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

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.

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:

In [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
```

Out[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. 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).

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

Out[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.

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

Out[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:

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

Out[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:

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

Out[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.

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:

In [6]:
```
df
```

Out[6]:

	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

In [7]:
```
df.dropna(axis=0) #0 denote row
```

Out[7]:

	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:

In [8]:
```
df.dropna(axis=1) #1 denote column
```

Out[8]:

	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:

In [9]:
```
# 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')
df.dropna(how='all')
```

Out[9]:

	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

In [10]:
```
# drop rows where any columns are NaN
df.dropna(how='any')
```

Out[10]:

	A	B	C	D
0	1.0	2.0	3.0	4.0

In [11]:
```
# drop rows that have less than 4 real values
df.dropna(thresh=4)
```

Out[11]:

	A	B	C	D
0	1.0	2.0	3.0	4.0

In [12]:
```
df.dropna(thresh=3)
```

Out[12]:

	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

In [13]:
```
# only drop rows where NaN appear in specific columns (here: 'C')
df.dropna(subset=['C'])
```

Out[13]:

	A	B	C	D
0	1.0	2.0	3.0	4.0
2	10.0	11.0	12.0	NaN

In [14]:
```
df.dropna(subset=['A'])
```

Out[14]:

	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

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.

Imputing missing values

Use sklearn.impute.SimpleImputer

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:

In [15]:
```
import numpy as np
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values = np.nan, strategy = 'median')
imp = imp.fit(df.values)
imputed_data = imp.transform(df.values)
imputed_data
```

Out[15]:
```
array([[ 1. ,  2. ,  3. ,  4. ],
       [ 5. ,  6. ,  7.5,  8. ],
       [10. , 11. , 12. ,  6. ]])
```

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.

Use 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 (https://scikit-learn.org/dev/modules/generated/sklearn.metrics.pairwise.nan_euclidean_distances.html), 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:

In [16]:
```
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
```

Out[16]:

	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

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" :

In [17]:
```
# weights= 'uniform'  : uniform weights.
#All points in each neighborhood are weighted equally.
imputer = KNNImputer(n_neighbors=2, weights="uniform")
imputer.fit_transform(X) # you will get a np array
```

Out[17]:
```
array([[1. , 2. , 5. ],
       [3. , 4. , 3. ],
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

In [18]:
```
# weights= 'distance'  : weight points by the inverse of their distance.
# In this case, closer neighbors of a query point
# will have a greater influence than neighbors which are further away.
imputer = KNNImputer(n_neighbors=3, weights="distance")
imputer.fit_transform(X)
```

Out[18]:
```
array([[1.        , 2.        , 3.93905505],
       [3.        , 4.        , 3.        ],
       [5.5       , 6.        , 5.        ],
       [8.        , 8.        , 7.        ]])
```

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.

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:

In [19]:
```
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
```

Out[19]:

	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:

In [20]:
```
df2_train.mean(axis = 0)
```

Out[20]:
```
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.

In [21]:
```
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
```

Out[21]:
```
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:

In [22]:
```
 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
```

Out[22]:

	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:

In [23]:
```
df2_test.mean(axis = 0)
```

Out[23]:
```
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.m

In [24]:
```
imputed_df2_test = imp2.transform(df2_test)
imputed_df2_test
```

Out[24]:
```
array([[2., 4., 5., 7.],
       [3., 5., 4., 8.],
       [3., 4., 5., 8.]])
```

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.

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.

Creating an example dataset

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

In [25]:
```
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
```

Out[25]:

	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.

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:

In [26]:
```
# we define the mapping manually using the dictionary
size_mapping = {'XL': 3,
                'L': 2,
                'M': 1}
df['size'] = df['size'].map(size_mapping)
df
```

Out[26]:

	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:

In [27]:
```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
inv_size_mapping
```

Out[27]:
```
{3: 'XL', 2: 'L', 1: 'M'}
```

In [28]:
```
df['size'].map(inv_size_mapping)
```

Out[28]:
```
0     M
1     L
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.

In [ ]:
```
```