Data Science Lab

Pre-processing in Scikit-learn

Andrea Pasini
Flavio Giobergia

DataBase and Data Mining Group

Data preprocessing

- **Normalization**
  - Standard Scaling, Min-Max scaling
- **Feature extraction**
  - Nominal and ordinal data
  - TF-IDF
- **Dimensionality reduction**
  - PCA
- **Missing values imputation**
- **Mixed pipelines**

Normalization

- min-max normalization: MinMaxScaler
  - Scales data to be in the [0,1] range
- z-score normalization: StandardScaler
  - Scales data to have 0 mean and variance of 1
- Operations performed column-wise

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]]

# NOTE: “learning” min & max values from training data only!
minmax_s.fit(X_train)
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]]
```

Feature extraction

PoliTo DBMG

- 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

PoliTo DBMG

- 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** grows with the number of distinct values for the attribute
    - Some models (e.g. KNN, clustering) may have problems while working with high dimensional data

Feature extraction

- 1-Hot encoding with `OneHotEncoder`
  - Allows passing data in tabular form ("feature matrix")
  - Numerical values are also encoded – **beware!**
    - Some manipulation required to only encode categorical features
    - The ColumnTransformer class can be used to specify different transformations for different columns
      - (more on this later!)

```
from sklearn.preprocessing import OneHotEncoder

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

ohe = OneHotEncoder(sparse_output=False, dtype=int)

X_cat = X[:, :1]
X_num = X[:, 1:].astype(int)
X_ohe = ohe.fit_transform(X_cat)

X_enc = np.hstack([ X_ohe, X_num])
```

Feature extraction

- In ordinal data, there is an established order among elements
  - E.g., XS < S < M < L < XL
- To preserve this information, we can map ordinal attributes to integers (`OrdinalEncoder`)

```
from sklearn.preprocessing import
OrdinalEncoder

X = np.array([
  ["XL", "v1"],
  ["M", "v2"],
  ["L", "v3"],
  ["S", "v1"],
  ["S", "v2"],
  ["XS", "v3"]
])


oe = OrdinalEncoder()
X_enc = oe.fit_transform(X)
print(X_enc)
```

```
[[3. 0.]
 [1. 1.]
 [0. 2.]
 [2. 0.]
 [2. 1.]
 [4. 2.]]
```

Unless otherwise specified, lexicographic order is used, so:
col1 : L<M<S<XL<XS [wrong!]
col2: v1<v2<v3 [ok :D]

```
from sklearn.preprocessing import
OrdinalEncoder

X = np.array([
  ["XL", "v1"],
  ["M", "v2"],
  ["L", "v3"],
  ["S", "v1"],
  ["S", "v2"],
  ["XS", "v3"]
])

oe = OrdinalEncoder(categories=[
   ["XS", "S", "M", "L", "XL"],
   ["v1", "v2", "v3"]
])
X_enc = oe.fit_transform(X)
print(X_enc)
```

```
[[4. 0.]
 [2. 1.]
 [3. 2.]
 [1. 0.]
 [1. 1.]
 [0. 2.]]
```

Column-wise transformation

- Different columns may have different data type
  - thus requiring different pre-processing
- E.g., numerical data + nominal data + ordinal data
- ColumnTransformer can be used to specify, for each column, what process should be applied

```
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.compose import ColumnTransformer


X = np.array([
  [5.1, 3.5, 1.4, "A", "Low", "Type1"],
  [4.9, 3.0, 1.4, "B", "Med", "Type2"],
  [4.7, 3.2, 1.3, "A", "High", "Type1"],
  [4.6, 3.1, 1.5, "C", "Med", "Type2"],
  [5.0, 3.6, 1.4, "B", "Low", "Type1"]
])
```

ColumnTransformer

```
X = np.array([
[5.1, 3.5, 1.4, "A", "Low", "Type1"],
[4.9, 3.0, 1.4, "B", ”Med", "Type2"],
[4.7, 3.2, 1.3, "A", "High", "Type1"],
[4.6, 3.1, 1.5, "C", "Med", "Type2"],
[5.0, 3.6, 1.4, "B", "Low", "Type1"]
])


preprocessor = ColumnTransformer(transformers=[
  ('num', StandardScaler(), [0, 1]),
  ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), [3]),
  ('ord', OrdinalEncoder(categories=[["Low", ”Med", "High"], ["Type1", "Type2"]]), [4, 5])
], remainder='passthrough')
```

Columns that are not explicitly mentioned either get dropped (`remainder="drop"`), or left as-is (`remainder="passthrough"`)

Feature extraction

PoliTo DBMG

- 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	2

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

Feature extraction

PoliTo DBMG

- 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.get_feature_names_out())
```

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

array(['bites', 'cat', 'dog', 'house'])
```

bites cat dog house

stopword “and” has been removed

specific of this document

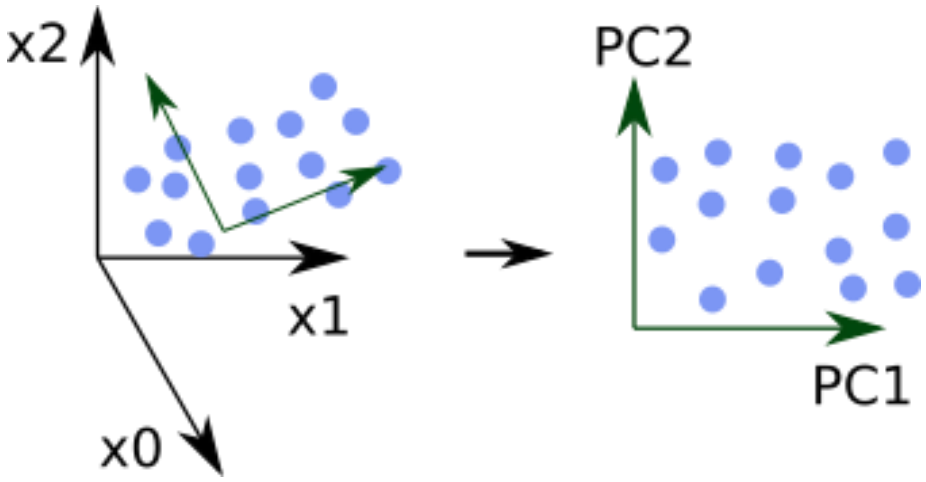
Dimensionality reduction

PoliTo DBMG

- 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



Dimensionality reduction

- PCA with Scikit-learn

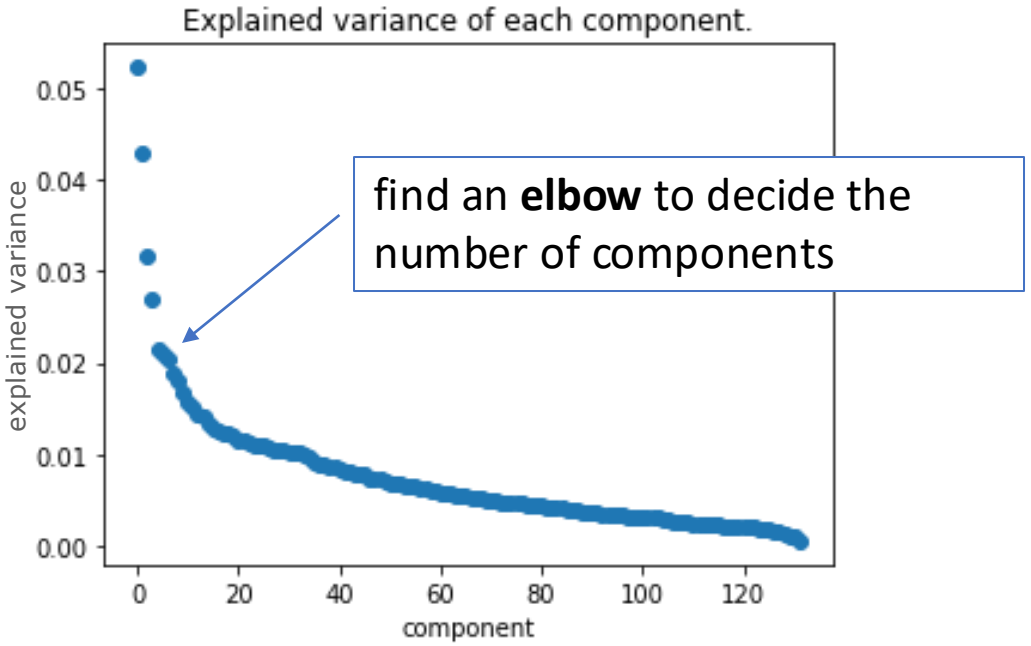
```
from sklearn.decomposition import PCA


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

Dimensionality reduction

- Choosing the correct number of components

```
pca = PCA(n_components=130)
X_projection = pca.fit_transform(X)
plt.plot(pca.explained_variance_ratio_, marker='o', linestyle='')
```



Dimensionality reduction

PoliTo DBMG

- 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)
```

Missing values imputation

PoliTo DBMG

- We can fill missing values based on:
  - Univariate approaches
    - Using naive strategies (e.g. “0”, mean value)
  - Multivariate approaches
    - Infer values from known data
- Both approaches are provided in scikit-learn

Univariate imputation

PoliTo DBMG

`sklearn.impute.SimpleImputer`

- Imputation based on:
  - Constant values
    - `strategy="constant", fill_value=value`
  - Statistics
    - Mean, median, most frequent
    - `strategy="mean"|"median"|"most_frequent"`

Multivariate imputation

e.g., `sklearn.impute.KNNImputer`

- Indentify k nearest neighbors
  - Distance defined as distance over non-missing features
  - By default, NaN-aware Euclidean distance used, defined as:
    - $d^2(a,b) = w(a,b) \sum_i^n \mathbf{1}(a_i \neq \perp \wedge b_i \neq \perp)(a_i - b_i)^2$
    - $w(a,b) = \frac{n}{\sum \mathbf{1}(a_i \neq \perp \wedge b_i \neq \perp)}$
    - i.e., distance over non-missing dimensions
    - weighted by w
      - (larger w = more missing dimensions, the ones available are weighted more)
- Impute value based on mean over neighbors

Distance example

- a = [1, NaN, 2, NaN]
- b = [5, 3, 2, NaN]

$$d^2(a,b) = w(a,b) \sum_i^n \mathbf{1}(a_i \neq \perp \wedge b_i \neq \perp)(a_i - b_i)^2$$

$$w(a,b) = \frac{n}{\sum \mathbf{1}(a_i \neq \perp \wedge b_i \neq \perp)}$$

- n = 4
- $w(a,b) = \frac{n}{\sum \mathbf{1}(a_i \neq \perp \wedge b_i \neq \perp)}$ = 4/2 = 2
  - (2 cols with non-missing values for a,b)
- $\sum_i^n \mathbf{1}(a_i \neq \perp \wedge b_i \neq \perp)(a_i - b_i)^2$
  - 1*(1-5)² + 0*(?)² + 1* (2-2)² + 0*(?)² = 16
- d²(a,b) = 2 * 16 = 32

Scikit-learn documentation

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