Intro5_DataNormalization_professor

September 15, 2020

1 Data preprocessing methods: Normalization

```
Notebook version:

* 1.0 (Sep 15, 2020) - First version

Authors: Jesús Cid Sueiro (jcid@ing.uc3m.es)
```

1.1 1. Data preprocessing

The aim of data preprocessing methods is to transform the data into a form that is ready to apply machine learning algorithms. This my include:

-
- Data imputation: assign values to features that may be missed for some data samples
- Feature extraction: transform the original data to compute new features that are more appropiate for a specific prediction task
- Dimensionality reduction: remove features that are not relevant for the prediction task.
- Outlier removal: remove samples that may contain errors and are not reliable for the prediction task.

In this notebook we will focus on data normalization.

[6]:
```
# Let's import some libraries
import numpy as np
import matplotlib.pyplot as plt
```

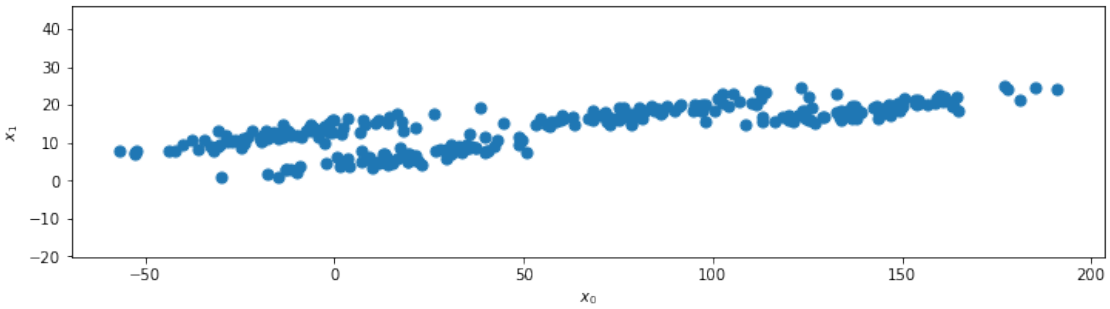
1.2 2. Data normalization

Consider a dataset $\mathcal{S} = \{\mathbf{x}_k, k = 0, \ldots, K-1\}$ of m-dimensional samples arranged by rows in a $K \times m$ data matrix, $\mathbf{X}$. Each one of the m data features may represent variables of very different nature (e.g. time, distance, price, volume, pixel intensity). Thus, the scale and the range of variation of each feature can be completely different.

As an illustration, consider the 2-dimensional dataset in the figure

[8]:
```
from sklearn.datasets.samples_generator import make_blobs
X, y = make_blobs(n_samples=300, centers=4, random_state=0, cluster_std=0.60)
X = X @ np.array([[30, 4], [-8, 1]]) + np.array([90, 10])

plt.figure(figsize=(12, 3))
plt.scatter(X[:, 0], X[:, 1], s=50);
plt.axis('equal')
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.show()
```



We can see that the first data feature (x_0) has a much large range of variation than the second. In practice, this may be problematic: the convergence properties of some machine learning algorithms may depend critically on the feature distributions and, in general, features sets ranging over similar scales use to offer a better performance.

For this reason, transforming the data in order to get similar range of variations for all features is desirable. This can be done in several ways.

1.2.1 2.1. Standard scaling.

A common normalization method consists on applying an affine transformation

$$\mathbf{t}_k = \mathbf{D}(\mathbf{x} - \mathbf{m})$$

where $\mathbf{D}$ is a diagonal matrix, in such a way that the transformed dataset $\mathcal{S}' = \{\mathbf{t}_k, k = 0, \ldots, K-1\}$ has zero sample mean, i.e.,

$$\frac{1}{K}\sum_{k=0}^{K-1} \mathbf{t}_k = 0$$

and zero sample variance, i.e.,

$$\frac{1}{K}\sum_{k=0}^{K-1} t_{ki}^2 = 1$$

It is not difficult to verify that this can be done by taking $\mathbf{m}$ equal to the sample mean

$$\mathbf{m} = \frac{1}{K}\sum_{k=0}^{K-1} \mathbf{x}_k$$

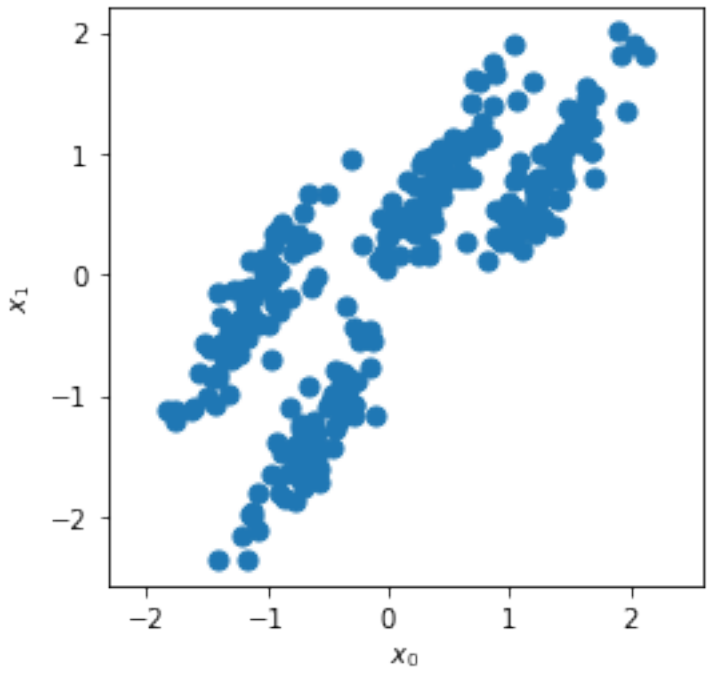
and the diagonal components of $\mathbf{D}$ equal to the inverse of the standard deviation of each feature, i.e.,

$$d_{ii} = \frac{1}{\frac{1}{K}\sum_{k=0}^{K-1}(x_{ki} - m_i)^2}$$

Using the data matrix $\mathbf{X}$ and the *broadcasting* property of the basic mathematical operators, the implementation of this normalization in Python is straightforward:

[7]:
```
m = np.mean(X, axis=0)      # Compute the sample mean
d = np.std(X, axis=0)       # Compute the standard deviation of each feature
T = (X-m)/d                 # Normalize

plt.figure(figsize=(4, 4))
plt.scatter(T[:, 0], T[:, 1], s=50);
plt.axis('equal')
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.show()
```



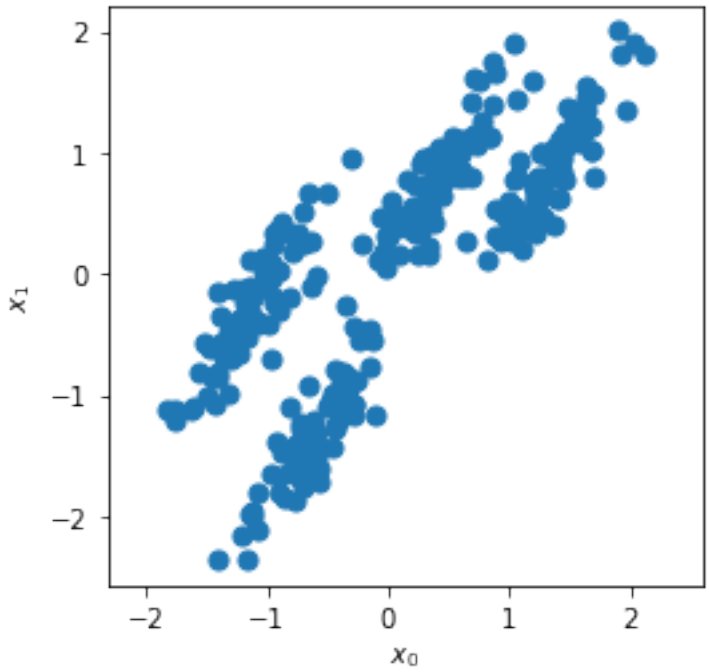
1.2.2 2.1.1. Implementation in `sklearn`

The `sklearn` package containing a method to perform the standard scaling over a given data matrix.

[5]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X)
print(f'The sample mean is m = {scaler.mean_}')

T2 = scaler.transform(X)
plt.figure(figsize=(4, 4))
plt.scatter(T2[:, 0], T2[:, 1], s=50);
plt.axis('equal')
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.show()
```

```
The sample mean is m = [58.06760405 13.94251037]
```



Note that, once we have defined the sclaler object in python, you can apply the scaling transformation to other datasets. This will be usefull in further topics, when the dataset may be split in

several matrices. We may be interested in defining the transformation using some data, and apply it to others

1.2.3 2.2. Other normalizations.

The are some alternatives to the standard scaling that may be interesting for some datasets. Here we show some of them, available at the preprocessing module in `sklearn`:

- preprocessing.MaxAbsScaler: Scale each feature by its maximum absolute value. As a result, all feature values will lie in the interval [-1, 1].
- preprocessing.MinMaxScaler: Transform features by scaling each feature to a given range. Also, all feature values will lie in the interval [-1, 1].
- preprocessing.Normalizer: Normalize samples individually to unit norm. That is, it applies the transformation $\mathbf{t}_k = \frac{1}{\|\mathbf{x}_k\|}\mathbf{x}_k$
- preprocessing.PowerTransformer: Apply a power transform featurewise to make data more Gaussian-like.
- preprocessing.QuantileTransformer: Transform features using quantile information. The transformed features follow a specific target distribution (uniform or normal).
- preprocessing.RobustScaler: Scale features using statistics that are robust to outliers. This way, anomalous values in one or very few samples cannot have a strong influence in the normalization.

You can find more detailed explanation of these transformations `sklearn` documentation.