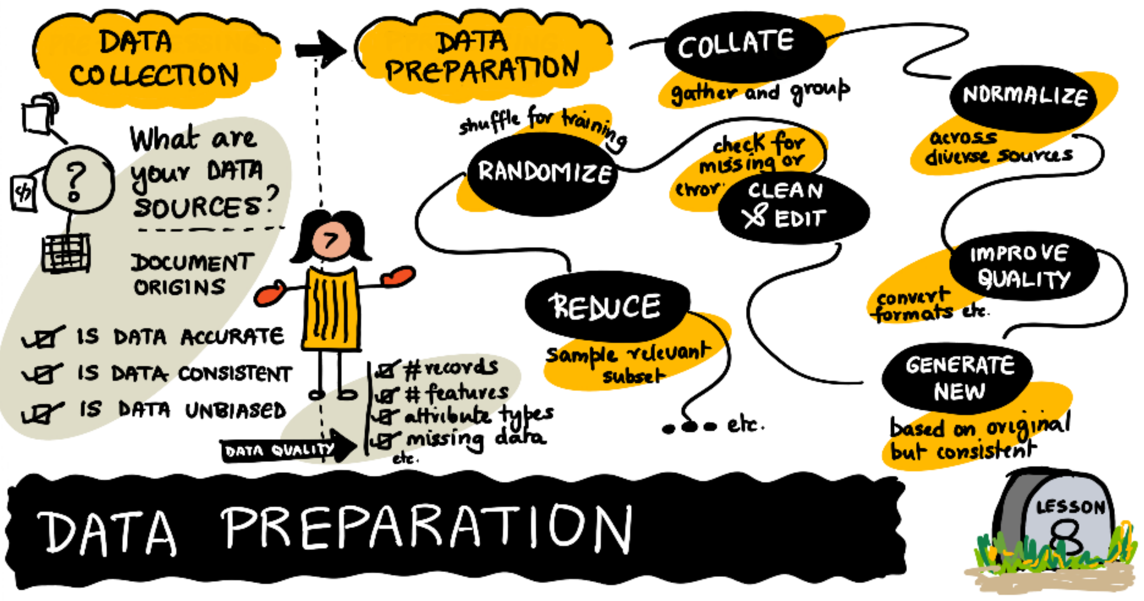
Working with Data: Data Preparation



Data Preparation - *Sketchnote by @nitya*

Pre-Lecture Quiz

Depending on its source, raw data may contain some inconsistencies that will cause challenges in analysis and modeling. In other words, this data can be categorized as "dirty" and will need to be cleaned up. This lesson focuses on techniques for cleaning and transforming the data to handle challenges of missing, inaccurate, or incomplete data. Topics covered in this lesson will utilize Python and the Pandas library and will be demonstrated in the notebook within this directory.

The importance of cleaning data

- **Ease of use and reuse**: When data is properly organized and normalized it's easier to search, use, and share with others.
- **Consistency**: Data science often requires working with more than one dataset, where datasets from different sources need to be joined together. Making sure that each individual data set has common standardization will ensure that the data is still useful when they are all merged into one dataset.
- **Model accuracy**: Data that has been cleaned improves the accuracy of models that rely on it.

Common cleaning goals and strategies

- **Exploring a dataset**: Data exploration, which is covered in a later lesson can help you discover data that needs to be cleaned up. Visually observing values within a dataset can set expectations of what that rest of it will look like, or provide an idea of the problems that can be resolved. Exploration can involve basic querying, visualizations, and sampling.

- **Formatting**: Depending on the source, data can have inconsistencies in how it's presented. This can cause problems in searching for and representing the value, where it's seen within the dataset but is not properly represented in visualizations or query results. Common formatting problems involve resolving whitespace, dates, and data types. Resolving formatting issues is typically up to the people who are using the data. For example, standards on how dates and numbers are presented can differ by country.
- **Duplications**: Data that has more than one occurrence can produce inaccurate results and usually should be removed. This can be a common occurrence when joining two or more datasets together. However, there are instances where duplication in joined datasets contain pieces that can provide additional information and may need to be preserved.
- **Missing Data**: Missing data can cause inaccuracies as well as weak or biased results. Sometimes these can be resolved by a "reload" of the data, filling in the missing values with computation and code like Python, or simply just removing the value and corresponding data. There are numerous reasons for why data may be missing and the actions that are taken to resolve these missing values can be dependent on how and why they went missing in the first place.

Exploring DataFrame information

> **Learning goal:** By the end of this subsection, you should be comfortable finding general information about the data stored in pandas DataFrames.

Once you have loaded your data into pandas, it will more likely than not be in a DataFrame(refer to the previous lesson for detailed overview). However, if the data set in your DataFrame has 60,000 rows and 400 columns, how do you even begin to get a sense of what you're working with? Fortunately, pandas provides some convenient tools to quickly look at overall information about a DataFrame in addition to the first few and last few rows.

In order to explore this functionality, we will import the Python scikit-learn library and use an iconic dataset: the **Iris data set**.

```python
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()
iris_df = pd.DataFrame(data=iris['data'], columns=iris['feature_names'])
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

- **DataFrame.info**: To start off, the `info()` method is used to print a summary of the content present in a `DataFrame`. Let's take a look at this dataset to see what we have:

```
iris_df.info()
```

```
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
dtypes: float64(4)
memory usage: 4.8 KB
```

From this, we know that the *Iris* dataset has 150 entries in four columns with no null entries. All of the data is stored as 64-bit floating-point numbers.

- **DataFrame.head()**: Next, to check the actual content of the `DataFrame`, we use the `head()` method. Let's see what the first few rows of our `iris_df` look like:

```
iris_df.head()
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
```

- **DataFrame.tail()**: Conversely, to check the last few rows of the `DataFrame`, we use the `tail()` method:

```
iris_df.tail()
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8
```

> **Takeaway:** Even just by looking at the metadata about the information in a DataFrame or the first and last few values in one, you can get an immediate idea about the size, shape, and content of the data you are dealing with.

Dealing with Missing Data

> **Learning goal:** By the end of this subsection, you should know how to replace or remove null values from DataFrames.

Most of the time the datasets you want to use (of have to use) have missing values in them. How missing data is handled carries with it subtle tradeoffs that can affect your final analysis and real-world outcomes.

Pandas handles missing values in two ways. The first you've seen before in previous sections: `NaN`, or Not a Number. This is a actually a special value that is part of the IEEE floating-point specification and it is only used to indicate missing floating-point values.

For missing values apart from floats, pandas uses the Python `None` object. While it might seem confusing that you will encounter two different kinds of values that say essentially the same thing, there are sound programmatic reasons for this design choice and, in practice, going this route enables pandas to deliver a good compromise for the vast majority of cases. Notwithstanding this, both `None` and `NaN` carry restrictions that you need to be mindful of with regards to how they can be used.

Check out more about `NaN` and `None` from the notebook!

- **Detecting null values**: In `pandas`, the `isnull()` and `notnull()` methods are your primary methods for detecting null data. Both return Boolean masks over your data. We will be using `numpy` for `NaN` values:

```
import numpy as np

example1 = pd.Series([0, np.nan, '', None])
example1.isnull()
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

Look closely at the output. Does any of it surprise you? While `0` is an arithmetic null, it's nevertheless a perfectly good integer and pandas treats it as such. `''` is a little more subtle. While we used it in Section 1 to represent an empty string value, it is nevertheless a string object and not a representation of null as far as pandas is concerned.

Now, let's turn this around and use these methods in a manner more like you will use them in practice. You can use Boolean masks directly as a `Series` or `DataFrame` index, which can be useful when trying to work with isolated missing (or present) values.

> **Takeaway**: Both the `isnull()` and `notnull()` methods produce similar results when you use them in `DataFrame`s: they show the results and the index of those results, which will help you enormously as you wrestle with your data.

- **Dropping null values**: Beyond identifying missing values, pandas provides a convenient means to remove null values from `Series` and `DataFrame`s. (Particularly on large data sets, it is often more advisable to simply remove missing [NA] values from your analysis than deal with them in other ways.) To see this in action, let's return to `example1`:

```
example1 = example1.dropna()
example1
```

```
0    0
2
dtype: object
```

Note that this should look like your output from `example3[example3.notnull()]`. The difference here is that, rather than just indexing on the masked values, `dropna` has removed those missing values from the `Series` `example1`.

Because `DataFrame`s have two dimensions, they afford more options for dropping data.

```
example2 = pd.DataFrame([[1,      np.nan, 7],
                         [2,      5,      8],
                         [np.nan, 6,      9]])
example2
```

	0	1	2
0	1.0	NaN	7
1	2.0	5.0	8
2	NaN	6.0	9

(Did you notice that pandas upcast two of the columns to floats to accommodate the `NaN`s?)

You cannot drop a single value from a `DataFrame`, so you have to drop full rows or columns. Depending on what you are doing, you might want to do one or the other, and so pandas gives you options for both. Because in data science, columns generally represent variables and rows represent observations, you are more likely to drop rows of data; the default setting for `dropna()` is to drop all rows that contain any null values:

```
example2.dropna()
```

```
    0   1   2
1   2.0 5.0 8
```

If necessary, you can drop NA values from columns. Use `axis=1` to do so:

```
example2.dropna(axis='columns')
```

```
    2
0   7
1   8
2   9
```

Notice that this can drop a lot of data that you might want to keep, particularly in smaller datasets. What if you just want to drop rows or columns that contain several or even just all null values? You specify those setting in `dropna` with the `how` and `thresh` parameters.

By default, `how='any'` (if you would like to check for yourself or see what other parameters the method has, run `example4.dropna?` in a code cell). You could alternatively specify `how='all'` so as to drop only rows or columns that contain all null values. Let's expand our example `DataFrame` to see this in action.

```
example2[3] = np.nan
example2
```

	0	1	2	3
0	1.0	NaN	7	NaN
1	2.0	5.0	8	NaN
2	NaN	6.0	9	NaN

The `thresh` parameter gives you finer-grained control: you set the number of *non-null* values that a row or column needs to have in order to be kept:

```
example2.dropna(axis='rows', thresh=3)
```

```
    0   1   2   3
1   2.0 5.0 8   NaN
```

Here, the first and last row have been dropped, because they contain only two non-null values.

- **Filling null values**: Depending on your dataset, it can sometimes make more sense to fill null values with valid ones rather than drop them. You could use `isnull` to do this in place, but that can be laborious, particularly if you have a lot of values to fill. Because this is such a common task in data science, pandas provides `fillna`, which returns a copy of the `Series` or `DataFrame` with the missing values replaced with one of your choosing. Let's create another example `Series` to see how this works in practice.

```
example3 = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
example3
```

```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

You can fill all of the null entries with a single value, such as `0`:

```
example3.fillna(0)
```

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

You can **forward-fill** null values, which is to use the last valid value to fill a null:

```
example3.fillna(method='ffill')
```

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

You can also **back-fill** to propagate the next valid value backward to fill a null:

```
example3.fillna(method='bfill')
```

```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

As you might guess, this works the same with `DataFrame`s, but you can also specify an `axis` along which to fill null values. taking the previously used `example2` again:

```
example2.fillna(method='ffill', axis=1)
```

```
    0   1   2   3
0   1.0 1.0 7.0 7.0
1   2.0 5.0 8.0 8.0
2   NaN 6.0 9.0 9.0
```

Notice that when a previous value is not available for forward-filling, the null value remains.

> **Takeaway:** There are multiple ways to deal with missing values in your datasets. The specific strategy you use (removing them, replacing them, or even how you replace them) should be dictated by the particulars of that data. You will develop a better sense of how to deal with missing values the more you handle and interact with datasets.

Removing duplicate data

> **Learning goal:** By the end of this subsection, you should be comfortable identifying and removing duplicate values from DataFrames.

In addition to missing data, you will often encounter duplicated data in real-world datasets. Fortunately, `pandas` provides an easy means of detecting and removing duplicate entries.

- **Identifying duplicates: `duplicated`**: You can easily spot duplicate values using the `duplicated` method in pandas, which returns a Boolean mask indicating whether an entry in a `DataFrame` is a duplicate of an earlier one. Let's create another example `DataFrame` to see this in action.

```
example4 = pd.DataFrame({'letters': ['A','B'] * 2 + ['B'],
                         'numbers': [1, 2, 1, 3, 3]})
```

```
example4
```

	letters	numbers
0	A	1
1	B	2
2	A	1
3	B	3
4	B	3

```
example4.duplicated()
```

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

- **Dropping duplicates: `drop_duplicates`:** simply returns a copy of the data for which all of the `duplicated` values are `False`:

```
example4.drop_duplicates()
```

```
    letters numbers
0    A    1
1    B    2
3    B    3
```

Both `duplicated` and `drop_duplicates` default to consider all columns but you can specify that they examine only a subset of columns in your `DataFrame`:

```
example4.drop_duplicates(['letters'])
```

```
letters numbers
0    A    1
1    B    2
```

> **Takeaway:** Removing duplicate data is an essential part of almost every data-science project. Duplicate data can change the results of your analyses and give you inaccurate results!

🚀 Challenge

All of the discussed materials are provided as a Jupyter Notebook. Additionally, there are exercises present after each section, give them a try!

Post-Lecture Quiz

Review & Self Study

There are many ways to discover and approach preparing your data for analysis and modeling and cleaning the data is an important step that is a "hands on" experience. Try these challenges from Kaggle to explore techniques that this lesson didn't cover.

- Data Cleaning Challenge: Parsing Dates
- Data Cleaning Challenge: Scale and Normalize Data

Assignment

Evaluating Data from a Form