17. Data Pre-Processing

Learning Outcomes

- Understand numerical and categorical data.
- Deal with datasets with missing data.
- Deal with outliers in data.
- Normalize data.

Real data is often messy. It is therefore necessary to pre-process it before feeding it to a machine learning algorithm. We will think of data as a matrix where each row represents an observation, and the columns are the values of different variables that have been recorded for each observation. This is the standard way to handle data, and is how data is represented in the popular Python data science library `pandas`. Here is an example of a data set that we will use for illustration.

Example 17.1

	NAME	AGE	SEX	FAV_COL	WEIGHT	BIKES_TO_WORK
1	Adam	250	M	Blue	180	No
2	Becky	33	F	Blue		Yes
3	Claudia	18	F	Green	100	Sometimes
4	Dennis	45	M	Red	250	Yes
5	Eduardo	35	M	Green	166	No

We will write $\texttt{VAR}(m)$ to denote the value of variable `VAR` recorded in observation m. To exemplify, in the data set above $\texttt{SEX}(2) = \text{F}$ and $\texttt{BIKES\_TO\_WORK}(3) = \text{Sometimes}$

There are a number of problems with this dataset. First of all, many variables are not ready to be fed into a machine learning algorithm that does not understand "Green" or "Sometimes". There is also a missing entry in row 2 for the `WIGHT` variable, and the entry for `AGE` in row 1 is suspiciously high and may be the result of a measurement error. Finally, some algorithms work better if all variables are of comparable magnitude, which requires normalization. Below we will address these shortcomings one by one, following the process in Figure 17.

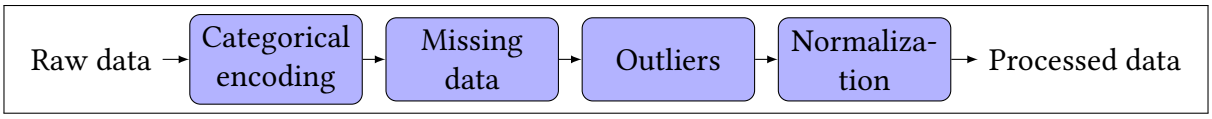

Figure 17.1.: Pre-processing pipeline

17.1. Types of Data

The first step is to understand the *type* of each variable. It can be either *numerical* or *categorical*. Machine learning algorithms expect numerical data; if the data set contains categorical variables we must therefore transform it into numerical data.

Numerical data is *quantitative* in nature, and the value has a sense of size or magnitude. Examples of numerical data include weights, distances, prices, and temperatures. On the other hand, categorical data is *qualitative* in nature—it distinguishes between categories. A variable that expresses the blood type of a patient, or the nationality of a person, are examples of categorical data. Sometimes categorical data is actually given by numbers, zip code is one such example that may appear numerical but is in fact categorical.

Since machine learning algorithms only understand numerical data, it is necessary to transform categorical data into a numerical representation. When a categorical variable takes a finite number of values the categorical variable can be represented by introducing dummy numerical variables. Two ways of introducing numerical variables are called **one-hut encoding** and **reference encoding**.

One-hut encoding is the most straightforward way to transform categorical data into numerical data. To convert a categorical variable `VAR` that varies between n different categories we introduce n dummy variables `D_VAR_`i and let

$$\texttt{D\_VAR\_i}(m) = \begin{cases} 1, & \text{if } \texttt{VAR(m)} \text{ is the } i\text{th category,} \\ 0, & \text{otherwise.} \end{cases} \tag{17.1}$$

While one-hut encoding is the most straightforward encoding method, it introduces *linear dependence*[1] into the data which poses a problem for certain machine learning algorithms. A remedy that avoids linear dependence is reference encoding, which lets one category be the reference category and introduces a dummy variable defined as (17.1) for each of the remaining $n-1$ categories. For this encoding the reference value is characterized by all dummy variables being equal to zero.

[1]The dummy variables sum to one for every row.

Example 17.2

We convert the categorical variable FAV_COLOR from Example 17.1 into numerical variables using one-hut encoding. While there are many different colors, only three appear as values of this variable. We therefore introduce three dummy variables D_BLUE, D_RED and D_GREEN for one-hut encoding.

	NAME	FAV_COL	D_BLUE	D_RED	D_GREEN
1	Adam	Blue	1	0	0
2	Becky	Blue	1	0	0
3	Claudia	Green	0	0	1
4	Dennis	Red	0	1	0
5	Eduardo	Green	0	0	1

We also illustrate reference encoding of the BIKES_TO_WORK variable. We let the reference category be "No" and introduce two dummy variable D_B and D_S as follows.

	NAME	BIKES_TO_WORK	D_B	D_BS
1	Adam	No	0	0
2	Becky	Yes	1	0
3	Claudia	Sometimes	0	1
4	Dennis	Yes	1	0
5	Eduardo	No	0	0

The categorical variable SEX can also be encoded via reference encoding using a single dummy variable IS_MALE.

17.2. Missing Data

In the example dataset above certain entries are missing, which is common in real-life datasets. Since algorithms typically expect data to be available for all entries, something must be done to eliminate missing data. The most obvious strategy is to exclude any row that contains missing data from the dataset. However, if a lot of data is absent this may exclude a significant portion of the dataset, and not enough may remain to perform a meaningful analysis. In that case there are different strategies for filling in the missing data fields.

A simple strategy is to fill in missing data fields with the averages of the variables over all other data rows. This makes sense since we introduce minimal artificial variance in the variable, but we can still capture meaningful relationships in the remaining variables. This may however introduce

various forms of biases that affect the outcome of learning algorithms[2]. A more advanced but similar method is to fill in missing data using the average of the k nearest neighbors rather than the average of all rows.

Example 17.3

We continue with our example; in the table below the missing value WEIGHT(2) has been filled in using the average rule: 174 is the average of the known weights 180, 100, 250 and 166. This may not be the best way since weight may correlate with other variables. A better idea would perhaps be to fill in with the average weight of all other females in the dataset.

	NAME	AGE	IS_MALE	D_BLUE	D_RED	D_GREEN	WEIGHT	D_B	D_BS
1	Adam	250	1	1	0	0	180	0	0
2	Becky	33	0	1	0	0	**174**	1	0
3	Claudia	18	0	0	0	1	100	0	1
4	Dennis	45	1	0	1	0	250	1	0
5	Eduardo	35	1	0	0	1	166	0	0

17.3. Outliers

Sometimes datasets contain *outliers*, which are values that stand out by being significantly larger or smaller than other values of the same variable. These can be the result of measurement errors or mistakes in manual data handling. Some algorithms are sensitive to extreme values so it can be a good idea to exclude outliers prior to analysis.

The first step in dealing with outliers is to identify them, which requires a principled rule. For example, we can mark all values that differ by more than three standard deviations from the statistical mean as outliers, or select some other rule that excludes obvious outliers. Detecting outliers is more of an art than science; the important thing is to apply the same rule to all rows in the dataset.

Once outliers have been detected, the next question is how to deal with them. Here it makes sense to treat outliers as missing data, and apply the same strategies. That is, we can either exclude data rows that contain outliers, or we fill in some value calculated from the remaining variables.

[2]This is especially problematic if missingness is not random but correlates with the value of the variable

Example 17.4

We marked the AGE in row 1 as an outlier since no human being has yet reached an age of more than 122 years. We chose to exclude the first row altogether, which leaves us with four rows of data with no obvious outliers.

	NAME	AGE	IS_MALE	D_BLUE	D_RED	D_GREEN	WEIGHT	D_B	D_BS
1	Becky	33	0	1	0	0	174	1	0
2	Claudia	18	0	0	0	1	100	0	1
3	Dennis	45	1	0	1	0	250	1	0
4	Eduardo	35	1	0	0	1	166	0	0

17.4. Normalization

Many machine learning algorithms work better if all variables take similar magnitudes. This is true for example for support vector machines and logistic regression, but not for example for classification trees. For algorithms that are sensitive to magnitudes it is a good idea to normalize the data prior to analysis. The most common way is to apply a translation and scaling that makes the statistical sample mean equal to zero and the sample variance equal to 1. This is achieved via the following transformation:

$$\texttt{VAR\_N}(m) = \frac{\texttt{VAR\_N}(m) - \mu(\texttt{VAR})}{\sigma(\texttt{VAR})}, \tag{17.2}$$

where the sample mean $\mu(\texttt{VAR})$ and sample variance $\sigma(\texttt{VAR})$ of VAR are given by:

$$\mu(\texttt{VAR}) = \frac{\sum_{m=1}^{M} \texttt{VAR(m)}}{M}, \quad \sigma(\texttt{VAR}) = \sqrt{\frac{\sum_{m=1}^{M} (\texttt{VAR}(m) - \mu)^2}{M-1}}. \tag{17.3}$$

In Python normalization of a numpy array can be done as follows:

Code 17.1: Normalizing a numpy array

```
import numpy as np

X = np.array([174, 100, 250, 166])
X_n = (X - np.mean(X))/np.std(X, ddof=1)

print(X_n)
# [ 0.02443308 -1.18093213  1.26237572 -0.10587667]

print(np.mean(X_n), np.std(X_n))
```

```
# 1.3877787807814457e-17 0.9999999999999999
```

Example 17.5

Below the variables AGE and WEIGHT have been replaced by their normalized counterparts AGE_N and WEIGHT_N that have sample mean zero and variance 1. The data is now ready to be fed into a machine learning algorithm

	AGE_N	IS_MALE	D_BLUE	D_RED	D_GREEN	WEIGHT_N	D_B	D_BS
2	**0.022**	0	1	0	0	**0.024**	1	0
3	**-1.323**	0	0	0	1	**-1.181**	0	1
4	**1.100**	1	0	1	0	**1.262**	1	0
5	**0.202**	1	0	0	1	**-0.106**	0	0

18. Retrieving Data From Web APIs

Learning Outcomes

- Be familiar with the HTTP GET method.
- JSON objects.
- Accessing APIs with Python `requests`.

When we browse the web our browser (Firefox, Chrome or Safari) sends data requests to remote servers via a protocol called Hypertext Transfer Protocol (HTTP) . The data that is sent back is used to render the websites in our browser. Suppose that we are working on a project that involves analysis of Twitter tweets. One way to obtain data would be to use our browser to search for and save tweets, but this manual task is laborious and does not scale well. Fortunately, many on-line services offer us a short-cut: we can programmatically request the data from servers directly in Python without having to go through the browser. Servers that provide data in this manner are commonly known as web Application Programming Interfaces (APIs). In this lecture we will learn how request data from web APIs, how to work with the data that is sent back, and how to use APIs that require authentication.

18.1. The HTTP protocol

The most important HTTP method is the GET method, which is used to request information from a remote server. A basic GET request contains an address and an optional set of headers. Browsers rely on GET to retrieve web site data that is used to render the webpage. In most browsers it is possible to inspect HTTP methods that are used when communicating with remote servers[1]. Figure 18.1 (left) shows a GET request sent when accessing google.com. The information contained in the header allows the server to tailor the response to this specific computer.

Code 18.1 shows an example of how to make a GET request to the Wikipedia API directly from python, using the `requests` library.

[1] In Firefox, this is under Web Developer -> Network, in Chrome it is under Developer Tools -> Network

Figure 18.1.: Left: A HTTP GET request sent in the Firefox browser. Right: The output of line 17 in Code 18.2 is a JSON object represented as a hierarchical Python `dict`.

Code 18.1: Making a HTTP GET request in Python [link]

```
1  import requests
2
3  url = 'https://en.wikipedia.org/w/api.php'
4  headers = {}
5  params = {
6    'action': 'query',
7    'titles': 'Albert Einstein',
8    'prop': 'pageviews',
9    'pvipdays': '5',
10   'format': 'xml',
11 }
12
13 res = requests.get(url, headers=headers, params=params)
14
15 print(res.text)
```

The `requests.get` method takes a URL and a `dict` that specifies headers. In addition, we can supply a dict `params` that consists of key-value pairs that are appended to the URL. That is, the request below is equivalent to a request to the URL `https://en.wikipedia.org/w/api.php?action=query&titles=Albert%20Einstein&prop=pageviews&pvipdays=5&format=xml` (try this address in a browser). The request returns a response with information about how many times the page about Albert Einstein has been viewed over the past five days. For the full speci-

fication of the Wikipedia API, see https://en.wikipedia.org/w/api.php?action=help.

In addition to GET, there are other HTTP methods like HEAD, POST, PUT, and DELETE, that are used for server communication.

18.2. JSON data

JavaScript is ubiquitous in the on-line world, many websites and service rely extensively on it. As a result, JavaScript Object Notation (JSON) has emerged as a de facto standard for data transfer, so most APIs return data in the form of a JSON object. This includes the Wikipedia API from which we can request JSON data by modifying the `format` parameter on line 11. In Python we can use the `json()` method of the `requests` library to convert JSON objects into familiar Python objects such as `dicts`, and `lists`. Code 18.2 contains the the same request as above, except that we specify on line 11 that we wish to get the result back as a JSON object. We then use the `json` package to pretty-print the result. The result is shown in Figure 18.1 (right). Since this is a hierarchy of Python dict objects, it is easy to access components in the response. Lines 20-21 show how we can extract the number of pageviews for each day in a for loop.

Code 18.2: Requesting and reading JSON data in Python [link]

```
1  import requests
2  import json
3
4  url = 'https://en.wikipedia.org/w/api.php'
5  headers = {}
6  params = {
7    'action': 'query',
8    'titles': 'Albert Einstein',
9    'prop': 'pageviews',
10   'pvipdays': '5',
11   'format': 'json',
12 }
13
14 res = requests.get(url, headers=headers, params=params).json()
15
16 # inspect complete response
17 print(json.dumps(res, indent=2))
18
19 # grab a property
20 for date, views in res['query']['pages']['736']['pageviews'].items():
21     print ("There were {} views on {}".format(views, date))
```

18.3. APIs that require authentication

Most online services provide API access to their systems so that developers can interface their code efficiently. Look up a service you have in mind, chances are it also has an API! For inspiration, a list of free APIs is maintained at https://github.com/toddmotto/public-apis.

However, many companies require authentication for API access. Accessing API endpoints that require authentication is a little bit more involved. Typically, it requires creating an account with the API provider, and then performing an initial token exchange before the API can be used as normal. Here is an example of how authentication is done to access the API of Twitter. First a HTTP POST request is sent that contains a private key `CON_KEY` and secret `CON_SECRET` encoded as specified by the Twitter documentation. The response to this first request contains an access token that can be included in the header of subsequent requests. As an example, we show a request that searches Twitter for the top five posts about "True Detective". Note that running this code requires your own key and secret on lines 5-6, which can be obtained by signing up for the Twitter developer program.

Code 18.3: Authenticating with the Twitter API

```
1  import base64
2  import requests
3  import json
4
5  CON_KEY = "MYKEY"
6  CON_SECRET = "MYSECRET"
7
8  # PREPARE ACCESS TOKEN REQUEST
9  auth_url = 'https://api.twitter.com/oauth2/token'
10
11 key_secret = '{}:{}'.format(CON_KEY, CON_SECRET).encode('ascii')
12 b64_encoded_key = base64.b64encode(key_secret)
13 b64_encoded_key = b64_encoded_key.decode('ascii')
14
15 auth_headers = {
16   'Authorization': 'Basic {}'.format(b64_encoded_key),
17   'Content-Type': 'application/x-www-form-urlencoded;charset=UTF-8'
18 }
19 auth_data = {
20   'grant_type': 'client_credentials'
21 }
22 # SEND ACCESS TOKEN REQUEST
23 auth_res = requests.post(auth_url, headers=auth_headers, data=auth_data)
24
25 # PREPARE SEARCH REQUEST
26 s_url = 'https://api.twitter.com/1.1/search/tweets.json'
27 access_token = auth_res.json()['access_token']
28
```

```
s_headers = {
  'Authorization': 'Bearer {}'.format(access_token)
}

s_params = {
  'q': 'True Detective',
  'count': 5,
  'tweet_mode': 'extended',
  'result_type': 'recent',
}
# SEND SEARCH REQUEST
s_res = requests.get(s_url, headers=s_headers, params=s_params)

print(json.dumps(s_res.json(), indent=2))
```