# Python Tutorial 2

### January 22, 2022

This tutorial is for Prof. Xin Tong's DSO 530 class at the University of Southern California in spring 2022. It aims to help you to learn how to preprocess data and build good training sets.

The quality of data and the amount of useful information that it contains are key factors that determine how well a machine learning algorithm can learn. Therefore, it is critical that we make sure to examine and preprocess a dataset before we feed it to a learning algorithm. In this tutorial, we will discuss the essential data preprocessing techniques that will help us build good machine learning models.

The topics that we will cover in this tutorial are as follows: • Removing and imputing missing values from the dataset • Getting categorical data into shape for machine learning algorithms • Partitioning a dataset into separate training and test sets • Bringing features onto the same scale

## 1 Dealing with missing data

It is not uncommon in real-world applications for our samples to be missing one or more values for various reasons. There could have been an error in the data collection process, certain measurements are not applicable, or particular fields could have been simply left blank in a survey. We typically see missing values as the blank spaces in our data table or as placeholder strings such as NaN, which stands for not a number, or NULL (a commonly used indicator of unknown values in relational databases).

Unfortunately, most computational tools are unable to handle such missing values, or produce unpredictable results if we simply ignore them. Therefore, it is crucial that we take care of those missing values before we proceed with further analyses. In this section, we will work through several practical techniques for dealing with missing values by removing entries from our dataset or imputing missing values from other samples and features.

### 1.1 Identifying missing values in tabular data

But before we discuss several techniques for dealing with missing values, let's create a simple example data frame from a Comma-separated Values (CSV) file to get a better grasp of the problem:

```
[1]: import pandas as pd
     from io import StringIO
     csv_data = \
     '''A,B,C,D
      1.0,2.0,3.0,4.0
      5.0,6.0,,8.0
      10.0,11.0,12.0,'''
```

```
df = pd.read_csv(StringIO(csv_data))
df
```

[1]:
```
      A     B     C    D
0   1.0   2.0   3.0  4.0
1   5.0   6.0   NaN  8.0
2  10.0  11.0  12.0  NaN
```

Here, we use StringIO to read strings as a file. We also use the backslash (\) to indicate that a statement is continued on the next line. In this way, we split a statement into multiple lines in Python. A, B, C and D are different features of the observations.

Using the preceding code, we read CSV-formatted data into a pandas DataFrame via the read_csv function and noticed that the two missing cells were replaced by NaN. The StringIO function in the preceding code example was simply used for the purposes of illustration. It allows us to read the string assigned to csv_data into a pandas DataFrame as if it was a regular CSV file on our hard drive.

For a larger DataFrame, it can be tedious to look for missing values manually; in this case, we can use the isnull method to return a DataFrame with Boolean values that indicate whether a cell contains a numeric value (False) or if data is missing (True).

[2]:
```
df.isnull()
```

[2]:
```
       A      B      C      D
0  False  False  False  False
1  False  False   True  False
2  False  False  False   True
```

notnull is the negation of isnull.

[3]:
```
df.notnull()
```

[3]:
```
      A     B      C      D
0  True  True   True   True
1  True  True  False   True
2  True  True   True  False
```

Using the sum method, we can then return the number of missing values per column as follows:

[4]:
```
df.isnull().sum()
```

[4]:
```
A    0
B    0
C    1
D    1
dtype: int64
```

This way, we can count the number of missing values per column; in the following subsections, we will take a look at different strategies for dealing with these missing data.

P.S. Although scikit-learn was developed for working with NumPy arrays, it can sometimes be more convenient to preprocess data using pandas' DataFrame. We can always access the underlying NumPy array of a DataFrame via the values attribute before we feed it into a scikit-learn estimator:

```
[5]: df.values
```

```
[5]: array([[ 1.,  2.,  3.,  4.],
            [ 5.,  6., nan,  8.],
            [10., 11., 12., nan]])
```

The main API implemented by scikit-learn is that of the estimator. An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a transformer that extracts or filters useful features from raw data.

Note that estimator in the field of Statistics means something different.

## 1.2 Eliminating observations or features with missing values

One of the easiest ways to deal with missing data is to simply remove the corresponding features (columns) or observations (rows) from the dataset entirely; rows with missing values can be easily dropped via the dropna method:

```
[6]: df.dropna(axis=0)
```

```
[6]:      A    B    C    D
      0  1.0  2.0  3.0  4.0
```

If we just use df.dropna(), it is the same as to specify that axis = 0.

Similarly, we can drop columns that have at least one NaN in any row by setting the axis argument to 1:

```
[7]: df.dropna(axis=1)
```

```
[7]:       A     B
      0   1.0   2.0
      1   5.0   6.0
      2  10.0  11.0
```

The dropna method supports several additional parameters that can come in handy:

```
[8]: # only drop rows where all columns are NaN
     # (returns the whole array here since we don't
     # have a row with where all values are NaN
     df.dropna(how='all')
```

```
[8]:       A      B      C     D
     0   1.0    2.0    3.0   4.0
     1   5.0    6.0    NaN   8.0
     2  10.0   11.0   12.0   NaN
```

```
[9]: # drop rows that have less than 4 real values
     df.dropna(thresh=4)
```

```
[9]:      A      B     C    D
     0  1.0    2.0   3.0  4.0
```

```
[10]: # only drop rows where NaN appear in specific columns (here: 'C')
      df.dropna(subset=['C'])
```

```
[10]:       A      B      C     D
      0   1.0    2.0    3.0   4.0
      2  10.0   11.0   12.0   NaN
```

Although the removal of missing data seems to be a convenient approach, it also comes with certain disadvantages; for example, we may end up removing too many observations, which might make reliable analysis impossible. Or, if we remove too many feature columns, we will run the risk of losing valuable information that our classifier needs to discriminate between classes. In the next section, we will thus look at one of the most commonly used alternatives for dealing with missing values: imputation techniques.

## 1.3 Imputing missing values

### 1.3.1 Use sklearn.impute.SimpleImputer

https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html#sklearn.impute.SimpleIm

Often, the removal of entire rows or dropping of entire columns is simply not feasible, because we might lose too much valuable data. In this case, we can use different imputation techniques to estimate the missing values from the other non-null entries in our training data. One of the most common imputation techniques is mean imputation, where we simply replace the missing value with the mean value of the entire feature column. A convenient way to achieve this is by using the SimpleImputer class from scikit-learn, as shown in the following code:

```
[11]: import numpy as np
      from sklearn.impute import SimpleImputer
      imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
      imp = imp.fit(df.values)
      imputed_data = imp.transform(df.values)
      imputed_data
```

```
[11]: array([[ 1. ,  2. ,  3. ,  4. ],
             [ 5. ,  6. ,  7.5,  8. ],
             [10. , 11. , 12. ,  6. ]])
```

4

Here, we replaced each NaN value with the corresponding column mean. Other options for the strategy parameter are median, constant or most_frequent, where the last replaces the missing values with the most frequent values. This is useful for imputing categorical feature values, for example, a feature column that stores an encoding of color names, such as red, green, and blue, and we will encounter examples of such data later in this tutorial.

When strategy == "constant", there is another parameter named fill_value, which is used to replace all occurrences of missing_values. If left to the default, fill_value will be 0 when imputing numerical data and "missing_value" for strings or object data types.

### 1.3.2   Use sklearn.impute.KNNImputer

https://scikit-learn.org/stable/modules/generated/sklearn.impute.KNNImputer.html#sklearn-impute-knnimputer

The KNNImputer class provides imputation for filling in missing values using the k-Nearest Neighbors approach.

By default, a euclidean distance metric that supports missing values, nan_euclidean_distances, is used to find the nearest neighbors. Each observation's missing values are imputed using the mean values from n_neighbors nearest neighbors found in the training data sets. Two observations are close if the features that neither is missing are close.

The following snippet demonstrates how to replace missing values, encoded as np.nan, using the mean feature value of the two nearest neighbors of samples with missing values:

```python
[12]: import numpy as np
      from sklearn.impute import KNNImputer
      X = [[1, 2, np.nan], [3, 4, 3], [np.nan, np.nan, 5], [8, 8, 7]]
      df = pd.DataFrame(X, columns=['A', 'B', 'C'])
      df
```

```
[12]:     A    B    C
      0  1.0  2.0  NaN
      1  3.0  4.0  3.0
      2  NaN  NaN  5.0
      3  8.0  8.0  7.0
```

Handwritten annotations:

$dist^2(0,1) = [(1-3)^2 + (2-4)^2](3/2) = 12 \leftarrow$ closest 2 rows to row ⓪.

$dist^2(0,2) = \infty$

$dist^2(0,3) = [(1-8)^2 + (2-8)^2](3/2) = 127.5 \leftarrow$ ∴ avg of $\frac{3 \& 7}{2} = 5$.

We set n_neighbors to 2, which means that the number of neighboring observations to use for imputation is 2. n_neighbors' default value is 5.

We also set weights to "uniform", which means that all points in each neighborhood are weighted equally. Besides, weights can be also set to "distance", which means that we weight points by the inverse of their distance. In other word, closer neighbors of a query point will have a greater influence than neighbors which are further away. weights' default value is "uniform" :

```python
[13]: imputer = KNNImputer(n_neighbors=2, weights="uniform")
      imputer.fit_transform(X) # you will get a np array
```

```
[13]: array([[1. , 2. , 5. ],
             [3. , 4. , 3. ],
```

$$(a_1, \ldots a_p) \ldots (b_1 \ldots b_p) \implies \sqrt{(a_1-b_1)^2 + \ldots (a_p-b_p)^2}$$

```
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

One example about "distance" mentioned above is computed as the following way:

the distance between $[3, np.nan, np.nan, 6]$ and $[1, np.nan, 4, 5]$ is $\sqrt{\frac{4}{2}\left((3-1)^2 + (6-5)^2\right)}$

*[handwritten annotations: "total no. of words", "no. of words where no value is null."]*

## 1.4 Understanding the scikit-learn estimator API

In the previous section, we used the SimpleImputer and KNNImputer classes from scikit-learn to impute missing values in our dataset. The SimpleImputer and KNNImputer classes belong to the so-called transformer classes in scikit-learn, which are used for data transformation. The two essential methods of those estimators are fit and transform. The fit method is used to learn the parameters from the training data, and the transform method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model. Figure 1-1 illustrates how a transformer, fitted on the training data, is used to transform a training dataset as well as a new test dataset:

[14]:
```python
from IPython.display import Image
%matplotlib inline
Image(filename='Figure1-1.png', width=400)
```

[14]:
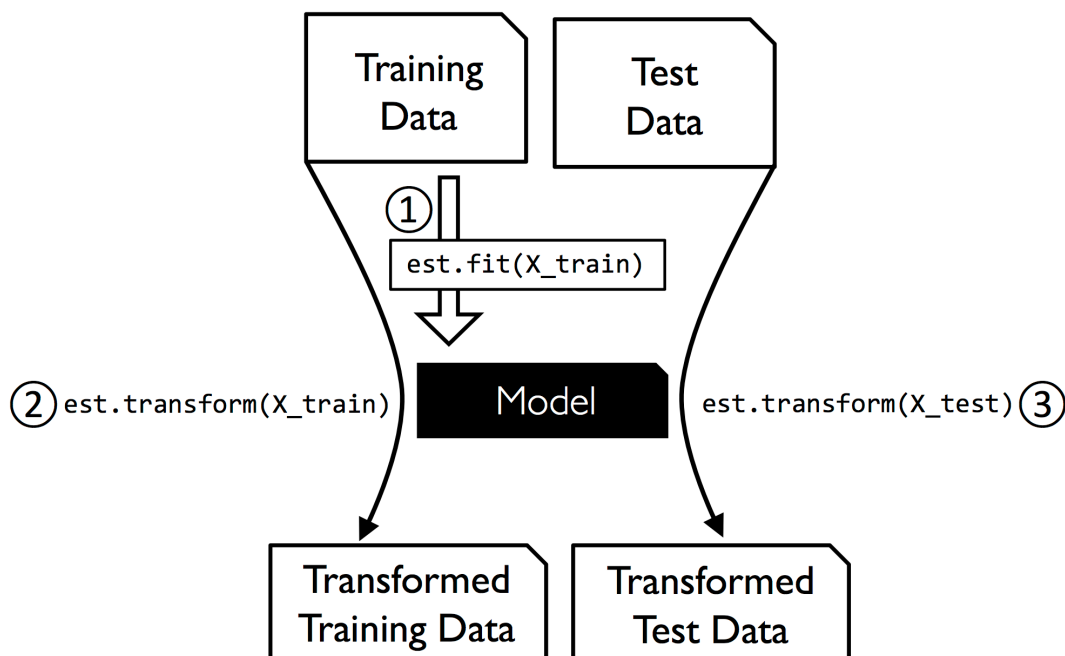


We give a simple demonstration to show that the missing values of training data and test data are both imputed using the Model trained by training data. The training data is as follows and A, B,

C and D are as different features of the observations:

```
[15]: data_train = {'A': [3, 2, np.nan, 4, 3], 'B': [3, np.nan, 4, 4, 5], 'C':[np.
      ↪nan, 4.8, 5.1, 4.9, 5.2], 'D':[6, 7, 9, np.nan, 10]}
      df2_train = pd.DataFrame(data = data_train)
      df2_train
```

```
[15]:      A    B    C     D
      0  3.0  3.0  NaN   6.0
      1  2.0  NaN  4.8   7.0
      2  NaN  4.0  5.1   9.0
      3  4.0  4.0  4.9   NaN
      4  3.0  5.0  5.2  10.0
```

The mean of df2_train is as follows:

```
[16]: df2_train.mean(axis = 0)
```

```
[16]: A    3.0
      B    4.0
      C    5.0
      D    8.0
      dtype: float64
```

We use the mean imputation technique to impute the missing data. First, we use fit to build the model imp2 (in this case, calculating the mean value of each column in the training data sets), then we use transform to impute the missing data in the training data sets. We could see that the imputed values are the mean of each column in the original training data sets.

```
[17]: imp2 = SimpleImputer(missing_values = np.nan, strategy = 'mean')
      imp2 = imp2.fit(df2_train)
      imputed_df2_train = imp2.transform(df2_train) # old version need "df2_train" to␣
      ↪be "df2_train.values"
      imputed_df2_train
```

```
[17]: array([[ 3. ,  3. ,  5. ,  6. ],
             [ 2. ,  4. ,  4.8,  7. ],
             [ 3. ,  4. ,  5.1,  9. ],
             [ 4. ,  4. ,  4.9,  8. ],
             [ 3. ,  5. ,  5.2, 10. ]])
```

Our test data df2_test is as follows:

```
[18]: data_test = {'A': [2, np.nan, 3], 'B': [4, 5, np.nan], 'C':[np.nan, 4, 5], 'D':
      ↪[7, 8, np.nan]}
      df2_test = pd.DataFrame(data = data_test)
      df2_test
```

```
[18]:      A    B    C    D
      0  2.0  4.0  NaN  7.0
      1  NaN  5.0  4.0  8.0
      2  3.0  NaN  5.0  NaN
```

The mean of df2_test is as follows:

```
[19]: df2_test.mean(axis = 0)
```

```
[19]: A    2.5
      B    4.5
      C    4.5
      D    7.5
      dtype: float64
```

Use imp2.transform to impute the missing data in the test data sets. We can see that the imputed values are the means of the original training data instead of the test data.

```
[20]: imputed_df2_test = imp2.transform(df2_test)
      imputed_df2_test
```

```
[20]: array([[2., 4., 5., 7.],
             [3., 5., 4., 8.],
             [3., 4., 5., 8.]])
```

# 2 Handling categorical data

So far, we have only been working with numerical values. However, it is not uncommon that real-world datasets contain one or more categorical feature columns. In this section, we will make use of simple yet effective examples to see how we deal with this type of data in numerical computing libraries.

## 2.1 Nominal and ordinal features

When we are talking about categorical data, we have to further distinguish between nominal and ordinal features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, t-shirt size would be an ordinal feature, because we can define an order XL > L > M. In contrast, nominal features don't imply any order and, to continue with the previous example, we could think of t-shirt color as a nominal feature since it typically doesn't make sense to say that, for example, red is larger than blue.

### 2.1.1 Creating an example dataset

Before we explore different techniques to handle such categorical data, let's create a new DataFrame to illustrate the problem:

```
[21]: import pandas as pd
```

```
df = pd.DataFrame([['green', 'M', 10.1, 'class1'],
                   ['red', 'L', 13.5, 'class2'],
                   ['blue', 'XL', 15.3, 'class1']])

df.columns = ['color', 'size', 'price', 'classlabel']
df
```

[21]:     color size  price classlabel
      0  green   M   10.1     class1
      1    red   L   13.5     class2
      2   blue  XL   15.3     class1

As we can see in the preceding output, the newly created DataFrame contains a nominal feature (color), an ordinal feature (size), and a numerical feature (price) column. The class labels (assuming that we created a dataset for a supervised learning task) are stored in the last column.

## 2.2 Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our size feature, so we have to define the mapping manually. In the following simple example, let's assume that we know the numerical difference between features, for example, $XL = L + 1 = M + 2$:

[22]:
```
# we define the mapping manually using the dictionary
size_mapping = {'XL': 3,
                'L': 2,
                'M': 1}

df['size'] = df['size'].map(size_mapping)
df
```

[22]:     color  size  price classlabel
      0  green     1   10.1     class1
      1    red     2   13.5     class2
      2   blue     3   15.3     class1

If we want to transform the integer values back to the original string representation at a later stage, we can simply define a reverse-mapping dictionary inv_size_ mapping = {v: k for k, v in size_mapping.items()} that can then be used via the pandas map method on the transformed feature column, similar to the size_mapping dictionary that we used previously. We can use it as follows:

[23]:
```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
df['size'].map(inv_size_mapping)
```

[23]: 0    M
      1    L

9

```
2    XL
Name: size, dtype: object
```

P.S. Only domain experts can do this numerical conversion from ordinal to numerical. If we do not know this conversion, we will have to let go of the order information and just convert the variable as if it is just the usual categorical variable, or nominal variable.

## 2.3   Encoding class labels

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in scikit-learn convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously. We need to remember that class labels are not ordinal, and it doesn't matter which integer number we assign to a particular string label. Thus, we can simply enumerate the class labels, starting at 0.

There is a convenient LabelEncoder class directly implemented in scikit-learn to achieve this:

```python
[24]: from sklearn.preprocessing import LabelEncoder

      # Label encoding with sklearn's LabelEncoder
      class_le = LabelEncoder()
      y = class_le.fit_transform(df['classlabel'].values)
      y
```

```
[24]: array([0, 1, 0])
```

```python
[25]: df['classlabel'] = y
      df
```

```
[25]:    color  size  price  classlabel
      0  green     1   10.1           0
      1    red     2   13.5           1
      2   blue     3   15.3           0
```

We can use the inverse_transform method to transform the integer class labels back into their original string representation:

```python
[26]: # reverse mapping
      df['classlabel'] = class_le.inverse_transform(y)
      df
```

```
[26]:    color  size  price classlabel
      0  green     1   10.1     class1
      1    red     2   13.5     class2
      2   blue     3   15.3     class1
```

Note that the fit_transform method is just a shortcut for calling fit and transform separately:

```
[27]: class_le2 = LabelEncoder()
      class_le2 = class_le2.fit(df['classlabel'].values)
      y2 = class_le2.transform(df['classlabel'].values)
      y2
```

[27]: array([0, 1, 0])

## 2.4 Performing one-hot encoding on nominal features

In the previous section, we used a simple dictionary-mapping approach to convert the ordinal size feature into integers. Since scikit-learn's estimators for classification treat class labels as categorical data that does not imply any order (nominal), we used the convenient LabelEncoder to encode the string labels into integers. It may appear that we could use a similar approach to transform the nominal color column of our dataset, as follows:

```
[28]: X = df[['color', 'size', 'price']].values

      X
```

[28]: array([['green', 1, 10.1],
             ['red', 2, 13.5],
             ['blue', 3, 15.3]], dtype=object)

```
[29]: color_le = LabelEncoder()
      X[:, 0] = color_le.fit_transform(X[:, 0])
      X
```

[29]: array([[1, 1, 10.1],
             [2, 2, 13.5],
             [0, 3, 15.3]], dtype=object)

After executing the preceding code, the first column of the NumPy array X now holds the new color values, which are encoded as follows:

- blue $= 0$

- green $= 1$

- red $= 2$

If we stop at this point and feed the array to our classifier, we will make one of the most common mistakes in dealing with categorical data. Can you spot the problem? Although the color values don't come in any particular order, a learning algorithm will now assume that green is larger than blue, and red is larger than green.

A common workaround for this problem is to use a technique called one-hot encoding. The idea behind this approach is to create a new dummy feature for each unique value in the nominal feature column. Here, we would convert the color feature into three new features: blue, green, and red. Binary values can then be used to indicate the particular color of a sample; for example, a blue t-shirt can be encoded as blue=1, green=0, red=0. To perform this transformation, we can use the

11

OneHotEncoder that is implemented in the scikit-learn.preprocessing module and we also need to import ColumnTransformer from sklearn.compose to specify which column to transform. Here we use ColumnTransformer to specify that we are transforming the [0] column :

```
[30]: from sklearn.preprocessing import OneHotEncoder
      from sklearn.compose import ColumnTransformer

      X = df[['color', 'size', 'price']].values
      ct = ColumnTransformer([("color",OneHotEncoder(), [0])], remainder =␣
       ↪'passthrough') # the "color" parameter seems to be redundant, but it must␣
       ↪exist when using this package
      X = ct.fit_transform(X)


      X
```

```
[30]: array([[0.0, 1.0, 0.0, 1, 10.1],
             [0.0, 0.0, 1.0, 2, 13.5],
             [1.0, 0.0, 0.0, 3, 15.3]], dtype=object)
```

An even more convenient way to create those dummy features via one-hot encoding is to use the get_dummies method implemented in pandas. Applied to a DataFrame, the get_dummies method will only convert string columns and leave all other columns unchanged:

```
[31]: # one-hot encoding via pandas
      pd.get_dummies(df[['price', 'color', 'size']])
```

```
[31]:    price  size  color_blue  color_green  color_red
      0   10.1     1           0            1          0
      1   13.5     2           0            0          1
      2   15.3     3           1            0          0
```

When we are using one-hot encoding datasets, we have to keep in mind that it introduces multi-collinearity, which can be an issue for certain methods (for instance, methods that require matrix inversion). If features are highly correlated, matrices are computationally difficult to invert, which can lead to numerically unstable estimates. To reduce the correlation among variables, we can simply remove one feature column from the one-hot encoded array. Note that we do not lose any important information by removing a feature column, though; for example, if we remove the column color_blue, the feature information is still preserved since if we observe color_green=0 and color_red=0, it implies that the observation must be blue. For a nominal variable with K categories, we only need K-1 dummies.

If we use the get_dummies function, we can drop the first column by passing a True argument to the drop_first parameter, as shown in the following code example:

```
[32]: # multicollinearity guard in get_dummies
      pd.get_dummies(df[['price', 'color', 'size']], drop_first=True)
```

```
[32]:    price  size  color_green  color_red
      0   10.1     1            1          0
      1   13.5     2            0          1
      2   15.3     3            0          0
```

If we use the OneHotEncoder and ColumnTransformer function, we can drop the first column by passing a first argument to the drop parameter in OneHotEncoder, as shown in the following code example:

```
[33]: X = df[['color', 'size', 'price']].values
      ct = ColumnTransformer([("color", OneHotEncoder(drop = 'first'), [0])],␣
      ↪remainder = 'passthrough')
      X = ct.fit_transform(X)

      X
```

```
[33]: array([[1.0, 0.0, 1, 10.1],
             [0.0, 1.0, 2, 13.5],
             [0.0, 0.0, 3, 15.3]], dtype=object)
```

# 3   Partitioning a dataset into separate training and test sets

We aim to build a machine learning model from some data that can achieve our goal. But before we can apply our model to new measurements, we need to know whether it actually works—that is, whether we should trust its predictions.

Unfortunately, we cannot use the data we used to build the model to evaluate it. This is because our model can always simply remember the whole training set, and will therefore always predict the correct label for any point in the training set. This "remembering" does not indicate to us whether our model will generalize well (in other words, whether it will also perform well on new data).

To assess the model's performance, we show it new data (data that it hasn't seen before) for which we have labels. This is usually done by splitting the labeled data we have collected into two parts. One part of the data is used to build our machine learning model, and is called the training data or training set. The rest of the data will be used to assess how well the model works; this is called the test data, test set, or hold-out set.

In this section, we will prepare a new dataset, the Wine dataset. After we have preprocessed the dataset, we will explore different techniques for feature selection to reduce the dimensionality of a dataset.

The Wine dataset is another open-source dataset that is available from the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/Wine); it consists of 178 wine observations with 13 features describing their different chemical properties.

Using the pandas library, we will directly read in the open-source Wine dataset from the UCI machine learning repository:

```
[34]: df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
       ↪'machine-learning-databases/wine/wine.data', header=None)

      # if the Wine dataset is temporarily unavailable from the
      # UCI machine learning repository, un-comment the following line
      # of code to load the dataset from a local path:

      # df_wine = pd.read_csv('wine.data', header=None)

      df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                         'Alcalinity of ash', 'Magnesium', 'Total phenols',
                         'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                         'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
                         'Proline']

      print('Class labels', np.unique(df_wine['Class label']))
      df_wine.head()
```

Class labels [1 2 3]

```
[34]:    Class label  Alcohol  Malic acid   Ash  Alcalinity of ash  Magnesium  \
      0            1    14.23        1.71  2.43               15.6        127
      1            1    13.20        1.78  2.14               11.2        100
      2            1    13.16        2.36  2.67               18.6        101
      3            1    14.37        1.95  2.50               16.8        113
      4            1    13.24        2.59  2.87               21.0        118

         Total phenols  Flavanoids  Nonflavanoid phenols  Proanthocyanins  \
      0           2.80        3.06                  0.28             2.29
      1           2.65        2.76                  0.26             1.28
      2           2.80        3.24                  0.30             2.81
      3           3.85        3.49                  0.24             2.18
      4           2.80        2.69                  0.39             1.82

         Color intensity   Hue  OD280/OD315 of diluted wines  Proline
      0             5.64  1.04                          3.92     1065
      1             4.38  1.05                          3.40     1050
      2             5.68  1.03                          3.17     1185
      3             7.80  0.86                          3.45     1480
      4             4.32  1.04                          2.93      735
```

The 13 different features in the Wine dataset, describing the chemical properties of the 178 wine samples, are listed in the above table.

A bottle of wine in the collection belongs to one of three different classes, 1, 2, and 3, which refer to the three different types of grape grown in the same region in Italy but derived from different wine cultivars, as described in the dataset summary (https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.names).

A convenient way to randomly partition this dataset into separate training and test datasets is to use the train_test_split function from scikit-learn's model_selection submodule:

```
[35]: from sklearn.model_selection import train_test_split

      X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

      X_train, X_test, y_train, y_test =\
          train_test_split(X, y,
                           test_size=0.3,
                           random_state=0,
                           stratify=y)
```

*[handwritten annotations: → 70%; 3 classes; c1 c2 c3; training data has to be enough for good model; prop of classes in training = prop of classes in test set]*

First, we assigned the NumPy array representation of the feature columns 1-13 to the variable X; we assigned the class labels from the first column to the variable y. Then, we used the train_test_split function to randomly split X and y into separate training and test datasets. By setting test_size=0.3 (the default value of test_size is 0.25), we assigned 30 percent of the wine samples to X_test and y_test, and the remaining 70 percent of the samples were assigned to X_train and y_train, respectively. We use the random_state parameter to set the random seed, which is important for the reproducibility of the results. Providing the class label array y as an argument to stratify ensures that both training and test datasets have the same class proportions as the original dataset.

P.S. If we are dividing a dataset into training and test datasets, we have to keep in mind that we are withholding valuable information that the learning algorithm could benefit from. Thus, we don't want to allocate too much information on the test set. However, the larger the test set, the more inaccurate the estimation of the generalization error. Dividing a dataset into training and test sets is all about balancing this trade-off. In practice, the most commonly used splits are 60:40, 70:30, or 80:20, depending on the size of the initial dataset. However, for large datasets, 90:10 or 99:1 splits into training and test subsets are also common and appropriate. Instead of discarding the allocated test data after model training and evaluation, it is a common practice to retrain a classifier on the entire dataset as it can improve the predictive performance of the model.

## 4 Bringing features onto the same scale

*[handwritten annotations: → decreases interpretability; → not required for linear models]*

Feature scaling is a crucial step in our preprocessing pipeline that can easily be forgotten. In this tutorial, we will not discuss when and where we should take a feature scaling step because whether bringing different features onto the same scale can improve our model is subtle and that should be discussed case by case. Here we only introduce two approaches to feature scaling.

Now, there are two common approaches to bring different features onto the same scale: normalization and standardization. These terms are often used quite loosely in different fields, and the meaning has to be derived from the context. Most often, normalization refers to the rescaling of the features to a range of $[0, 1]$, which is a special case of min-max scaling. To normalize our data, we can simply apply the min-max scaling to each feature column, where the new value $x_{norm}^{(i)}$ of a sample $x^{(i)}$ can be calculated as follows:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

15

Here, $x^{(i)}$ is the i-th observation, $x_{min}$ is the smallest value in a feature column, and $x_{max}$ is the largest value.

The min-max scaling procedure is implemented in scikit-learn and can be used as follows:

```
[36]: from sklearn.preprocessing import MinMaxScaler

      mms = MinMaxScaler()
      X_train_norm = mms.fit_transform(X_train)
      X_test_norm = mms.transform(X_test)
```

P.S. Note that if $x_{min}$ in the test data is smaller than $x_{min}$ in the training data, then the transformed test point might have a value smaller than 0.

```
[37]: ex_train = np.array([3, 4, 5, 6, 7, 8]).reshape(6, 1)
      ex_train
```

```
[37]: array([[3],
             [4],
             [5],
             [6],
             [7],
             [8]])
```

```
[38]: mms_ex = MinMaxScaler()
      ex_train_norm = mms_ex.fit_transform(ex_train)
      ex_train_norm
```

```
[38]: array([[0. ],
             [0.2],
             [0.4],
             [0.6],
             [0.8],
             [1. ]])
```

```
[39]: ex_test = np.array([2, 3, 4, 5]).reshape(4, 1)
      ex_test
```

```
[39]: array([[2],
             [3],
             [4],
             [5]])
```

```
[40]: ex_test_norm = mms_ex.transform(ex_test)
      ex_test_norm
```

```
[40]: array([[-0.2],
             [ 0. ],
```

```
        [ 0.2],
        [ 0.4]])
```

Using standardization, we center the feature columns at mean 0 with standard deviation 1.

The procedure for standardization can be expressed by the following equation:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Here, $\mu_x$ is the sample mean of a particular feature column and $\sigma_x$ corresponding standard deviation.

Similar to the MinMaxScaler class, scikit-learn also implements a class for standardization:

```
[41]: from sklearn.preprocessing import StandardScaler

      stdsc = StandardScaler()
      X_train_std = stdsc.fit_transform(X_train)
      X_test_std = stdsc.transform(X_test)
```

Again, it is also important to highlight that we fit the StandardScaler class only once—on the training data—and use those parameters to transform the test set or any new data point.

The following table illustrates the difference between the two commonly used feature scaling techniques, standardization and normalization, on a simple sample dataset consisting of numbers 0 to 5:

| Input | Standardized | Min-max normalized |
|-------|--------------|--------------------|
| 0.0   | -1.46385     | 0.0                |
| 1.0   | -0.87831     | 0.2                |
| 2.0   | -0.29277     | 0.4                |
| 3.0   | 0.29277      | 0.6                |
| 4.0   | 0.87831      | 0.8                |
| 5.0   | 1.46385      | 1.0                |

You can perform the standardization and normalization shown in the table manually by executing the following code examples:

```
[42]: ex = np.array([0, 1, 2, 3, 4, 5])
      ex
```

```
[42]: array([0, 1, 2, 3, 4, 5])
```

stdsc.fit_transform and mms.fit_transform both expect 2D array, but ex is an 1D array. Thus, we use array.reshape (-1,1) to reshape the 1D array to a 2D array.

array.reshape (-1,k) means that we reshape the array into k columns and the number of rows will be inferred automatically. Similarly, array.reshape (k,-1) means that we reshape the array into k rows and the number of columns will be inferred automatically.

```
[43]: ex_std = stdsc.fit_transform(ex.reshape(-1,1))
      ex_norm = mms.fit_transform(ex.reshape(-1, 1))
      print('standardized:', ex_std.reshape(1,-1))
      print('normalized:', ex_norm.reshape(1,-1))
```

```
standardized: [[-1.46385011 -0.87831007 -0.29277002  0.29277002  0.87831007
  1.46385011]]
normalized: [[0.  0.2 0.4 0.6 0.8 1. ]]
```

We can use the formula of definition to validate the results:

```
[44]: print('standardized:', (ex - ex.mean()) / ex.std())
      print('normalized:', (ex - ex.min()) / (ex.max() - ex.min()))
```

```
standardized: [-1.46385011 -0.87831007 -0.29277002  0.29277002  0.87831007
  1.46385011]
normalized: [0.  0.2 0.4 0.6 0.8 1. ]
```

References:

Müller, Andreas C; Guido, Sarah. (2017). Introduction to Machine Learning with Python.

Raschka, S. (2017). Python Machine Learning.

https://scikit-learn.org/stable/index.html

https://archive.ics.uci.edu/ml/datasets.php

https://en.wikipedia.org/wiki/