

13 Data Cleaning

Lab Objective: *The quality of a data analysis or model is limited by the quality of the data used. In this lab we learn techniques for cleaning data, creating features, and determining feature importance.*

Almost every dataset has problems that make it unsuitable for regression or other modeling. Some problems will throw errors in your code and so will be easily observed. Other data problems are less noticeable. If code executes on poorly maintained data, the results could vary significantly from the true results which might have been obtained if the underlying dataset were better prepared.

Data cleaning is the process of identifying and correcting bad data. This could be data that is missing, duplicated, irrelevant, inconsistent, incorrect, in the wrong format, or otherwise does not make sense. Though it can be tedious, data cleaning is the most important step of data analysis. Without accurate and legitimate data, any results or conclusions are suspect and may be incorrect. We will demonstrate common issues with data and how to correct them using the following dataset. It consists of family members and some basic details.

```
# Example dataset
>>> df = pd.read_csv('toy_dataset.csv')

>>>
      Name  Age      name      DOB  Marital_Status
0   John Doe  30     john  01/01/2010      Divorcee
1   Jane Doe  29     jane  12/02/1990      Divorced
2   Jill smith  40      NaN  03/04/1980      married
3   Jill smith  40     jill  03/04/1980      married
4   jack smith 100     jack   4/4/1980     marrieed
5   Jenny Smith  5      NaN  05/05/2015         NaN
6   JAMES Smith  2      NaN  20/06/2018      single
7      Rover  2      NaN  05/05/2018         NaN

      Height  Weight  Marriage_Len      Spouse
0      72.0    175           5         NaN
1       5.5    125           5     John Doe
```

2	64.0	120	10	Jack Smith
3	64.0	120	NaN	jack smith
4	1.8	220	10	jill smith
5	105.0	40	NaN	NaN
6	27.0	25	Not Applicable	NaN
7	36.0	50	NaN	NaN

Inspection

The first step of data cleaning is to analyze the quality of the data. If the quality is poor, the data might not be worth using. Knowing the quality of the data will also give you an idea of how long it will take to clean it. A quality dataset is one in which the data is valid, accurate, complete, consistent, and uniform. Some of these issues, like uniformity, are fairly easy to fix during cleaning, while other aspects like accuracy are more difficult, if not impossible, to address.

Validity is the degree that the data conforms to given rules. If a column corresponds to the temperature in Salt Lake City, measured in degrees Fahrenheit, then a value over 110 or below 0 should make you suspicious, since those would be extreme values for Salt Lake City. In fact, checking the all-time temperature records for Salt Lake shows that the values in this column should never be more than 107 and never less than -30 . Any values outside that range are almost certainly errors and should probably be reset to *NaN*, unless you have special information that allows you to impute more accurate values.

Some standard considerations when determining the validity of a dataset are:

- **data type:** The data types of each column should all be the same.
- **data range:** The data of a column, typically numbers or dates, should all be within some valid range.
- **mandatory constraints:** Certain columns cannot have missing entries.
- **unique constraint:** Entries in certain columns must be unique.
- **regular expression patterns:** A text column must be in the same format (for example, forcing phone numbers to be formatted as 999-999-9999).
- **cross-field validation:** Conditions must hold across multiple columns (for example, a hospital discharge date can't be earlier than the admittance date).
- **duplicated data:** Rows or columns that are repeated. In some cases, they may not be exact.

We can check the data type in Pandas using `dtype`. A `dtype` of `object` means that the data in that column contains either strings or mixed `dtypes`. These fields should be investigated to determine if they contain mixed datatypes. In our toy example, we would expect that `Marriage_Len` is numerical, so an `object dtype` is suspicious. Looking at the data, we see that James has `Not Applicable`, which is a string.

```
# Check validity of data
# Check Data Types
>>> df.dtypes
Name                object
```

```

Age          int64
name         object
DOB          object
Marital_Status  object
Height       float64
Weight       int64
Marriage_Len  object
Spouse       object
dtype: object

```

Duplicates

Duplicates can be easily identified in Pandas using the `df.duplicated()` function. When no parameters are passed, it returns a DataFrame of the first duplicates. We can identify rows that are duplicated in only some columns by passing in the column names. The `keep` parameter has three possible values, `first`, `last`, and `False`. `False` keeps all duplicated values, while `first` and `last` keep only the first and last instances, respectively.

```

# Display duplicated rows
>>> df[df.duplicated()]
Empty DataFrame
Columns: [Name, Age, name, DOB, Marital_Status, Height, Weight, Marriage_Len, Spouse]
Index: []

# Display rows that have duplicates in some columns
>>> df[df.duplicated(['Name', 'DOB', 'Marital_Status'], keep=False)]
   Name  Age  name      DOB  Marital_Status  Height  Weight  Marriage_Len  Spouse
2  Jill smith  40  NaN  03/04/1980      married    64.0    120      NaN      NaN
10  Jack Smith
3  Jill smith  40  jill  03/04/1980      married    64.0    120      NaN  jack smith

```

Range

We can check the range of values in a numeric column using the `min` and `max` attributes. Other options for looking at the values include line plots, histograms, and boxplots. Some other useful Pandas commands for evaluating the breadth of a dataset include `df.nunique()` (which returns a series giving the name of each column and the number of unique values in each column), `pd.unique()` (which returns an array of the unique values in a series), and `value_counts()` (which counts the number of instances of each unique value in a column, like a histogram).

```

# Count the number of unique values in each row
>>> df.nunique()
Name          7
Age           6

```

```

name          2
DOB           7
Marital_Status 5
Height        7
Weight        7
Marriage_Len  4
Spouse        4
dtype: int64

# Print the unique Marital_Status values
>>> pd.unique(df['Marital_Status'])
array(['Divorcee', 'Divorced', 'married', 'marrieed', nan, 'single'],
      dtype=object)

# Count the number of each Marital_Status values
>>> df['Marital_Status'].value_counts()
married      2
single       1
marrieed     1
Divorcee     1
Divorced     1
Name: Marital_Status, dtype: int64

```

Accuracy

The accuracy of the data, how close the data is to reality, is harder to confirm. Just because a data point is valid, doesn't mean that it is true. For example, a valid street address doesn't have to exist, or a person might lie about their weight. The first case could be checked using mapping software, but the second could be unverifiable.

Missing Data

The percentage of missing data is the completeness of the data. All uncleaned data will have missing values, but datasets with large amounts of missing data, or lots of missing data in key columns, are not going to be as useful. Pandas has several functions to help identify and count missing values. In Pandas, all missing data is considered a `NaN` and does not affect the dtype of a column. `df.isna()` returns a boolean DataFrame indicating whether each value is missing. `df.notnull()` returns a boolean DataFrame with `True` where a value is not missing.

```

# Count number of missing data points in each column
>>> df.isna().sum()
Name          0
Age           0
name          6
DOB           0
Marital_Status 2

```

```

Height      0
Weight      0
Marriage_Len  2
Spouse      4
dtype: int64

```

Consistency

Consistency measures how cohesive the data is, both within the dataset and across multiple datasets. For example, in our toy dataset Jack Smith is 100 years old, but his birth year is 1980. Data is inconsistent across datasets when the data points should be the same and are different. This could be due to incorrect entries or syntax errors.

Uniformity

Lastly, uniformity is the measure of how similarly the data is formatted. Data that has the same units of measure and syntax are considered uniform. Looking at the `Height` column in our dataset, we see values ranging from 1.8 to 105. This is likely the result of different units of measure.

Uniformity also matters across multiple datasets. For example, if you use multiple finance datasets to build a predictive model then the dates in each dataset should have the same format so that they can all be used equally in the model.

No Set Rules

When looking at the quality of the data, there are no set rules on how to measure these concepts or at what point the data is considered bad data. Sometimes the only data available is of poor quality and so must still be used. Other times, higher quality data may be available elsewhere. An idea of the quality of the data will inform you of which cleaning steps are needed and will influence the strength of your final analysis.

You should always investigate the quality of the dataset they wish to use because a model is only as good as the data it relies on. Such an investigation should include statistics summarizing the principles discussed in this section, visualizations to identify outliers in the data, and written descriptions of the mitigating steps taken to improve the data set. Using various data visualizations can also give one a general sense of the quality of their data. Using histograms, box plots, and hexbins can identify outliers in the data. Outliers should be investigated to determine if they are accurate. Removing outliers will improve your model, but you should only remove an outlier if you have a legitimate reason. Columns that have a small distribution or variance, or consist of one value, could be worth removing since they might contribute little to the model.

Problem 1. The `g_t_results.csv` file is a set of parent-reported scores on their child's Gifted and Talented tests. The two tests, OLSAT and NNAT, are used by NYC to determine if children are qualified for gifted programs. The OLSAT Verbal has 16 questions for Kindergartners and 30 questions for first, second, and third graders. The NNAT has 48 questions. Using this dataset, answer the following questions.

1. What column has the highest number of null values and what percent of its values are null? Print the answer as a tuple with (column name, percentage). Make sure the second value is a percent.
 2. List the columns that should be numeric that aren't. Print the answer as a tuple.
 3. How many third graders have scores outside the valid range for the OLSAT Verbal Score? Print the answer
 4. How many data values are missing (NaN)? Print the number.
- Each part is one point.

Cleaning

After the data has been inspected, it's time to start cleaning. There are many aspects and methods of cleaning; not all of them will be used in every dataset. Which ones you choose should be based on your dataset and the goal of the project.

Unwanted Data

Removing unwanted data typically falls into two categories, duplicated data and irrelevant data. Duplicated observations usually occur when data is scraped, combined from multiple datasets, or submitted twice by a user. Irrelevant data consists of observations that don't fit the specific problem you are trying to solve or don't have enough variation to affect the model. We can drop duplicated data using the `duplicated()` function described above with `drop()` or using `drop_duplicates`, which has the same parameters as `duplicated`.

Validity Errors

After removing unwanted data, we correct any validity errors found during inspection. All features should have a consistent type, standard formatting (like capitalization), and the same units. Syntax errors should be fixed, and white space at the beginning and ends of strings should be removed. Some data might need to be padded so that it's all the same length.

Method	Description
<code>series.str.lower()</code>	Convert to all lower case
<code>series.str.upper()</code>	Convert to all upper case
<code>series.str.strip()</code>	Remove all leading and trailing white space
<code>series.str.lstrip()</code>	Remove leading white space
<code>series.str.replace(" ", "")</code>	Remove all spaces
<code>series.str.pad()</code>	Pad strings

Table 13.1: Pandas String Formatting Methods

Validity also includes correcting or removing contradicting values. This might be two values in a row or values across datasets. For example, a child shouldn't have a marital status of married. Or, if two columns should sum to a third but don't, then your data has invalid values which may need to be removed.

Missing Data

There will always be missing data in any uncleaned dataset. Some commonly suggested methods for handling data are removing the missing data and setting the missing values to some value based on other observations. However, missing data can be informative and removing or replacing missing data erases that information. Removing missing values from a dataset might result in losing significant amounts of data or even in a less accurate model. Retaining the missing values can help increase accuracy.

We have several options to deal with missing data:

- Dropping missing data is the easiest method. Dropping rows should only be done if there are a small number of missing data points in a column or if the row is missing a significant amount of data. If a column is very sparse, consider dropping the entire column. If dropping missing data is inappropriate, you may instead choose to estimate the missing values. There are many ways to do this including mean, mode, median, randomly choosing from a distribution, linear regression, and hot-decking, to name a few.
- Hot-decking is when you fill in the data based on similar observations. It can be applied to numerical and categorical data, unlike most of the other options listed above. Sequential hot-decking sorts the column with missing data based on an auxiliary column and then fills in the data with the value from the next available data point. K-Nearest Neighbors can also be used to identify similar data points.
- The last option is to flag the data as missing. This retains the information from missing data and removes the missing data (by replacing it). For categorical data, simply replace the data with a new category. For numerical data, we can fill the missing data with 0, or some value that makes sense, and add an indicator variable for missing data.

```
## Replace missing data
import numpy as np

# Add an indicator column based on missing Marriage_Len
>>> df['missing_ML'] = df['Marriage_Len'].isna()

# Fill in all missing data with 0
>>> df['Marriage_Len'] = df['Marriage_Len'].fillna(0)

# Change all other NaNs to missing
>>> df = df.fillna('missing')

# Change Not Applicable row to NaNs
>>> df = df.replace('Not Applicable', np.nan)

# Drop rows with NaNs
>>> df = df.dropna()

>>> df
```

	Name	Age	DOB	Marital_Status
0	JOHN DOE	30	01/01/2010	divorcee

1	JANE DOE	29	12/02/1990	divorced
2	JILL SMITH	40	03/04/1980	married
3	JACK SMITH	40	4/4/1980	married
4	JENNY SMITH	5	05/05/2015	missing

	Height	Weight	Marriage_Len	Spouse	missing_ML
0	72.0	175	5	missing	False
1	68.0	125	5	John Doe	False
2	64.0	120	10	Jack Smith	False
3	71.0	220	10	jill smith	False
4	41.0	40	0	missing	True

Nonnumerical Values Misencoded as Numbers

Recording data as a numerical data type (`float` or `int`) when no numerical meaning applies to the situation causes errors which can be extremely difficult to debug. Some data should be recorded in data types that cannot be multiplied or summed.

Missing data should always be stored in a form that cannot accidentally be incorporated into the model. Typically this is done by storing missing values as *NaN*. However, some algorithms will not run on data with *NaN* values, in which case you may choose to fill missing data with a string `'missing'`. Unfortunately, many datasets have recorded missing values with a 0 or some other number. You should verify that this does not occur in your dataset. Similarly, a survey with a scale from 1 to 5 will sometimes have the additional choice of “N/A” (meaning “not applicable”), which could be coded as 6, not because the value 6 is meaningful, but just because that is the next thing after 5. Again, this should be fixed so that the “N/A” choice cannot accidentally be used for any computations.

Categorical data are also often encoded as numerical values. These values should not be left as numbers that can be computed with. For example, postal codes are shorthand for locations, and there is no numerical meaning to the code. It makes no sense to add, subtract, or multiply postal codes, so it is important not to let those accidentally be added, subtracted, or multiplied, for example by inadvertently including them in the design matrix (unless they are one-hot encoded or given some other meaningful numerical value). It is good practice to convert postal codes, area codes, ID numbers, and other non-numeric data into strings or other data types that cannot be computed with.

Ordinal Data

Ordinal data is data that has a meaningful order but the differences between the values aren't consistent, or maybe aren't even meaningful at all. For example, a survey question might ask about your level of education, with 1 being high-school graduate, 2 bachelor's degree, 3 master's degree, and 4 doctoral degree. These values are called ordinal data because it is meaningful to talk about an answer of 1 being less than an answer of 2. However, the difference between 1 and 2 is not necessarily the same as the difference between 3 and 4, and it would not make sense to compute an average answer—the average of a high school diploma and a masters degree is not a bachelor's degree, despite the fact that the average of 1 and 3 is 2. Treating these like categorical data loses the information of the ordering, but treating it like regular numerical data implies that a difference of 2 has the same meaning whether it comes as $3 - 1$ or $4 - 2$. If that difference of 2 has approximately the same meaning, then it may be ok to treat these data as numerical in your model, but if that assumption is not correct then it may be better to treat the variable as categorical.

Problem 2. `imdb.csv` contains a small set of information about 99 movies. Clean the data set by doing the following in order:

1. Remove duplicate rows by dropping the first or last. Print the shape of the dataframe after removing the rows.
2. Drop all rows that contain missing data. Print the shape of the dataframe after removing the rows.
3. Remove rows that have data outside valid data ranges and explain briefly how you determined your ranges for each column.
4. Identify and drop columns with three or fewer different values. Print a tuple with the names of the columns dropped.
5. Convert the titles to all lower case.

Print the first five rows of your dataframe.

Feature Engineering

One often needs to construct new columns, commonly referred to as **features** in the context of machines learning, for a dataset, because the dependent variable is not necessarily a linear function of the features in the original dataset. Constructing new features is called *feature engineering*. Once new features are created, we can analyze how much a model depends on each feature. Features with low importance probably do not contributed much and could potentially be removed.

Fognets are fine mesh nets that collect water that condenses on the netting. These are used in some desert cities in Morocco to produce drinking water. Consider a dataset measuring the amount of water Y collected from fognets, where one of the features `WindDir` is the wind direction, measured in degrees. This feature is not likely to contribute meaningfully in a linear model because the direction 359 is almost the same as the direction 0, but no nonzero linear multiple of `WindDir` will reflect this relation. One way to improve the situation is to replace the `WindDir` with two new (engineered) features: $\sin\left(\frac{\pi}{180}\text{WindDir}\right)$ and $\cos\left(\frac{\pi}{180}\text{WindDir}\right)$.

Discrete Fourier transforms and wavelet decomposition often reveal important properties of data collected over time (*called time-series*), like sound, video, economic indicators, etc. In many such settings it is useful to engineer new features from a wavelet decomposition, the DFT, or some other function of the data.

Problem 3. `basketball.csv` contains data for all NBA players between 2001 and 2018. Each row represents a player's stats for a year. The features in this data set are

- `player` (str): the player's name
- `age` (int): the player's age
- `team_id` (cat): the player's team
- `per` (float): player efficiency rating, how much a player produced in one minute of play
- `ws` (float): win shares, an estimate of how much the player contributed to
- `bpm` (float): box plus/minus is the estimated number of points a player contributed to over 100 possessions
- `year` (int): the year

(float):

Create two new features:

- `career_length` (int): number of years player has been playing (start at 0).
- `target` (str): The target team if the player is leaving. If the player is retiring, the target should be 'retires'. A player is retiring if their name doesn't exist the next year. (Set the players in 2019 to NaN).

Remove all duplicates of a player in each year. Remove all rows except those where a player changes team, that is, `target` is not null nor 'retires'. Drop the `player`, `year`, and `team_id` columns.

Return the first ten lines of the dataframe.

Engineering for Categorical Variables

Categorical features are those that take only a finite number of values, and usually no categorical value has a numerical meaning, even if it happens to be number. For example in an election dataset, the names of the candidates in the race are categorical, and there is no numerical meaning (neither ordering nor size) to numbers assigned to candidates based solely on their names.

Consider the following election data.

Ballot number	For Governor	For President
001	Herbert	Romney
002	Cooke	Romney
003	Cooke	Obama
004	Herbert	Romney
005	Herbert	Romney
006	Cooke	Stein

A common mistake occurs when someone assigns a number to each categorical entry (say 1 for Cooke, 2 for Herbert, 3 for Romney, etc.). While this assignment is not, in itself, inherently incorrect, it is incorrect to use the value of this number in a statistical model. Any such model would be fundamentally wrong because a vote for Cooke cannot, in any reasonable way, be considered half of a vote for Herbert or a third of a vote for Romney. Many researchers have accidentally used categorical data in this way (and some have been very publicly embarrassed) because their categorical data was encoded numerically, which made it hard to recognize as categorical data.

Whenever you encounter categorical data that is encoded numerically like this, immediately change it either to non-numerical form (“Cooke,” “Herbert,” “Romney,”...) or apply a one-hot encoding as described below.

In order to construct a meaningful model with categorical data, one normally applies a *one-hot encoding* or *dummy variable encoding*.¹ To do this construct a new feature for every possible value of the categorical variable, and assign the value 1 to that feature if the variable takes that value and zero otherwise. Pandas makes one-hot encoding simple:

```
# one-hot encoding
df = pd.get_dummies(df, columns=['For President'])
```

The previous dataset, when the presidential race is one-hot encoded, becomes

Ballot number	Governor	Romney	Obama	Stein
001	Herbert	1	0	0
002	Cooke	1	0	0
003	Cooke	0	1	0
004	Herbert	1	0	0
005	Herbert	1	0	0
006	Cooke	0	0	1

Note that the sum of the terms of the one-hot encoding in each row is 1, corresponding to the fact that every ballot had exactly one presidential candidate.

When the gubernatorial race is also one-hot encoded, this becomes

Ballot number	Cooke	Herbert	Romney	Obama	Stein
001	0	1	1	0	0
002	1	0	1	0	0
003	1	0	0	1	0
004	0	1	1	0	0
005	0	1	1	0	0
006	1	0	0	0	1

¹Yes, these are silly names, but they are the most common names for it. Unfortunately, it is probably too late to change these now.

Now the sum of the terms of the one-hot encodings in each row is 2, corresponding to the fact that every ballot had two names—one gubernatorial candidate and one presidential candidate.

Summing the columns of the one-hot-encoded data gives the total number of votes for the candidate of that column. So the numerical values in the one-hot encodings are actually numerically meaningful, and summing the entries gives meaningful information. One-hot encoding also avoids the pitfalls of incorrectly using numerical proxies for categorical data.

The main disadvantage of one-hot encoding is that it is an inefficient representation of the data. If there are C categories and n datapoints, a one-hot encoding takes an $n \times 1$ -dimensional feature and turns it into an $n \times C$ sparse matrix. But there are ways to store these data efficiently and still maintain the benefits of the one-hot encoding.

ACHTUNG!

When performing linear regression, it is good practice to add a constant column to your dataset and to remove one column of the one-hot encoding of each categorical variable. (Adding a constant column should only be done in linear regression).

To see why, notice that summing terms in one row corresponding to the one-hot encoding of a specific categorical variable (for example the presidential candidate) always gives 1. If the dataset already has a constant column (which you really always should add if it isn't there already), then the constant column is a linear combination of the one-hot encoded columns. This cause the matrix to fail to be invertible and can cause identifiability problems.

The standard way to deal with this is to remove one column of the one-hot embedding for each categorical variable. For example, with the elections dataset above, we could remove the Cooke and Romney columns. Doing that means that in the new dataset a row sum of 0 corresponds to a ballot with a vote for Cooke and a vote for Romney, while a 1 in any column indicates how the ballot differed from the base choice of Cooke and Romney.

When using pandas, you can drop the first column of a one-hot encoding by passing in `drop_first=True`.

Problem 4. Load `housing.csv` into a dataframe with `index=0`. Descriptions of the features are in `housing_data_description.txt`. The goal is to construct a regression model that predicts `SalePrice` using the other features of the dataset. Do this as follows:

1. Identify and handle the missing data. Hint: Dropping every row with some missing data is not a good choice because it gives you an empty dataframe. What can you do instead?
2. Identify the variable with nonnumerical values that are misencoded as numbers. One-hot encode it. Hint: don't forget to remove one of the encoded columns to prevent collinearity with the constant column).
3. Add a constant column to the dataframe.
4. Save a copy of the dataframe.
5. Choose four categorical features that seem very important in predicting `SalePrice`. One-hot encode these features, and remove all other categorical features.

6. Run an OLS regression on your model.

Print the ten features that have the highest coefficient in your model and the summary. To run an OLS model in python, use the following code.

```
import statsmodels.api as sm

>>> results = sm.OLS(y, X).fit()

# Print the summary
>>> results.summary()

# Convert the summary table to a dataframe
>>> results_as_html = a.tables[1].as_html()
>>> result_df = pd.read_html(results_as_html, header=0, index_col=0)[0]
```

Problem 5. Using the copy of the dataframe you created in Problem 4, one-hot encode all the categorical variables. Print the shape of you database, and Run OLS.

Print the ten features that have the highest coefficient in your model and the summary. Write a couple of sentences discussing which model is better and why.