3
Feature Selection and Feature Engineering

Feature engineering is the first step in a machine learning pipeline and involves all the techniques adopted to clean existing datasets, increase their signal-noise ratio, and reduce their dimensionality. Most algorithms have strong assumptions about the input data, and their performances can be negatively affected when raw datasets are used. Moreover, the data is seldom isotropic; there are often features that determine the general behavior of a sample, while others that are correlated don't provide any additional pieces of information. So, it's important to have a clear view of a dataset and know the most common algorithms used to reduce the number of features or select only the best ones.

scikit-learn toy datasets

scikit-learn provides some built-in datasets that can be used for testing purposes. They're all available in the package `sklearn.datasets` and have a common structure: the data instance variable contains the whole input set `X` while target contains the labels for classification or target values for regression. For example, considering the Boston house pricing dataset (used for regression), we have:

```
from sklearn.datasets import load_boston

>>> boston = load_boston()
>>> X = boston.data
>>> Y = boston.target

>>> X.shape
(506, 13)
>>> Y.shape
```

```
(506,)
```

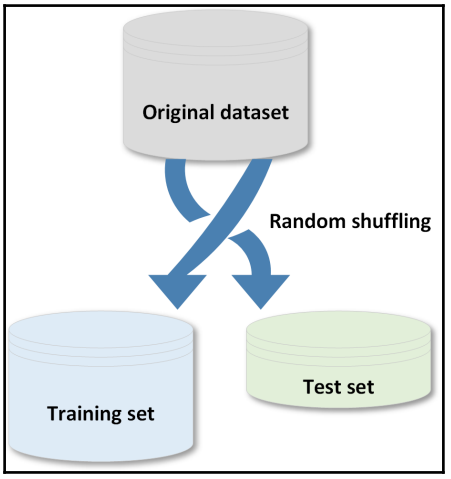
In this case, we have 506 samples with 13 features and a single target value. In this book, we're going to use it for regressions and the MNIST handwritten digit dataset (`load_digits()`) for classification tasks. scikit-learn also provides functions for creating dummy datasets from scratch: `make_classification()`, `make_regression()`, and `make_blobs()` (particularly useful for testing cluster algorithms). They're very easy to use and in many cases, it's the best choice to test a model without loading more complex datasets.

Visit `http://scikit-learn.org/stable/datasets/` for further information.

The MNIST dataset provided by scikit-learn is limited for obvious reasons. If you want to experiment with the original one, refer to the website managed by Y. LeCun, C. Cortes, C. Burges: `http://yann.lecun.com/exdb/mnist/`. Here you can download a full version made up of 70,000 handwritten digits already split into training and test sets.

Creating training and test sets

When a dataset is large enough, it's a good practice to split it into training and test sets; the former to be used for training the model and the latter to test its performances. In the following figure, there's a schematic representation of this process:



There are two main rules in performing such an operation:

- Both datasets must reflect the original distribution
- The original dataset must be randomly shuffled before the split phase in order to avoid a correlation between consequent elements

With scikit-learn, this can be achieved using the `train_test_split()` function:

```
from sklearn.model_selection import train_test_split

>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.25, random_state=1000)
```

The parameter `test_size` (as well as `training_size`) allows specifying the percentage of elements to put into the test/training set. In this case, the ratio is 75 percent for training and 25 percent for the test phase. Another important parameter is `random_state` which can accept a NumPy `RandomState` generator or an integer seed. In many cases, it's important to provide reproducibility for the experiments, so it's also necessary to avoid using different seeds and, consequently, different random splits:

My suggestion is to always use the same number (it can also be 0 or completely omitted), or define a global `RandomState` which can be passed to all requiring functions.

```
from sklearn.utils import check_random_state

>>> rs = check_random_state(1000)
<mtrand.RandomState at 0x12214708>

>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.25, random_state=rs)
```

In this way, if the seed is kept equal, all experiments have to lead to the same results and can be easily reproduced in different environments by other scientists.

For further information about NumPy random number generation, visit `https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.RandomState.html`.

Managing categorical data

In many classification problems, the target dataset is made up of categorical labels which cannot immediately be processed by any algorithm. An encoding is needed and scikit-learn offers at least two valid options. Let's consider a very small dataset made of 10 categorical samples with two features each:

```
import numpy as np

>>> X = np.random.uniform(0.0, 1.0, size=(10, 2))
>>> Y = np.random.choice(('Male','Female'), size=(10))
>>> X[0]
array([ 0.8236887 ,  0.11975305])
>>> Y[0]
'Male'
```

The first option is to use the `LabelEncoder` class, which adopts a dictionary-oriented approach, associating to each category label a progressive integer number, that is an index of an instance array called `classes_`:

```
from sklearn.preprocessing import LabelEncoder

>>> le = LabelEncoder()
>>> yt = le.fit_transform(Y)
>>> print(yt)
[0 0 0 1 0 1 1 0 0 1]

>>> le.classes_array(['Female', 'Male'], dtype='|S6')
```

The inverse transformation can be obtained in this simple way:

```
>>> output = [1, 0, 1, 1, 0, 0]
>>> decoded_output = [le.classes_[i] for i in output]
['Male', 'Female', 'Male', 'Male', 'Female', 'Female']
```

This approach is simple and works well in many cases, but it has a drawback: all labels are turned into sequential numbers. A classifier which works with real values will then consider similar numbers according to their distance, without any concern for semantics. For this reason, it's often preferable to use so-called **one-hot encoding**, which binarizes the data. For labels, it can be achieved using the `LabelBinarizer` class:

```
from sklearn.preprocessing import LabelBinarizer

>>> lb = LabelBinarizer()
>>> Yb = lb.fit_transform(Y)
array([[1],
```

```
        [0],
        [1],
        [1],
        [1],
        [1],
        [0],
        [1],
        [1],
        [1]])

>>> lb.inverse_transform(Yb)
array(['Male', 'Female', 'Male', 'Male', 'Male', 'Male', 'Female', 'Male',
       'Male', 'Male'], dtype='|S6')
```

In this case, each categorical label is first turned into a positive integer and then transformed into a vector where only one feature is 1 while all the others are 0. It means, for example, that using a softmax distribution with a peak corresponding to the main class can be easily turned into a discrete vector where the only non-null element corresponds to the right class. For example:

```
import numpy as np

>>> Y = lb.fit_transform(Y)
array([[0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0],
       [1, 0, 0, 0, 0]])

>>> Yp = model.predict(X[0])
array([[0.002, 0.991, 0.001, 0.005, 0.001]])

>>> Ypr = np.round(Yp)
array([[ 0.,  1.,  0.,  0.,  0.]])

>>> lb.inverse_transform(Ypr)
array(['Female'], dtype='|S6')
```

Another approach to categorical features can be adopted when they're structured like a list of dictionaries (not necessarily dense, they can have values only for a few features). For example:

```
data = [
   { 'feature_1': 10.0, 'feature_2': 15.0 },
   { 'feature_1': -5.0, 'feature_3': 22.0 },
   { 'feature_3': -2.0, 'feature_4': 10.0 }
]
```

In this case, scikit-learn offers the classes `DictVectorizer` and `FeatureHasher`; they both produce sparse matrices of real numbers that can be fed into any machine learning model. The latter has a limited memory consumption and adopts **MurmurHash 3** (read `https://en.wikipedia.org/wiki/MurmurHash`, for further information). The code for these two methods is shown as follows:

```
from sklearn.feature_extraction import DictVectorizer, FeatureHasher

>>> dv = DictVectorizer()
>>> Y_dict = dv.fit_transform(data)

>>> Y_dict.todense()
matrix([[ 10.,  15.,   0.,   0.],
        [ -5.,   0.,  22.,   0.],
        [  0.,   0.,  -2.,  10.]])

>>> dv.vocabulary_
{'feature_1': 0, 'feature_2': 1, 'feature_3': 2, 'feature_4': 3}

>>> fh = FeatureHasher()
>>> Y_hashed = fh.fit_transform(data)

>>> Y_hashed.todense()
matrix([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

In both cases, I suggest you read the original scikit-learn documentation to know all possible options and parameters.

When working with categorical features (normally converted into positive integers through `LabelEncoder`), it's also possible to filter the dataset in order to apply one-hot encoding using the `OneHotEncoder` class. In the following example, the first feature is a binary index which indicates `'Male'` or `'Female'`:

```
from sklearn.preprocessing import OneHotEncoder

>>> data = [
   [0, 10],
   [1, 11],
   [1, 8],
   [0, 12],
   [0, 15]
]

>>> oh = OneHotEncoder(categorical_features=[0])
>>> Y_oh = oh.fit_transform(data1)
```

```
>>> Y_oh.todense()
matrix([[  1.,   0.,  10.],
        [  0.,   1.,  11.],
        [  0.,   1.,   8.],
        [  1.,   0.,  12.],
        [  1.,   0.,  15.]])
```

Considering that these approaches increase the number of values (also exponentially with binary versions), all the classes adopt sparse matrices based on SciPy implementation. See `https://docs.scipy.org/doc/scipy-0.18.1/reference/sparse.html` for further information.

Managing missing features

Sometimes a dataset can contain missing features, so there are a few options that can be taken into account:

- Removing the whole line
- Creating sub-model to predict those features
- Using an automatic strategy to input them according to the other known values

The first option is the most drastic one and should be considered only when the dataset is quite large, the number of missing features is high, and any prediction could be risky. The second option is much more difficult because it's necessary to determine a supervised strategy to train a model for each feature and, finally, to predict their value. Considering all pros and cons, the third option is likely to be the best choice. scikit-learn offers the class `Imputer`, which is responsible for filling the holes using a strategy based on the mean (default choice), median, or frequency (the most frequent entry will be used for all the missing ones).

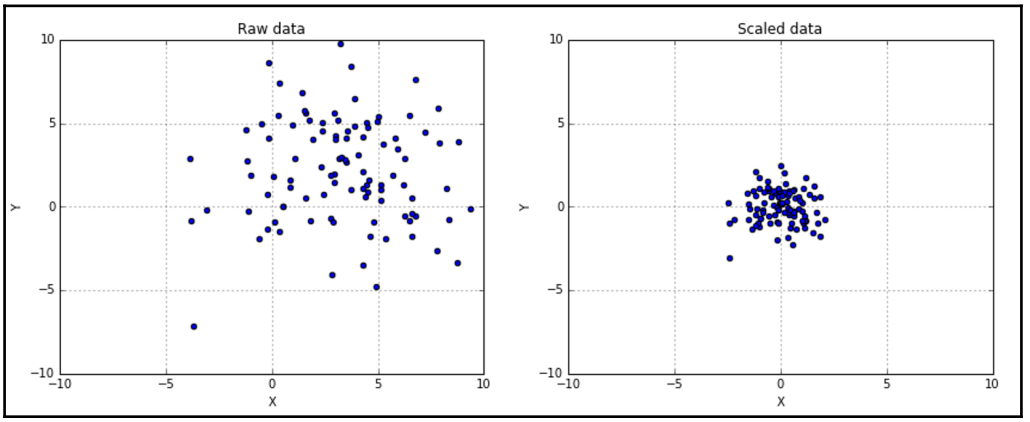
The following snippet shows an example using the three approaches (the default value for a missing feature entry is `NaN`. However, it's possible to use a different placeholder through the parameter `missing_values`):

```
from sklearn.preprocessing import Imputer

>>> data = np.array([[1, np.nan, 2], [2, 3, np.nan], [-1, 4, 2]])

>>> imp = Imputer(strategy='mean')
>>> imp.fit_transform(data)
array([[ 1. ,  3.5,  2. ],
       [ 2. ,  3. ,  2. ],
       [-1. ,  4. ,  2. ]])
```

```
>>> imp = Imputer(strategy='median')
>>> imp.fit_transform(data)
array([[ 1. ,  3.5,  2. ],
       [ 2. ,  3. ,  2. ],
       [-1. ,  4. ,  2. ]])

>>> imp = Imputer(strategy='most_frequent')
>>> imp.fit_transform(data)
array([[ 1.,  3.,  2.],
       [ 2.,  3.,  2.],
       [-1.,  4.,  2.]])
```

Data scaling and normalization

A generic dataset (we assume here that it is always numerical) is made up of different values which can be drawn from different distributions, having different scales and, sometimes, there are also outliers. A machine learning algorithm isn't naturally able to distinguish among these various situations, and therefore, it's always preferable to standardize datasets before processing them. A very common problem derives from having a non-zero mean and a variance greater than one. In the following figure, there's a comparison between a raw dataset and the same dataset scaled and centered:



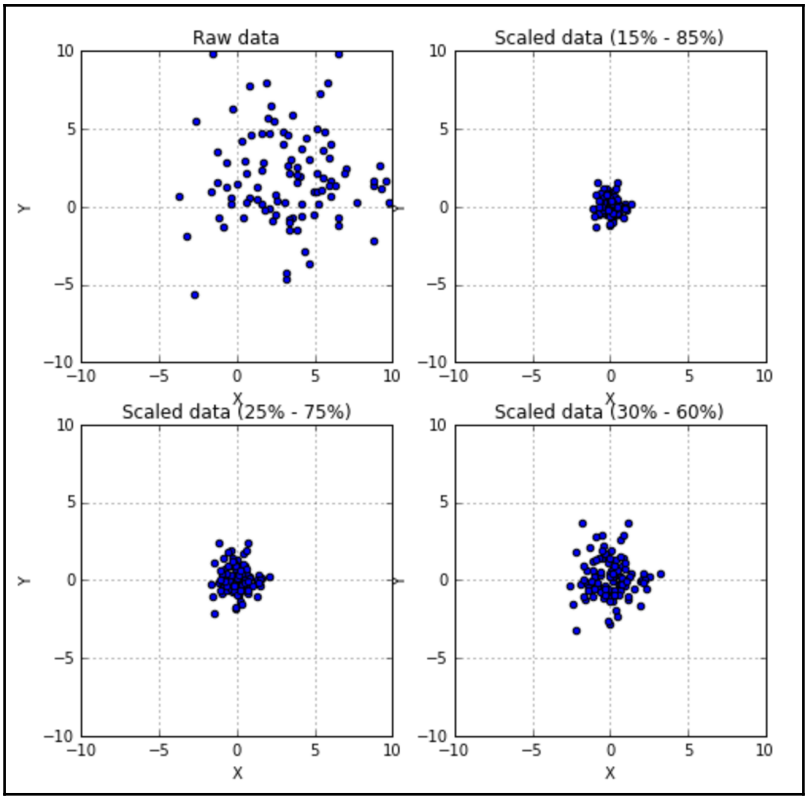
This result can be achieved using the `StandardScaler` class:

```
from sklearn.preprocessing import StandardScaler

>>> ss = StandardScaler()
>>> scaled_data = ss.fit_transform(data)
```

It's possible to specify if the scaling process must include both mean and standard deviation using the parameters `with_mean=True/False` and `with_std=True/False` (by default they're both active). If you need a more powerful scaling feature, with a superior control on outliers and the possibility to select a quantile range, there's also the class `RobustScaler`. Here are some examples with different quantiles:

```
from sklearn.preprocessing import RobustScaler

>>> rb1 = RobustScaler(quantile_range=(15, 85))
>>> scaled_data1 = rb1.fit_transform(data)

>>> rb1 = RobustScaler(quantile_range=(25, 75))
>>> scaled_data1 = rb1.fit_transform(data)

>>> rb2 = RobustScaler(quantile_range=(30, 60))
>>> scaled_data2 = rb2.fit_transform(data)
```

The results are shown in the following figures:



Other options include `MinMaxScaler` and `MaxAbsScaler`, which scale data by removing elements that don't belong to a given range (the former) or by considering a maximum absolute value (the latter).

scikit-learn also provides a class for per-sample normalization, `Normalizer`. It can apply `max`, `l1` and `l2` norms to each element of a dataset. In a Euclidean space, they are defined in the following way:

$$Max\ norm: \quad \|X\|_{max} = \frac{X}{|max_i\{X\}|}$$

$$L1\ norm: \quad \|X\|_{L1} = \frac{X}{\sum_i |x_i|}$$

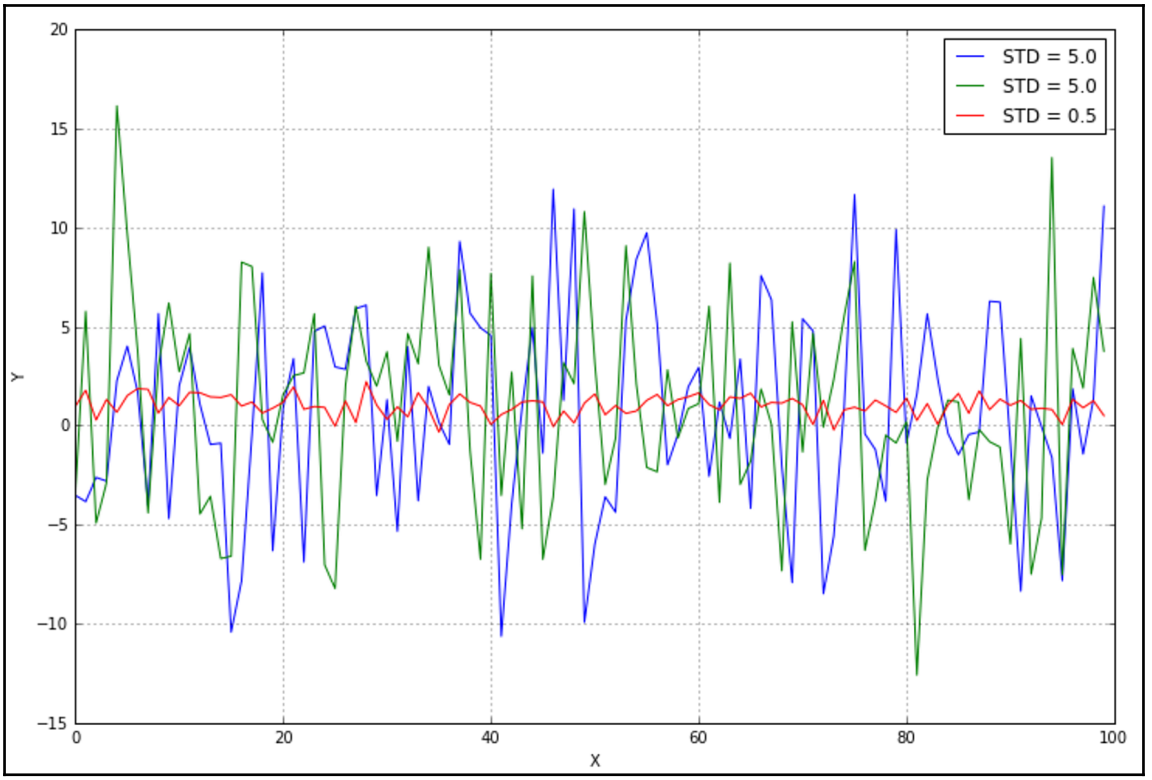
$$L2\ norm: \quad \|X\|_{L2} = \frac{X}{\sqrt{\sum_i |x_i|^2}}$$

An example of every normalization is shown next:

```
from sklearn.preprocessing import Normalizer

>>> data = np.array([1.0, 2.0])

>>> n_max = Normalizer(norm='max')
>>> n_max.fit_transform(data.reshape(1, -1))
[[ 0.5, 1. ]]

>>> n_l1 = Normalizer(norm='l1')
>>> n_l1.fit_transform(data.reshape(1, -1))
[[ 0.33333333,  0.66666667]]

>>> n_l2 = Normalizer(norm='l2')
>>> n_l2.fit_transform(data.reshape(1, -1))
[[ 0.4472136 ,  0.89442719]]
```

Feature selection and filtering

An unnormalized dataset with many features contains information proportional to the independence of all features and their variance. Let's consider a small dataset with three features, generated with random Gaussian distributions:



Even without further analysis, it's obvious that the central line (with the lowest variance) is almost constant and doesn't provide any useful information. If you remember the previous chapter, the entropy H(X) is quite small, while the other two variables carry more information. A variance threshold is, therefore, a useful approach to remove all those elements whose contribution (in terms of variability and so, information) is under a predefined level. scikit-learn provides the class `VarianceThreshold` that can easily solve this problem. By applying it on the previous dataset, we get the following result:

```
from sklearn.feature_selection import VarianceThreshold

>>> X[0:3, :]
array([[-3.5077778 , -3.45267063,  0.9681903 ],
       [-3.82581314,  5.77984656,  1.78926338],
       [-2.62090281, -4.90597966,  0.27943565]])

>>> vt = VarianceThreshold(threshold=1.5)
>>> X_t = vt.fit_transform(X)

>>> X_t[0:3, :]
array([[-0.53478521, -2.69189452],
       [-5.33054034, -1.91730367],
       [-1.17004376,  6.32836981]])
```

The third feature has been completely removed because its variance is under the selected threshold (1.5 in this case).

There are also many univariate methods that can be used in order to select the best features according to specific criteria based on F-tests and p-values, such as chi-square or ANOVA. However, their discussion is beyond the scope of this book and the reader can find further information in Freedman D., Pisani R., Purves R., *Statistics*, Norton & Company.

Two examples of feature selection that use the classes `SelectKBest` (which selects the best *K* high-score features) and `SelectPercentile` (which selects only a subset of features belonging to a certain percentile) are shown next. It's possible to apply them both to regression and classification datasets, being careful to select appropriate score functions:

```
from sklearn.datasets import load_boston, load_iris
from sklearn.feature_selection import SelectKBest, SelectPercentile, chi2,
f_regression

>>> regr_data = load_boston()
>>> regr_data.data.shape
(506L, 13L)

>>> kb_regr = SelectKBest(f_regression)
>>> X_b = kb_regr.fit_transform(regr_data.data, regr_data.target)
```

```
>>> X_b.shape
(506L, 10L)

>>> kb_regr.scores_
array([  88.15124178,   75.2576423 ,  153.95488314,   15.97151242,
        112.59148028,  471.84673988,   83.47745922,   33.57957033,
         85.91427767,  141.76135658,  175.10554288,   63.05422911,
        601.61787111])

>>> class_data = load_iris()
>>> class_data.data.shape
(150L, 4L)

>>> perc_class = SelectPercentile(chi2, percentile=15)
>>> X_p = perc_class.fit_transform(class_data.data, class_data.target)

>>> X_p.shape
(150L, 1L)

>>> perc_class.scores_
array([  10.81782088,    3.59449902,  116.16984746,   67.24482759])
```

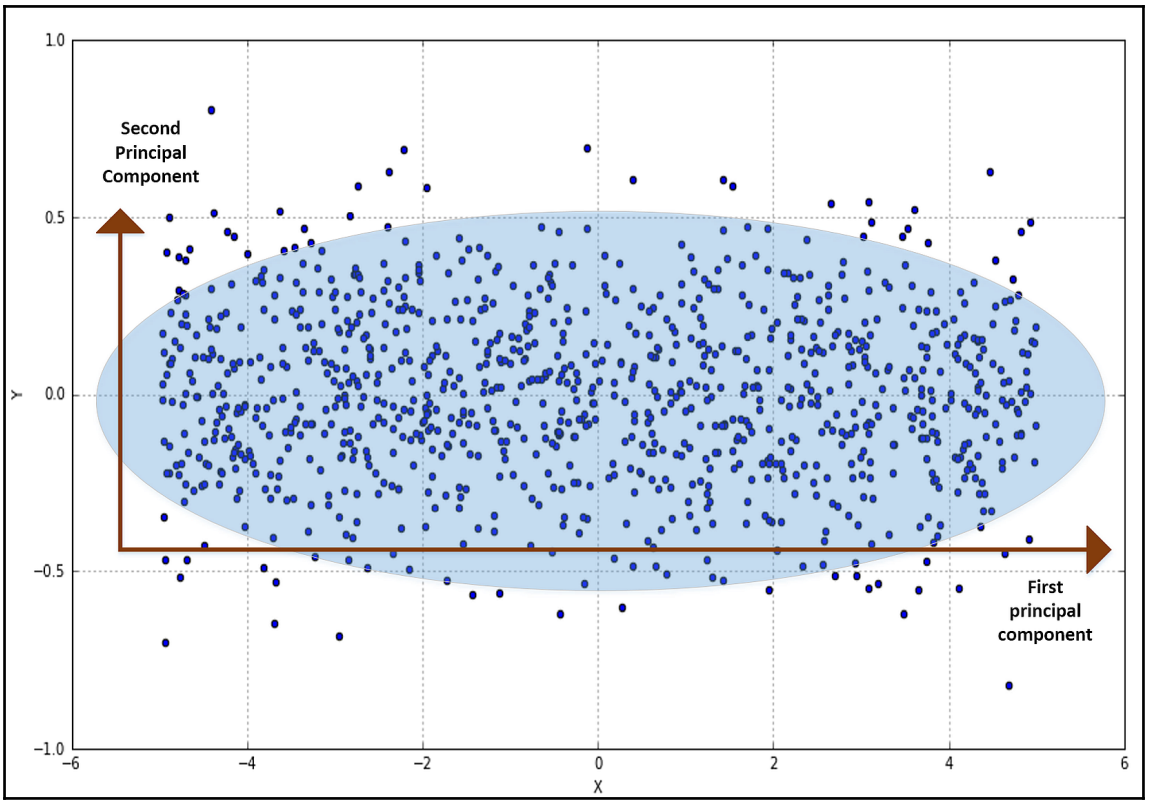
For further details about all scikit-learn score functions and their usage, visit `http://scikit-learn.org/stable/modules/feature_selection.html#univariate-feature-selection`.

Principal component analysis

In many cases, the dimensionality of the input dataset X is high and so is the complexity of every related machine learning algorithm. Moreover, the information is seldom spread uniformly across all the features and, as discussed in the previous chapter, there will be high entropy features together with low entropy ones, which, of course, don't contribute dramatically to the final outcome. In general, if we consider a Euclidean space, we have:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\} \; where \; \bar{x}_i \in \mathbb{R}^m \; and \;\; \bar{x}_i = x_{i1}\overrightarrow{e_1} + x_{i2}\overrightarrow{e_2} + \cdots + x_{im}\overrightarrow{e_m}$$

So each point is expressed using an orthonormal basis made of *m* linearly independent vectors. Now, considering a dataset *X*, a natural question arises: is it possible to reduce *m* without a drastic loss of information? Let's consider the following figure (without any particular interpretation):



It doesn't matter which distributions generated *X=(x,y)*, however, the variance of the horizontal component is clearly higher than the vertical one. As discussed, it means that the amount of information provided by the first component is higher and, for example, if the *x* axis is stretched horizontally keeping the vertical one fixed, the distribution becomes similar to a segment where the depth has lower and lower importance.

In order to assess how much information is brought by each component, and the correlation among them, a useful tool is the covariance matrix (if the dataset has zero mean, we can use the correlation matrix):

$$\boldsymbol{C} = \begin{pmatrix} {\sigma_1}^2 & \cdots & \sigma_{1m} \\ \vdots & \ddots & \vdots \\ \sigma_{m1} & \cdots & {\sigma_m}^2 \end{pmatrix}$$

$$where\ \sigma_{ij} = \frac{1}{m}\sum_k (x_{ki} - E[X_i])\left(x_{kj} - E\left[X_j\right]\right)$$

C is symmetric and positive semidefinite, so all the eigenvalues are non-negative, but what's the meaning of each value? The covariance matrix for the previous example is:

$$\boldsymbol{C} = \begin{pmatrix} 8.31 & -0.02 \\ -0.02 & 0.06 \end{pmatrix}$$

As expected, the horizontal variance is quite a bit higher than the vertical one. Moreover, the other values are close to zero. If you remember the definition and, for simplicity, remove the mean term, they represent the cross-correlation between couples of components. It's obvious that in our example, *X* and *Y* are uncorrelated (they're orthogonal), but in real-life examples, there could be features which present a residual cross-correlation. In terms of information theory, it means that knowing *Y* gives us some information about *X* (which we already know), so they share information which is indeed doubled. So our goal is also to decorrelate *X* while trying to reduce its dimensionality.

This can be achieved considering the sorted eigenvalues of *C* and selecting $g < m$ values:

$$\text{Let be } \Lambda = \{\lambda_1 \geq \lambda_2 \geq \cdots \lambda_m\}\ and\ \Lambda_g \subseteq \Lambda \text{ with } \dim(\Lambda_g) \leq \dim(\Lambda)$$

$$\boldsymbol{W} = \left(\vec{w}_{\lambda 1}, \vec{w}_{\lambda 2}, \ldots, \vec{w}_{\lambda g}\right)\ so\ that\ \bar{y}_R = \boldsymbol{W}\bar{y}\ \ where\ \bar{y}_R \in \mathbb{R}^g$$

So, it's possible to project the original feature vectors into this new (sub-)space, where each component carries a portion of total variance and where the new covariance matrix is decorrelated to reduce useless information sharing (in terms of correlation) among different features. In scikit-learn, there's the `PCA` class which can do all this in a very smooth way:

```
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA

>>> digits = load_digits()
```

A figure with a few random MNIST handwritten digits is shown as follows:

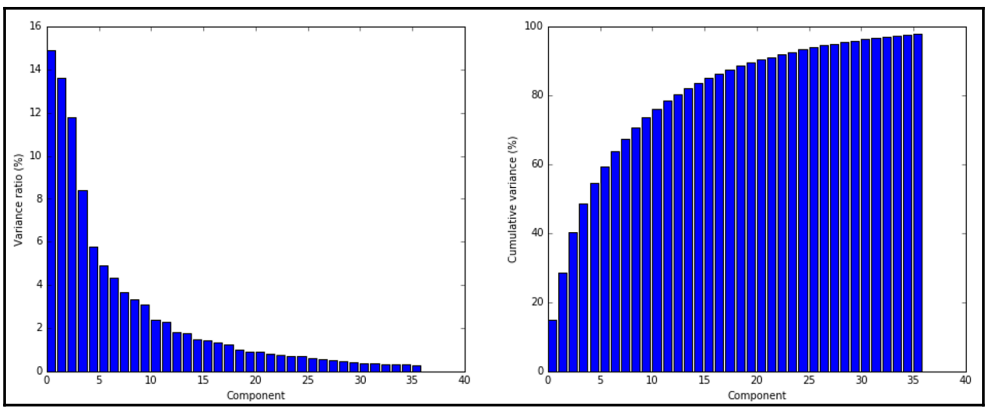
Each image is a vector of 64 unsigned int (8 bit) numbers (0, 255), so the initial number of components is indeed 64. However, the total amount of black pixels is often predominant and the basic signs needed to write 10 digits are similar, so it's reasonable to assume both high cross-correlation and a low variance on several components. Trying with 36 principal components, we get:

```
>>> pca = PCA(n_components=36, whiten=True)
>>> X_pca = pca.fit_transform(digits.data / 255)
```

In order to improve performance, all integer values are normalized into the range [0, 1] and, through the parameter `whiten=True`, the variance of each component is scaled to one. As also the official scikit-learn documentation says, this process is particularly useful when an isotropic distribution is needed for many algorithms to perform efficiently. It's possible to access the explained variance ratio through the instance variable `explained_variance_ratio_`, which shows which part of the total variance is carried by each single component:

```
>>> pca.explained_variance_ratio_
array([ 0.14890594,  0.13618771,  0.11794594,  0.08409979,  0.05782415,
        0.0491691 ,  0.04315987,  0.03661373,  0.03353248,  0.03078806,
        0.02372341,  0.02272697,  0.01821863,  0.01773855,  0.01467101,
        0.01409716,  0.01318589,  0.01248138,  0.01017718,  0.00905617,
        0.00889538,  0.00797123,  0.00767493,  0.00722904,  0.00695889,
        0.00596081,  0.00575615,  0.00515158,  0.00489539,  0.00428887,
        0.00373606,  0.00353274,  0.00336684,  0.00328029,  0.0030832 ,
        0.00293778])
```

A plot for the example of MNIST digits is shown next. The left graph represents the variance ratio while the right one is the cumulative variance. It can be immediately seen how the first components are normally the most important ones in terms of information, while the following ones provide details that a classifier could also discard:



As expected, the contribution to the total variance decreases dramatically starting from the fifth component, so it's possible to reduce the original dimensionality without an unacceptable loss of information, which could drive an algorithm to learn wrong classes. In the preceding graph, there are the same handwritten digits rebuilt using the first 36 components with whitening and normalization between 0 and 1. To obtain the original images, we need to inverse-transform all new vectors and project them into the original space:

```
>>> X_rebuilt = pca.inverse_transform(X_pca)
```

The result is shown in the following figure:

This process can also partially denoise the original images by removing residual variance, which is often associated with noise or unwanted contributions (almost every calligraphy distorts some of the structural elements which are used for recognition).

I suggest the reader try different numbers of components (using the explained variance data) and also `n_components='mle'`, which implements an automatic selection of the best dimensionality (Minka T.P, *Automatic Choice of Dimensionality for PCA*, NIPS 2000: 598-604).

scikit-learn solves the PCA problem with **SVD** (**Singular Value Decomposition**), which can be studied in detail in Poole D., *Linear Algebra*, Brooks Cole. It's possible to control the algorithm through the parameter `svd_solver`, whose values are `'auto', 'full', 'arpack', 'randomized'`. Arpack implements a truncated SVD. Randomized is based on an approximate algorithm which drops many singular vectors and can achieve very good performances also with high-dimensional datasets where the actual number of components is sensibly smaller.

Non-negative matrix factorization

When the dataset is made up of non-negative elements, it's possible to use **non-negative matrix factorization** (**NNMF**) instead of standard PCA. The algorithm optimizes a loss function (alternatively on *W* and *H*) based on the Frobenius norm:

$$L = \frac{1}{2}\|X - WH\|_{Frob}^2 \;\; where \;\; \|A\|_{Frob}^2 = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$

If *dim(X) = n x m*, then *dim(W) = n x p* and *dim(H) = p x m* with *p* equal to the number of requested components (the `n_components` parameter), which is normally smaller than the original dimensions *n* and *m*.

The final reconstruction is purely additive and it has been shown that it's particularly efficient for images or text where there are normally no non-negative elements. In the following snippet, there's an example using the Iris dataset (which is non-negative). The `init` parameter can assume different values (see the documentation) which determine how the data matrix is initially processed. A random choice is for non-negative matrices which are only scaled (no SVD is performed):

```
from sklearn.datasets import load_iris
from sklearn.decomposition import NMF

>>> iris = load_iris()
>>> iris.data.shape
(150L, 4L)

>>> nmf = NMF(n_components=3, init='random', l1_ratio=0.1)
>>> Xt = nmf.fit_transform(iris.data)

>>> nmf.reconstruction_err_
1.8819327624141866

>>> iris.data[0]
array([ 5.1,  3.5,  1.4,  0.2])
>>> Xt[0]
array([ 0.20668461,  1.09973772,  0.0098996 ])
>>> nmf.inverse_transform(Xt[0])
array([ 5.10401653,  3.49666967,  1.3965409 ,  0.20610779])
```

NNMF, together with other factorization methods, will be very useful for more advanced techniques, such as recommendation systems and topic modeling.

NNMF is very sensitive to its parameters (in particular, initialization and regularization), so I suggest reading the original documentation for further information: `http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html`.

Sparse PCA

scikit-learn provides different PCA variants that can solve particular problems. I do suggest reading the original documentation. However, I'd like to mention `SparsePCA`, which allows exploiting the natural sparsity of data while extracting principal components. If you think about the handwritten digits or other images that must be classified, their initial dimensionality can be quite high (a 10x10 image has 100 features). However, applying a standard PCA selects only the average most important features, assuming that every sample can be rebuilt using the same components. Simplifying, this is equivalent to:

$$y_R = c_1 y_{R1} + c_2 y_{R2} + \cdots + c_g y_{Rg}$$

On the other hand, we can always use a limited number of components, but without the limitation given by a dense projection matrix. This can be achieved by using sparse matrices (or vectors), where the number of non-zero elements is quite low. In this way, each element can be rebuilt using its specific components (in most cases, they will be always the most important), which can include elements normally discarded by a dense PCA. The previous expression now becomes:

$$y_R = (c_1 y_{R1} + c_2 y_{R2} + \cdots + c_g y_{Rg)} + (0 \cdot y_{Rg+1} + 0 \cdot y_{Rg+2} + \cdots + 0 \cdot y_{Rm})$$

Here the non-null components have been put into the first block (they don't have the same order as the previous expression), while all the other zero terms have been separated. In terms of linear algebra, the vectorial space now has the original dimensions. However, using the power of sparse matrices (provided by `scipy.sparse`), scikit-learn can solve this problem much more efficiently than a classical PCA.

The following snippet shows a sparse PCA with 60 components. In this context, they're usually called atoms and the amount of sparsity can be controlled via *L1*-norm regularization (higher `alpha` parameter values lead to more sparse results). This approach is very common in classification algorithms and will be discussed in the next chapters:

```
from sklearn.decomposition import SparsePCA

>>> spca = SparsePCA(n_components=60, alpha=0.1)
>>> X_spca = spca.fit_transform(digits.data / 255)

>>> spca.components_.shape
(60L, 64L)
```

For further information about SciPy sparse matrices, visit `https://docs.scipy.org/doc/scipy-0.18.1/reference/sparse.html`.

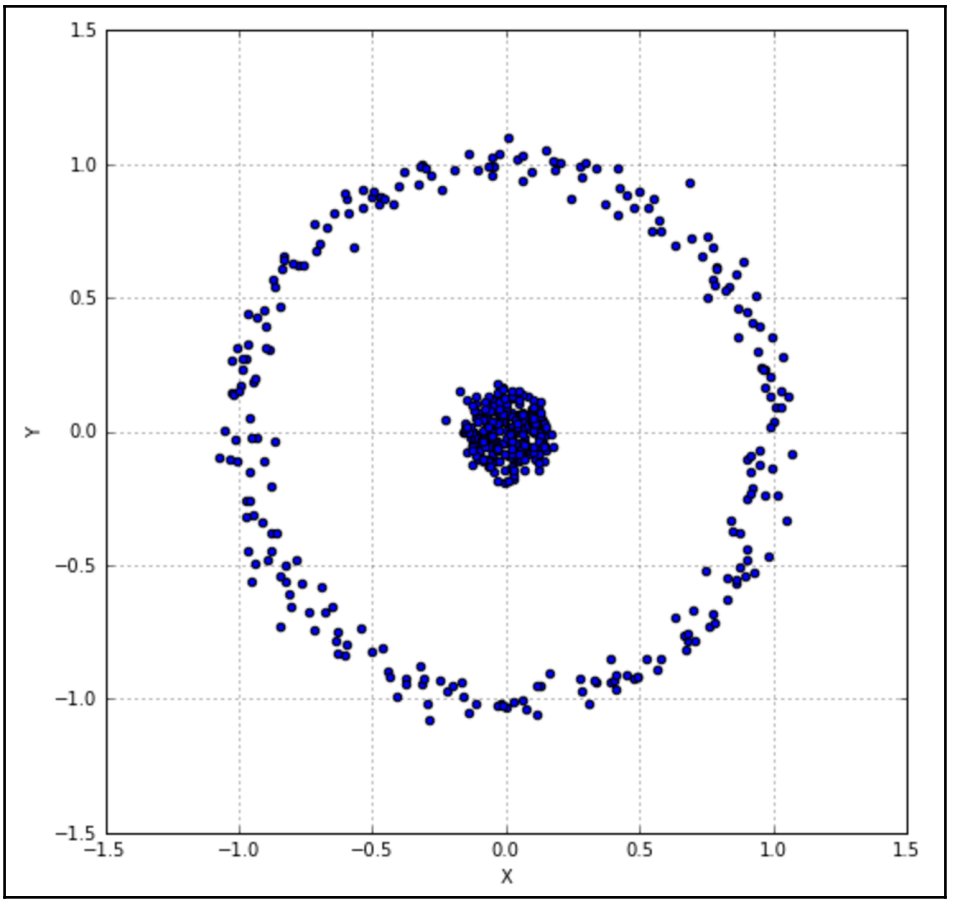
Kernel PCA

We're going to discuss kernel methods in `Chapter 7`, *Support Vector Machines*, however, it's useful to mention the class `KernelPCA`, which performs a PCA with non-linearly separable data sets. Just to understand the logic of this approach (the mathematical formulation isn't very simple), it's useful to consider a projection of each sample into a particular space where the dataset becomes linearly separable. The components of this space correspond to the first, second, ... principal components and a kernel PCA algorithm, therefore, computes the projection of our samples onto each of them.

Let's consider a dataset made up of a circle with a blob inside:

```
from sklearn.datasets import make_circles

>>> Xb, Yb = make_circles(n_samples=500, factor=0.1, noise=0.05)
```

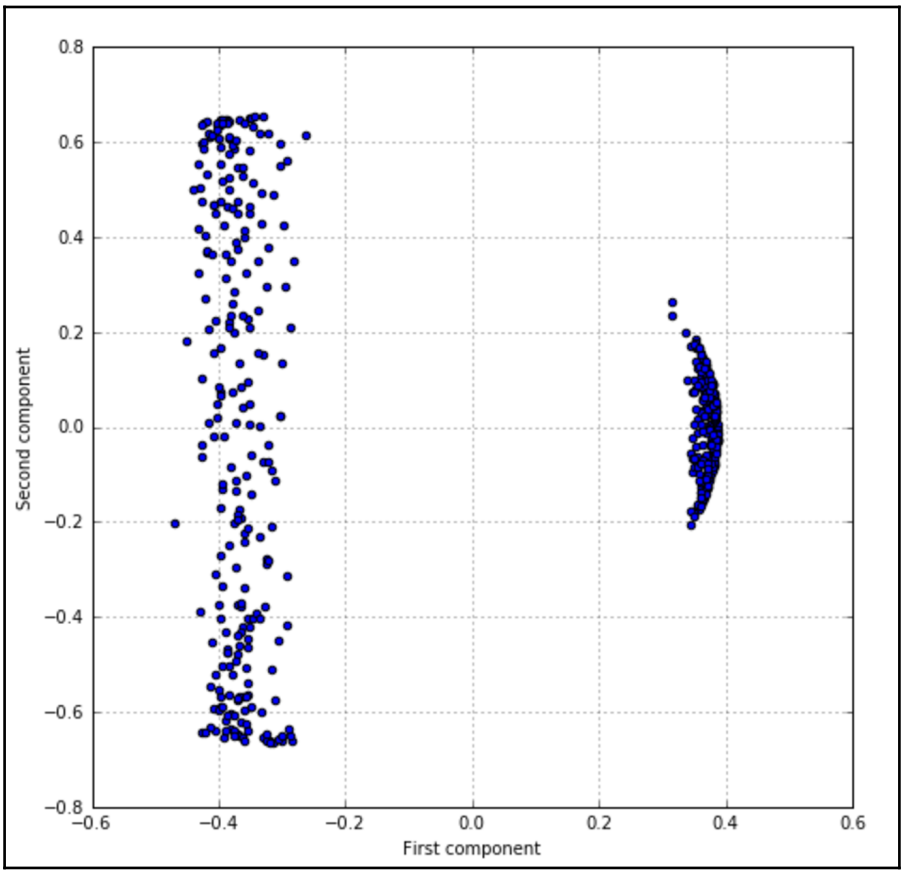
The graphical representation is shown in the following picture. In this case, a classic PCA approach isn't able to capture the non-linear dependency of existing components (the reader can verify that the projection is equivalent to the original dataset). However, looking at the samples and using polar coordinates (therefore, a space where it's possible to project all the points), it's easy to separate the two sets, only considering the radius:



Considering the structure of the dataset, it's possible to investigate the behavior of a PCA with a radial basis function kernel. As the default value for `gamma` is 1.0/number of features (for now, consider this parameter as inversely proportional to the variance of a Gaussian), we need to increase it to capture the external circle. A value of 1.0 is enough:

```
from sklearn.decomposition import KernelPCA

>>> kpca = KernelPCA(n_components=2, kernel='rbf',
fit_inverse_transform=True, gamma=1.0)
>>> X_kpca = kpca.fit_transform(Xb)
```

The instance variable `X_transformed_fit_` will contain the projection of our dataset into the new space. Plotting it, we get:



The plot shows a separation just like expected, and it's also possible to see that the points belonging to the central blob have a curve distribution because they are more sensitive to the distance from the center.

Kernel PCA is a powerful instrument when we think of our dataset as made up of elements that can be a function of components (in particular, radial-basis or polynomials) but we aren't able to determine a linear relationship among them.

For more information about the different kernels supported by scikit-learn, visit `http://scikit-learn.org/stable/modules/metrics.html#linear-kernel`.

Atom extraction and dictionary learning

Dictionary learning is a technique which allows rebuilding a sample starting from a sparse dictionary of atoms (similar to principal components). In Mairal J., Bach F., Ponce J., Sapiro G., *Online Dictionary Learning for Sparse Coding*, Proceedings of the 29th International Conference on Machine Learning, 2009 there's a description of the same online strategy adopted by scikit-learn, which can be summarized as a double optimization problem where:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\} \; where \; \bar{x}_i \in \mathbb{R}^m$$

Is an input dataset and the target is to find both a dictionary *D* and a set of weights for each sample:

$$\boldsymbol{D} \in \mathbb{R}^{m \times k} \; and \; \mathrm{A} = \{\bar{\alpha}_1, \bar{\alpha}_2, \ldots, \bar{\alpha}_m\} \; where \; \bar{\alpha}_i \in \mathbb{R}^k$$

After the training process, an input vector can be computed as:

$$\bar{x}_i = \boldsymbol{D}\bar{\alpha}_i$$

The optimization problem (which involves both *D* and alpha vectors) can be expressed as the minimization of the following loss function:

$$L(\boldsymbol{D}, A) = \frac{1}{2}\sum_i \|x_i - D\bar{\alpha}_i\|_2^2 + c\,\|\bar{\alpha}_i\|_1$$

Here the parameter *c* controls the level of sparsity (which is proportional to the strength of *L1* normalization). This problem can be solved by alternating the least square variable until a stable point is reached.

In scikit-learn, we can implement such an algorithm with the class `DictionaryLearning` (using the usual MNIST datasets), where `n_components`, as usual, determines the number of atoms:

```
from sklearn.decomposition import DictionaryLearning

>>> dl = DictionaryLearning(n_components=36, fit_algorithm='lars',
transform_algorithm='lasso_lars')
>>> X_dict = dl.fit_transform(digits.data)
```

A plot of each atom (component) is shown in the following figure:

This process can be very long on low-end machines. In such a case, I suggest limiting the number of samples to 20 or 30.

References

- Freedman D., Pisani R., Purves R., *Statistics,* Norton & Company
- Gareth J., Witten D., Hastie T., Tibshirani R., *An Introduction to Statistical Learning: With Application in R*, Springer
- Poole D., *Linear Algebra*, Brooks Cole
- Minka T.P, *Automatic Choice of Dimensionality for PCA*, NIPS 2000: 598-604
- Mairal J., Bach F., Ponce J., Sapiro G., *Online Dictionary Learning for Sparse Coding*, Proceedings of the 29th International Conference on Machine Learning, 2009

Summary

Feature selection is the first (and sometimes the most important) step in a machine learning pipeline. Not all the features are useful for our purposes and some of them are expressed using different notations, so it's often necessary to preprocess our dataset before any further operations.

We saw how to split the data into training and test sets using a random shuffle and how to manage missing elements. Another very important section covered the techniques used to manage categorical data or labels, which are very common when a certain feature assumes only a discrete set of values.

Then we analyzed the problem of dimensionality. Some datasets contain many features which are correlated with each other, so they don't provide any new information but increase the computational complexity and reduce the overall performances. Principal component analysis is a method to select only a subset of features which contain the largest amount of total variance. This approach, together with its variants, allows to decorrelate the features and reduce the dimensionality without a drastic loss in terms of accuracy. Dictionary learning is another technique used to extract a limited number of building blocks from a dataset, together with the information needed to rebuild each sample. This approach is particularly useful when the dataset is made up of different versions of similar elements (such as images, letters, or digits).

In the next chapter, we're going to discuss linear regression, which is the most diffused and simplest supervised approach to predict continuous values. We'll also analyze how to overcome some limitations and how to solve non-linear problems using the same algorithms.