chapter-2-1

January 23, 2022

Run in Google Colab

1 Introduction to Machine Learning: Preprocessing

1.1 Definitions

Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. On the other hand, the success of a machine learning algorithm highly depends on the quality of the data fed into the model. Real-world data is often dirty containing outliers, missing values, wrong data types, irrelevant features, or non-standardized data. The presence of any of these will prevent the machine learning model to properly learn. For this reason, transforming raw data into a useful format is an essential stage in the machine learning process. Therefore, it is absolutely critical to ensure that we examine and preprocess a dataset before we feed it to a learning algorithm. In this section, we will discuss the essential data preprocessing techniques that will help us to build good machine learning models.

The topics that we will cover in this lesson are as follows:

- Removing and imputing missing values from the dataset
- Getting categorical data into shape for machine learning algorithms
- Selecting relevant features for the model construction
- Feature Normalization

1.2 Dealing with missing data

Let's create a simple example data frame from a comma-separated values (CSV) file to get a better grasp of the problem:

[2]:
```
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,'''
# If you are using Python 2.7, you need
# to convert the string to unicode:
# csv_data = unicode(csv_data)
```

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

[2]:

	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

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

[3]:
```
df.dropna(axis=0)
```

[3]:

	A	B	C	D
0	1.0	2.0	3.0	4.0

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 samples, which will make a 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 look at one of the most commonly used alternatives for dealing with missing values: interpolation techniques.

1.2.1 Imputing missing values

One of the most common interpolation 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:

[4]:
```
from sklearn.impute import SimpleImputer
import numpy as np

imr = SimpleImputer(missing_values=np.nan, strategy='mean')
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
imputed_data
```

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

Alternatively, an even more convenient way to impute missing values is by using pandas' fillna method and providing an imputation method as an argument. For example, using pandas, we could achieve the same mean imputation directly in the DataFrame object via the following command:

[5]:
```
df.fillna(df.mean())
```

[5]:
```
      A     B     C    D
0   1.0   2.0   3.0  4.0
1   5.0   6.0   7.5  8.0
2  10.0  11.0  12.0  6.0
```

1.3 Categorical Data

When we are talking about categorical data, we have to further distinguish between ordinal and nominal 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. Before we explore different techniques for handling such categorical data, let's create a new DataFrame to illustrate the problem:

[6]:
```
import pandas as pd

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

[6]:
```
   color size  price classlabel
0  green    M   10.1     class2
1    red    L   13.5     class1
2   blue   XL   15.3     class2
```

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:

[7]:
```
size_mapping = {'XL': 3,'L': 2,'M': 1}
df['size'] = df['size'].map(size_mapping)
df
```

[7]:
```
   color  size  price classlabel
0  green     1   10.1     class2
1    red     2   13.5     class1
2   blue     3   15.3     class2
```

1.3.1 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 en-

code 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:

[8]:
```
import numpy as np

class_mapping = {label: idx for idx, label in enumerate(np.
 ↪unique(df['classlabel']))}
class_mapping
```

[8]:
```
{'class1': 0, 'class2': 1}
```

Next, we can use the mapping dictionary to transform the class labels into integers:

[9]:
```
df['classlabel'] = df['classlabel'].map(class_mapping)
df
```

[9]:

	color	size	price	classlabel
0	green	1	10.1	1
1	red	2	13.5	0
2	blue	3	15.3	1

We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

[10]:
```
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
df
```

[10]:

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

Alternatively, there is a convenient LabelEncoder class directly implemented in scikit-learn to achieve this:

[11]:
```
from sklearn.preprocessing import LabelEncoder

class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
y
```

[11]:
```
array([1, 0, 1])
```

When there is no a natural order we have to resort to a different approach that is to use the 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 an example; for example, a blue example can be encoded as *blue=1*, *green=0*, *red=0*. To perform this transformation, we can use the OneHotEncoder that is implemented in scikit-learn's preprocessing module:

[12]:
```
from sklearn.preprocessing import OneHotEncoder

X = df[['color', 'size', 'price']].values
color_ohe = OneHotEncoder()
color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
```

[12]:
```
array([[0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

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:

[13]:
```
pd.get_dummies(df[['price', 'color', 'size']])
```

[13]:

	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

Inserire rimando ad esempio Ch. 4-0

1.4 Feature Normalization

Many machine learning algorithms require that the selected features are on the same scale for optimal performance, this process is called "Feature Normalization" and is the subject of this paragraph.

Data Normalization is a common practice in machine learning which consists of transforming numeric columns to a common scale. In machine learning, some feature values differ from others multiple times. The features with higher values will dominate the leaning process. However, it does not mean those variables are more important to predict the outcome of the model. Data normalization transforms multiscaled data to the same scale. After normalization, all variables have a similar influence on the model, improving the stability and performance of the learning algorithm.

There are multiple normalization techniques in statistics. In this notebook, we will cover the most important ones:

- The maximum absolute scaling
- The min-max feature scaling
- The z-score method

1.4.1 The maximum absolute scaling

The maximum absolute scaling rescales each feature between -1 and 1 by dividing every observation by its maximum absolute value.

$$x_{new} = \frac{x_{old}}{\max |x_{old}|}$$

1.4.2 The min-max feature scaling

The min-max approach (often called normalization) rescales the feature to a fixed range of [0,1] by subtracting the minimum value of the feature and then dividing by the range:

$$x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$$

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

[18]:
```
#
# Here we have to load the file 'salary_vs_age_1.csv'
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[19]:
```
# Load the Pandas libraries with alias 'pd'
import pandas as pd
# Read data from file 'salary_vs_age_1.csv'
# (in the same directory that your python process is based)
# Control delimiters, with read_table
df1 = pd.read_table(path + "salary_vs_age_1.csv", sep=";")
# Preview the first 5 lines of the loaded data
print(df1.head())

columns_titles = ["Salary","Age"]
df2=df1.reindex(columns=columns_titles)
df2

df2['Salary'] = df2['Salary']/1000
df2['Age2']=df2['Age']**2
df2['Age3']=df2['Age']**3
df2['Age4']=df2['Age']**4
df2['Age5']=df2['Age']**5
df2
```

```
   Age  Salary
0   25  135000
1   27  105000
2   30  105000
3   35  220000
4   40  300000
```

[19]:

	Salary	Age	Age2	Age3	Age4	Age5
0	135.0	25	625	15625	390625	9765625
1	105.0	27	729	19683	531441	14348907
2	105.0	30	900	27000	810000	24300000
3	220.0	35	1225	42875	1500625	52521875
4	300.0	40	1600	64000	2560000	102400000
5	270.0	45	2025	91125	4100625	184528125
6	265.0	50	2500	125000	6250000	312500000
7	260.0	55	3025	166375	9150625	503284375
8	240.0	60	3600	216000	12960000	777600000
9	265.0	65	4225	274625	17850625	1160290625

[20]:
```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
df3 = pd.DataFrame(mms.fit_transform(df2))
df3
```

[20]:

	0	1	2	3	4	5
0	0.153846	0.000	0.000000	0.000000	0.000000	0.000000
1	0.000000	0.050	0.028889	0.015668	0.008065	0.003984
2	0.000000	0.125	0.076389	0.043919	0.024019	0.012633
3	0.589744	0.250	0.166667	0.105212	0.063574	0.037162
4	1.000000	0.375	0.270833	0.186776	0.124248	0.080515
5	0.846154	0.500	0.388889	0.291506	0.212486	0.151898
6	0.820513	0.625	0.520833	0.422297	0.335588	0.263127
7	0.794872	0.750	0.666667	0.582046	0.501718	0.428951
8	0.692308	0.875	0.826389	0.773649	0.719895	0.667377
9	0.820513	1.000	1.000000	1.000000	1.000000	1.000000

1.4.3 Z-Score

The **z-score** method (often called **standardization**) transforms the data into a distribution with a mean of 0 and a standard deviation of 1. Each standardized value is computed by subtracting the mean of the corresponding feature and then dividing by the standard deviation.

$$x_{new} = \frac{x_{old} - \mu}{\sigma}$$

Unlike min-max scaling, the z-score does not rescale the feature to a fixed range. The z-score typically ranges from -3.00 to 3.00 (more than 99% of the data) if the input is normally distributed.

It is important to bear in mind that z-scores are not necessarily normally distributed. They just scale the data and follow the same distribution as the original input. This transformed distribution has a mean of 0 and a standard deviation of 1 and is going to be the standard normal distribution only if the input feature follows a normal distribution.

Standardization can easily be achieved by using the built-in NumPy methods mean and std:

[21]:
```
import numpy as np

X = np.array([6, 7, 7, 12, 13, 13, 15, 16, 19, 22])

X_std = np.copy(X)
X_std = (X - X.mean()) / X.std()

print(X_std)
```

```
[-1.39443338 -1.19522861 -1.19522861 -0.19920477  0.          0.
  0.39840954  0.5976143   1.19522861  1.79284291]
```

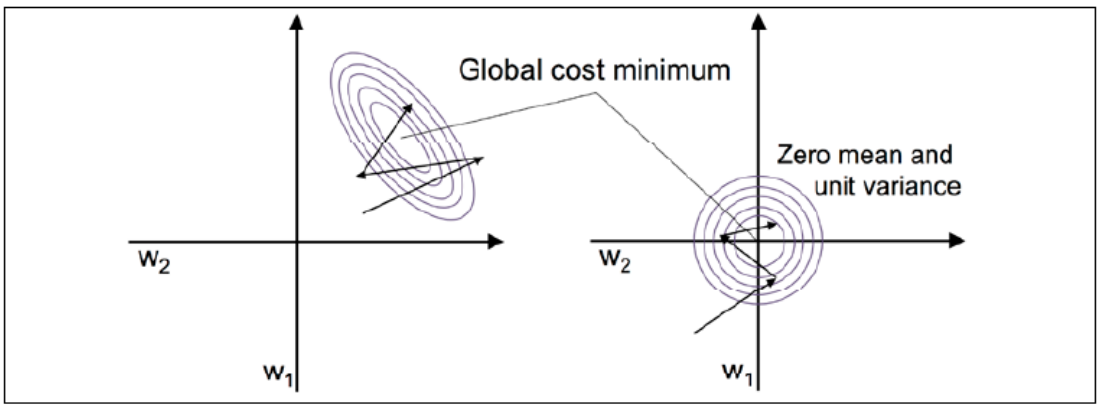
Or simply using the specific function of the stats module of scipy

[22]:
```
import scipy.stats as stats

stats.zscore(X)
```

[22]:
```
array([-1.39443338, -1.19522861, -1.19522861, -0.19920477,  0.        ,
        0.        ,  0.39840954,  0.5976143 ,  1.19522861,  1.79284291])
```

Standardization is very useful with gradient descent learning. In this case the optimizer has to go through fewer steps to find a good or optimal solution (the global cost minimum), as illustrated in the following figure, where the subfigures represent the cost surface as a function of two model weights in a two-dimensional classification problem:



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

[23]:
```
from sklearn.preprocessing import StandardScaler

stdsc = StandardScaler()
df4 = pd.DataFrame(stdsc.fit_transform(df2))
df4
```

[23]:

	0	1	2	3	4	5
0	-1.170242	-1.359724	-1.189131	-1.041783	-0.920815	-0.824435
1	-1.601005	-1.210304	-1.102065	-0.994071	-0.895974	-0.812022
2	-1.601005	-0.986174	-0.958907	-0.908042	-0.846835	-0.785069
3	0.050256	-0.612623	-0.686823	-0.721391	-0.725003	-0.708630
4	1.198959	-0.239072	-0.372880	-0.473014	-0.538122	-0.573535
5	0.768195	0.134478	-0.017078	-0.154092	-0.266345	-0.351091
6	0.696401	0.508029	0.380582	0.244194	0.112820	-0.004480
7	0.624608	0.881579	0.820102	0.730661	0.624511	0.512260
8	0.337432	1.255130	1.301481	1.314127	1.296512	1.255243
9	0.696401	1.628681	1.824719	2.003411	2.159252	2.291760