cleaning and manipulating data. This course will equip you with all the skills... www.datacamp.com	
--	--

Seven Clean Steps To Reshape Your Data With Pandas Or How I Use Python Where Excel Fails Concepts: multi-level indexing, pivoting, stacking, apply, lambda, and list-comprehension towardsdatascience.com	
--	--

[Data Science](#)
[Pandas](#)
[Python](#)
[Data Preprocess](#)
[Sklearn](#)

[About](#)
[Help](#)
[Legal](#)

Get the Medium app

