Politecnico di Torino

1859

Data Science and Machine Learning for Engineering Applications

Scikit-learn and Pandas

preprocessing

DataBase and Data Mining Group

Salvatore Greco
Andrea Pasini
Flavio Giobergia
Elena Baralis
Tania Cerquitelli

Data preprocessing

- Data preprocessing with Scikit-Learn and Pandas
- Summary
  - Missing values
  - Normalization
  - Feature extraction (examples)
    - Handling nominal data
    - Computing TF-IDF
  - Dimensionality reduction
    - PCA

Missing values

- Scikit-learn estimators are incompatible with **missing values** (e.g., NaN, blank, etc.)
- Scikit-learn estimators assume that all values in an array are **numerical**
- Handling missing values
  - **Discard** entire rows/columns containing missing values
  - **Impute** missing values (e.g., mean, median, constant, etc.)

Discard Missing values

- Create a DataFrame with some **missing values** (NaN values in this case)

```
In [1]:  data = np.array([[1, 2, 3, 4],
                          [5, 6, 7, np.nan],
                          [9, 10, np.nan, 11]])

         df = pd.DataFrame(data,columns=["A","B","C","D"])


         df
```

```
Out[1]:      A      B      C      D
         0   1.0    2.0    3.0    4.0
         1   5.0    6.0    7.0    NaN
         2   9.0    10.0   NaN    11.0
```

Discard Missing values

- You can create a **Mask** with **True** in the presence of **missing values** and False otherwise with *df.isnull()*

```
In [1]:  df.isnull()
```

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

Discard Missing values

- You can **count** the number of **missing values** in each column with the *.sum()* method

In [1]:

```
df.isnull().sum()
```

Out[1]:

```
A    0
B    0
C    1
D    1
dtype: int64
```

Discard Missing values

- You can **remove** rows or columns containing missing values with the *df.dropna(axis=)* method
  - Axis=0 to remove rows
  - Axis=1 to remove columns

```
In [1]:   df = df.dropna(axis=0) #Remove rows containing missing values
```

```
Out[1]:       A     B     C     D
          0   1.0   2.0   3.0   4.0
```

Note that all rows containing at least one column with a missing value are removed

```
In [1]:   df = df.dropna(axis=1) #Remove columns containing missing values
```

```
Out[1]:       A     B
          0   1.0   2.0
          1.  5.0   6.0
          2   9.0   10.0
```

Note that all columns containing at least one row with a missing value are removed

Discard Missing values

- You can **remove** rows or columns containing missing values with the *df.dropna(axis=)* method
  - Axis=0 to remove rows
  - Axis=1 to remove columns

```
In [1]:  df = df.dropna(axis=0) #Remove rows containing missing values
```

```
Out[1]:
        A     B     C     D
   0   1.0   2.0   3.0   4.0
```

Note that all rows containing at least one column with a missing value are removed

```
In [1]:  df = df.dropna(axis=1) #Remove columns containing missing values
```

```
Out[1]:
        A     B
   0   1.0   2.0
   1.  5.0   6.0
   2   9.0   10.0
```

Note that all columns containing at least one row with a missing value are removed

Discard Missing values

- You can **remove** rows containing missing values in a **specific column** specifying the subset parameter

```
In [1]:  df = df.dropna(subset = ["D"])        #Remove rows containing missing values in the ‘D’ column
```

```
Out[1]:
      A     B      C      D
0   1.0   2.0    3.0.   4.0
2.  9.0   10.0   NaN   11.0
```

Note that all rows containing a missing value in the ‘D’ column are removed

- You can also **select** all the rows with **not-null values** in a specific column

```
In [1]:  df = df[df['D'].notnull()]
```

Impute Missing values

- Scikit-Learn provides some built-in functions to **impute** values on missing data
  - ***SimpleImputer*** can **replace missing values** using a **descriptive statistic** (e.g., *mean*, *median*, or *most frequent*) along each column, or using a *constant* value.

In [1]:

```
from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean’)
imp_mean.fit(X_train.values) # Fit only on training data!
X_train = imp_mean.transform(X_train.values)
X_test = imp_mean.transform(X_ test.values)
X_train
```

Out[1]:

```
array([[ 1. , 2. , 3. , 4. ],
       [ 5. , 6. , 7. , 7.5],
       [ 9. , 10. , 5. , 11. ]])
```

Note that is returned a Numpy array. If you want, you can create a new DataFrame from it

Impute Missing values

- You can specify how to **impute** the value to replace missing values specifying the *strategy* parameter
  - *”mean”*: replace missing values using the **mean along each column**. It can only be used with numeric data.
  - *“median”:* replace missing values using the median along each column. It can only be used with numeric data.
  - *“most_frequent”:* replace missing using **the most frequent value along each column**. It can be used with strings or numeric data. If there is more than one such value, only the smallest is returned.
  - *“constant”*: replace missing values with *fill_value* parameter. It can be used with strings or numeric data.
    - For string or object data types, fill_value must be a string. If None, fill_value will be 0 when imputing numerical data and “missing_value” for strings or object data types.

Impute Missing values

- You can **impute** values on missing data with Pandas with the *df.fillna()* method
- You should specify the value that will replace the *NaN* values
  - It can be a constant, or a statistic computed with pandas (e.g., *np.mean()*, *np.median()*, etc.)

In [1]:

```
df.fillna(df.mean()) # Fill NaN values with column mean
```

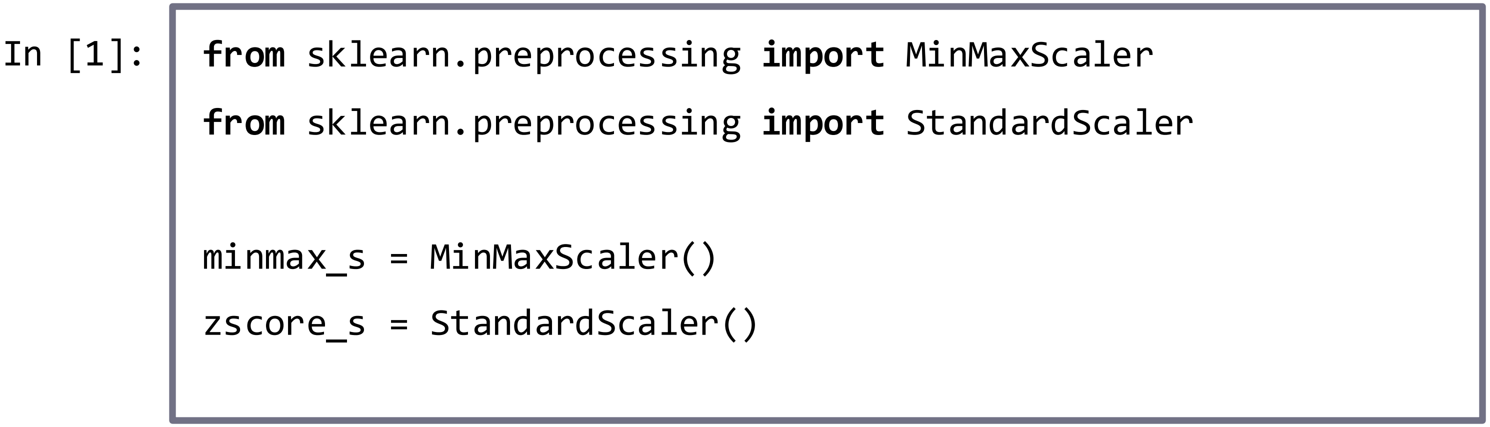
Out[1]:

```
     A     B      C      D
0   1.0   2.0    3.0    4.0
1   5.0   6.0    7.0    7.5
2   9.0   10.0   5.0.   11.0
```

Note that, in this case, it is returned a DataFrame

Normalization

- Examples:
  - min-max normalization: MinMaxScaler
  - z-score normalization: StandardScaler



```
In [1]:  from sklearn.preprocessing import MinMaxScaler
         from sklearn.preprocessing import StandardScaler


         minmax_s = MinMaxScaler()
         zscore_s = StandardScaler()
```

Normalization

- Applying normalization to training and test set

In [1]:

```
X_train = [[0, 10], [0, 20], [2, 10], [2, 20]]
X_test = [[1, 15]]

minmax_s.fit(X_train) # NOTE: “learning” on training data only!
X_train_norm = minmax_s.transform(X_train)
X_test_norm = minmax_s.transform(X_test) # correct
X_test_wrong = minmax_s.fit_transform(X_test) # do not fit on test
print(X_test_norm)
print(X_test_wrong)
```

Out[1]:

```
[[0.5 0.5]]
[[0, 0]]
```

ALWAYS REMEMBER

1. **Fit** the transformers only **on the training set**
2. **Transform** the training set (use fit_transform() if convenient)
3. **Only transform** the test set — **never fit on it!**

Feature extraction

- Necessary when a datasets presents samples that:
  - Are **not numerical vectors**
    - Example: nominal data, text, images
  - The **model** has a low capacity/can't extract enough knowledge from the row features
    - Example: extraction of polynomial features
    - Example: extraction of season from dates

Feature extraction

- Nominal data
  - Two nominal values can only be compared with the equality operator (**cannot be ordered**)
  - For this reason it is **incorrect** to map them to integer features:
    - E.g. ‘red’, ‘green’, ‘blue’ → [0, 1, 2]
    - Colors have no ordering
    - The model could infer ordering properties that do not describe correctly our data

Feature extraction

- Nominal data
  - One of the simplest solutions is to use **one-hot encoding**:
    - `Red   → 0, 0, 1`
    - `Green → 0, 1, 0`
    - `Blue  → 1, 0, 0`
  - Pay attention: the **size of the output vector** is linear with the number of distinct values for the attribute
    - Some models (e.g. KNN, clustering) may have problems while working with high dimensional data

Remind: **Curse of dimensionality:**
The curse of dimensionality refers to the exponential increase in data sparsity and computational complexity as the number of features (dimensions) grows, making learning and generalization harder.

Feature extraction

- Nominal data: 1-Hot vectors from dictionaries

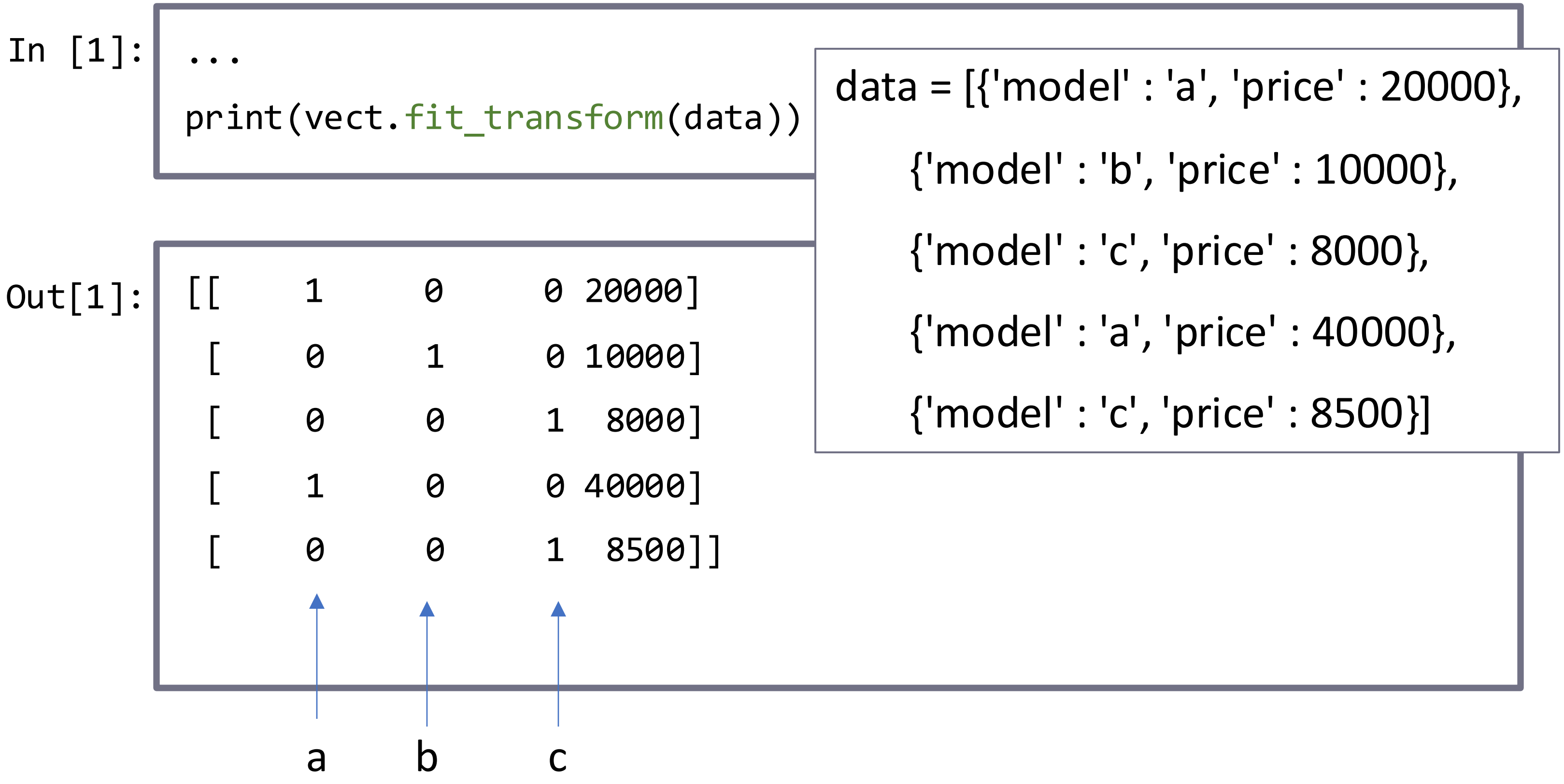
```
In [1]:   from sklearn.feature_extraction import DictVectorizer
          vect = DictVectorizer(sparse=False, dtype=int)

          data = [{'model' : 'a', 'price' : 20000},
Out[1]:          {'model' : 'b', 'price' : 10000},
                 {'model' : 'c', 'price' : 8000},
                 {'model' : 'a', 'price' : 40000},
                 {'model' : 'c', 'price' : 8500}]

          print(vect.fit_transform(data))
```

Feature extraction

- Nominal data: 1-Hot vectors from dictionaries



```
In [1]: ...
        print(vect.fit_transform(data))
```

```
data = [{'model' : 'a', 'price' : 20000},
        {'model' : 'b', 'price' : 10000},
        {'model' : 'c', 'price' : 8000},
        {'model' : 'a', 'price' : 40000},
        {'model' : 'c', 'price' : 8500}]
```

```
Out[1]: [[    1     0     0 20000]
         [    0     1     0 10000]
         [    0     0     1  8000]
         [    1     0     0 40000]
         [    0     0     1  8500]]
```

Feature extraction

- Nominal data: 1-Hot vectors from dictionaries
  - If you have training and test data use fit and transform separately:

```
In [1]:  train = data[:3]
         test = data[3:]

         vect = DictVectorizer(sparse=False, dtype=int)
         vect.fit(train) # Learn vocabulary from training set
         test_transformed = vect.transform(test)
```

Feature extraction

- 1-Hot encoding with *OneHotEncoder*
  - Allows passing data in tabular form (“feature matrix”)
  - Numerical values are also encoded – **beware!**
    - Some matrix manipulation is required to only encode categorical features

In [1]:

```
from sklearn.preprocessing import OneHotEncoder

X = np.array([
    ["a", 20000],
    ["b", 10000],
    ["c", 8000],
    ["a", 40000],
    ["c", 8500]
])

ohe = OneHotEncoder(sparse=False, dtype=int)
X_ohe = ohe.fit_transform(X[:,:1])

# same result as DictVectorizer (from before)
X_enc = np.hstack([ X_ohe, X[:,1:].astype(int)])
```

Feature extraction

- Textual data
  - Convert textual documents to count vectors
    - 1 feature for each word of the vocabulary that count the number of occurrences in the document
    - Scikit-learn transformer: *CountVectorizer*
    - Example:
      - “My cat. My dog. My cat.”
      - “My dog. My house.”

cat	dog	house	my
2	1	0	3
0	1	1	1

Feature extraction

- Textual data
  - Convert textual documents to count vectors
    - Drawback： frequent words have high scores for almost all documents
  - Solution：**TF-IDF**（Term Freq. Inverse Document Freq.）
    - Penalizes words that are common in all documents
    - Boosts words that are frequent in a document, but not in the others

A look into TF-IDF

- **Term Frequency** (TF):
  Measures how often a term t appears in a document d.
  Inverse Document Frequency (IDF):

$$\mathrm{TF(t,d)} = \frac{\#\, times\ term\ t\ appears\ in\ document\ d}{\#\, total\ terms\ in\ d}$$

- **Inverse Document Frequency (IDF) :**
  Reduces weight for terms common across many documents.

$$IDF(t) = \log\left(\frac{N}{1 + \#\, Document\ containing\ t}\right)$$

- TF-IDF = TF × IDF
  High TF-IDF ⟶ word is frequent in one doc but rare in corpus

Feature extraction

- Textual data: TF-IDF

In [1]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vect = TfidfVectorizer(stop_words="english")


data = ["dog bites cat", "cat bites dog", "cat and dog house"]
print(vect.fit_transform(data).toarray())
```

convert to Numpy array

Out[1]:

```
[[0.67325467 0.52284231 0.52284231 0.        ]
 [0.67325467 0.52284231 0.52284231 0.        ]
 [0.         0.45329466 0.45329466 0.76749457]]
```

Feature extraction

- Textual data: TF-IDF

```
In [1]:   ...
          data = ["dog bites cat", "cat bites dog", "cat and dog house"]
          print(vect.fit_transform(data).toarray())
          # Print the learned vocabulary
          print(vect.vocabulary_)
```

```
             bites        cat         dog         house
Out[1]:   [[0.67325467 0.52284231 0.52284231 0.        ]   Doc 1
           [0.67325467 0.52284231 0.52284231 0.        ]   Doc 2
           [0.         0.45329466 0.45329466 0.76749457]]  Doc 3

          {'dog': 2, 'bites': 0, 'cat': 1, 'house': 3}
```

stopword “and” has been removed

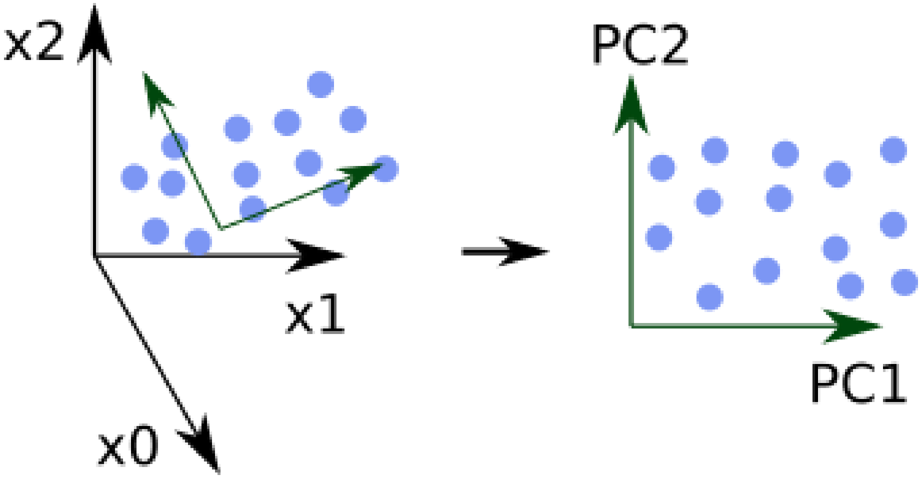
specific of this document

Dimensionality reduction

- Useful when you want to reduce the number of features for high-dimensional data
  - For **graphical** representations
  - Before applying **classification** and **clustering** to give the features matrix a more **compact** representation

Dimensionality reduction

- Example: **PCA**
  - Reduces the dimensionality by finding the directions in the space where data has more variance
  - *If you're into linear algebra*: PCA is just a rotation along the covariance matrix's eigenvectors!



Dimensionality reduction

PCA with Scikit-learn

```
from sklearn.decomposition import PCA


pca = PCA(n_components=5)
X_projection = pca.fit_transform(X)
```

- **n_components** specify the number of components that you want to keep after applying PCA
  - Should be <= the number of initial features
- The result is a features matrix with the specified number of features

ALWAYS REMEMBER

- **Normalize Before PCA!**

PCA is sensitive to the **scale of the features**.

Example: think to a screw:

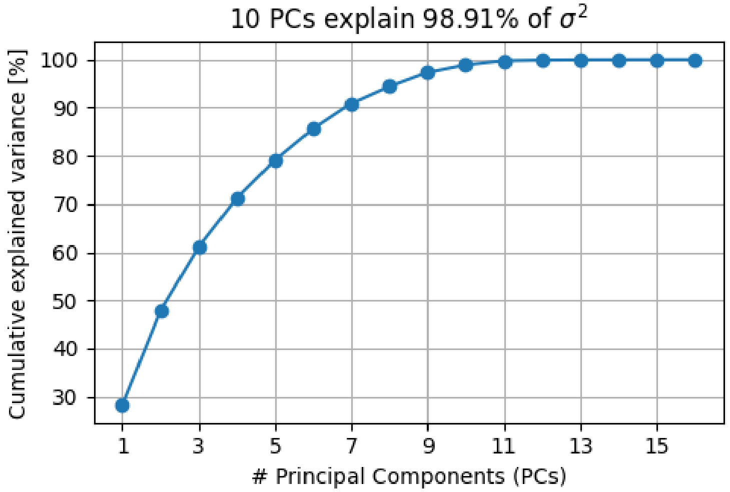
- The **length** is much greater than the **width**
- Without normalization, PCA will align mostly with the length!

- **Fix:** standardize your data (zero mean, unit variance) **before** applying PCA!

Dimensionality reduction

- Choosing the correct number of components – when you do not have too much features: **Cumulative explained variance ≥ th.**

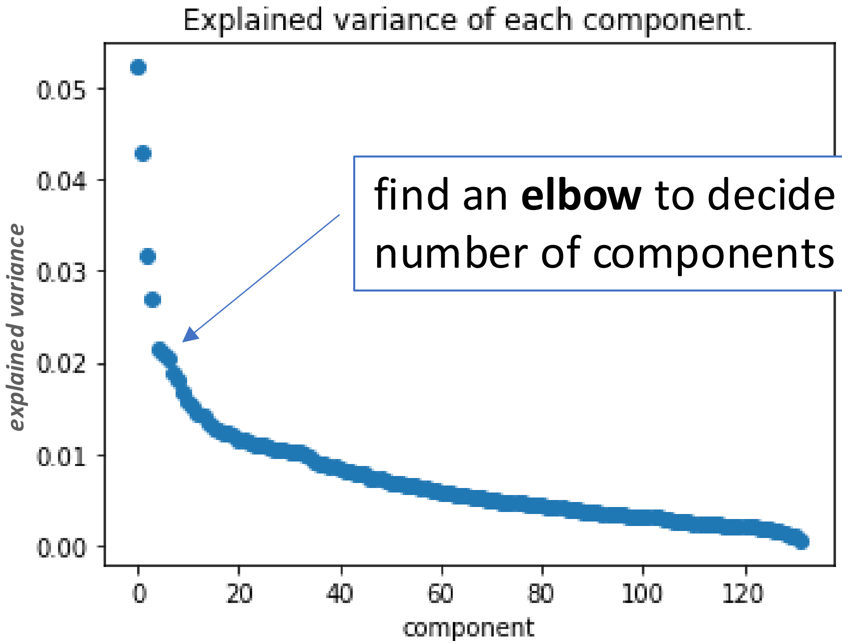
```
pca = PCA() # compute all the components
th = 0.9
X_projection = pca.fit_transform(X)
cumul_exp_var = np.cumsum(explained_variance)
n_components = np.where(cum_var >= threshold)[0][0]
```



Dimensionality reduction (when too much features)

- Choosing the correct number of components

```
pca = PCA(n_components=130) #compute the top-N components with largest variance
X_projection = pca.fit_transform(X)
plt.plot(pca.explained_variance_ratio_, marker='o', linestyle='')
```



find an **elbow** to decide the number of components

Dimensionality reduction

- Applying the transformation and a classifier

```
pca = PCA(n_components=6)
X_projection = pca.fit_transform(X_train)
my_classifier.train(X_projection, y_train)

# PCA is already fit on training data: do not fit it on test set!
X_test_proj = pca.transform(X_test)
y_test_pred = my_classifier.predict(X_test_proj)
```

Scikit-learn documentation

- **Other preprocessing methods**
  - https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing