12
Unsupervised Learning

In `Chapter 6`, *Machine Learning Process*, we discussed how unsupervised learning adds value by uncovering structures in the data without an outcome variable, such as a teacher, to guide the search process. This task contrasts with the setting for supervised learning that we focused on in the last several chapters.

Unsupervised learning algorithms can be useful when a dataset contains only features and no measurement of the outcome, or when we want to extract information independent of the outcome. Instead of predicting future outcomes, the goal is to study an informative representation of the data that is useful for solving another task, including the exploration of a dataset.

Examples include identifying topics to summarize documents (see `Chapter 14`, *Topic Modeling*), reducing the number of features to reduce the risk of overfitting and the computational cost for supervised learning, or grouping similar observations, as illustrated by the use of clustering for asset allocation at the end of this chapter.

Dimensionality reduction and clustering are the main tasks for unsupervised learning:

- **Dimensionality reduction** transforms the existing features into a new, smaller set, while minimizing the loss of information. A broad range of algorithms exists that differ only in how they measure the loss of information, whether they apply linear or non-linear transformations, or the constraints they impose on the new feature set.
- **Clustering algorithms** identify and group similar observations or features instead of identifying new features. Algorithms differ in how they define the similarity of observations and their assumptions about the resulting groups.

More specifically, this chapter covers the following:

- How **Principal Component Analysis** (**PCA**) and **Independent Component Analysis** (**ICA**) perform linear dimensionality reduction
- How to apply PCA to identify risk factors and eigen portfolios from asset returns
- How to use non-linear manifold learning to summarize high-dimensional data for effective visualization
- How to use t-SNE and UMAP to explore high-dimensional alternative image data
- How k-Means, hierarchical, and density-based clustering algorithms work
- How to use agglomerative clustering to build robust portfolios according to hierarchical risk parity

The code samples for each section are in the directory of the online GitHub repository for this chapter at `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`.

Dimensionality reduction

In linear algebra terms, the features of a dataset create a **vector space** whose dimensionality corresponds to the number of linearly independent columns (assuming there are more observations than features). Two columns are linearly dependent when they are perfectly correlated so that one can be computed from the other using the linear operations of addition and multiplication.

In other words, they are parallel vectors that represent the same rather than different directions or axes and only constitute a single dimension. Similarly, if one variable is a linear combination of several others, then it is an element of the vector space created by those columns, rather than adding a new dimension of its own.

The number of dimensions of a dataset matter because each new dimension can add a signal concerning an outcome. However, there is also a downside known as the **curse of dimensionality**: as the number of independent features grows while the number of observations remains constant, the average distance between data points also grows, and the density of the feature space drops exponentially.

The implications for machine learning are dramatic because prediction becomes much harder when observations are more distant; that is, different to each other. The next section addresses the resulting challenges.

Dimensionality reduction seeks to represent the information in the data more efficiently by using fewer features. To this end, algorithms project the data to a lower-dimensional space while discarding variation in the data that is not informative, or by identifying a lower-dimensional subspace or manifold on or near which the data lives.

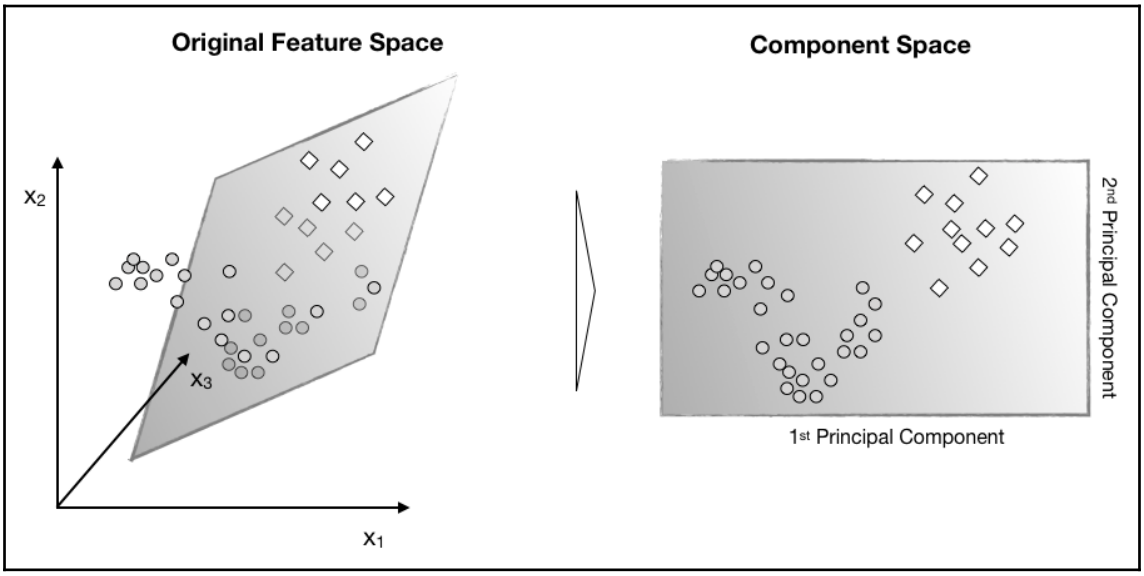
A manifold is a space that locally resembles Euclidean space. One-dimensional manifolds include lines and circles (but not screenshots of eight, due to the crossing point). The manifold hypothesis maintains that high-dimensional data often resides in a lower-dimensional space that, if identified, permits a faithful representation of the data in this subspace.

Dimensionality reduction thus compresses the data by finding a different, smaller set of variables that capture what matters most in the original features to minimize the loss of information. Compression helps counter the curse of dimensionality, economizes on memory, and permits the visualization of salient aspects of higher-dimensional data that is otherwise very difficult to explore.

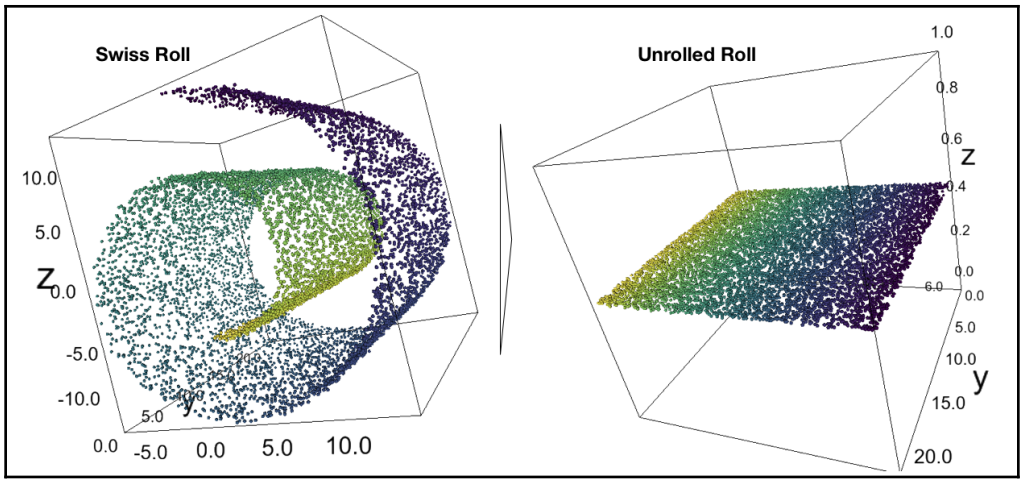
Linear and non-linear algorithms

Dimensionality reduction algorithms differ in the constraints they impose on the new variables and how they aim to minimize the loss of information:

- **Linear algorithms** such as PCA and ICA constrain the new variables to be linear combinations of the original features; that is, hyperplanes in a lower-dimensional space. Whereas PCA requires the new features to be uncorrelated, ICA goes further and imposes statistical independence—the absence of both linear and non-linear relationships. The following screenshot illustrates how PCA projects three-dimensional features into a two-dimensional space:



- Non-linear algorithms are not restricted to hyperplanes and can capture more complex structure in the data. However, given the infinite number of options, the algorithms still need to make assumptions to arrive at a solution. In this section, we show how **t-distributed Stochastic Neighbor Embedding** (**t-SNE**) and **Uniform Manifold Approximation and Projection** (**UMAP**) are very useful for visualizing higher-dimensional data. The following screenshot illustrates how manifold learning identifies a two-dimensional sub-space in the three-dimensional feature space (the `manifold_learning` notebook illustrates the use of additional algorithms, including local linear embedding):



The curse of dimensionality

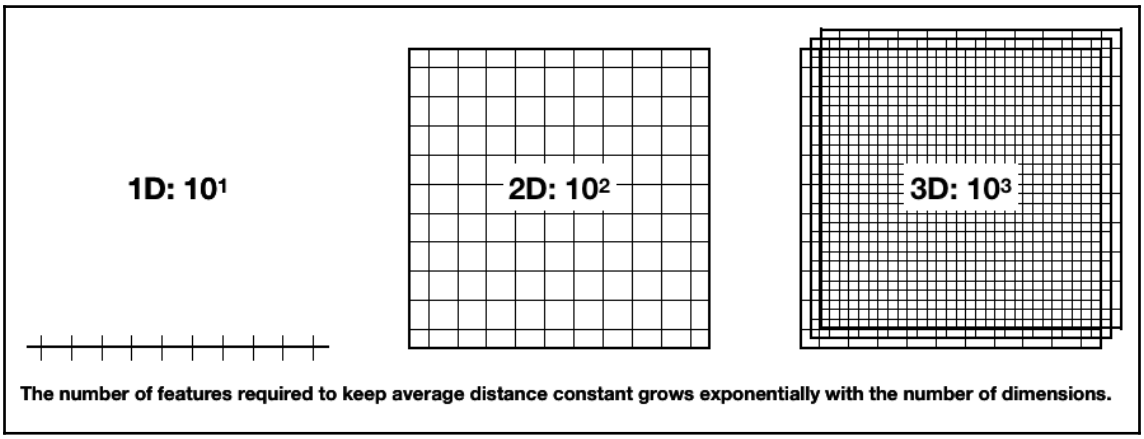
An increase in the number of dimensions of a dataset means there are more entries in the vector of features that represents each observation in the corresponding Euclidean space. We measure the distance in a vector space using Euclidean distance, also known as the **L2 norm**, which we applied to the vector of linear regression coefficients to train a regularized Ridge Regression model.

The Euclidean distance between two n-dimensional vectors with Cartesian coordinates $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$ is computed using the familiar formula developed by Pythagoras:

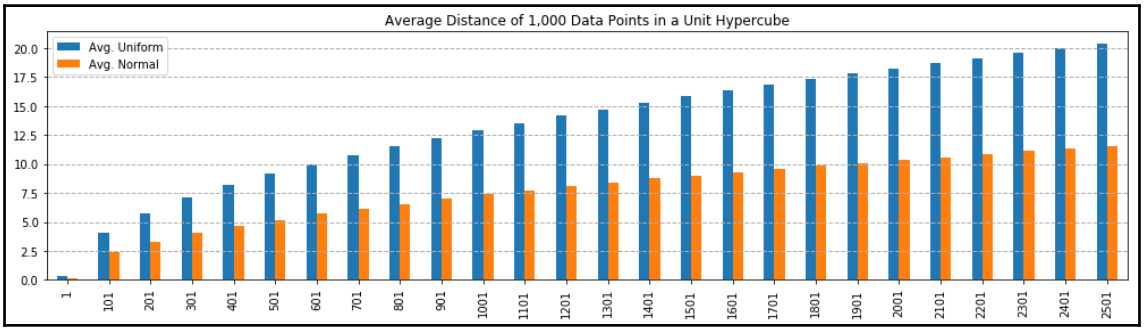
$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

Hence, each new dimension adds a non-negative term to the sum, so that the distance increases with the number of dimensions for distinct vectors. In other words, as the number of features grows for a given number of observations, the feature space becomes increasingly sparse; that is, less dense or emptier. On the flip side, the lower data density requires more observations to keep the average distance between data points the same.

The following chart shows how many data points we need to maintain the average distance of 10 observations uniformly distributed on a line. It increases exponentially from 10^1 in a single dimension to 10^2 in two and 10^3 in three dimensions, as the data needs to expand by a factor of 10 each time we add a new dimension:



The `curse_of_dimensionality` notebook in the GitHub repo folder for this section simulates how the average and minimum distances between data points increase as the number of dimensions grows:



The simulation draws features in the range [0, 1] from uncorrelated uniform or correlated normal distributions, and gradually increases the number of features to 2,500. The average distance between data points increases to over 11 times the feature range for features drawn from the normal distribution, and to over 20 times in the (extreme) case of uncorrelated uniform distribution.

When the distance between observations grows, supervised machine learning becomes more difficult because predictions for new samples are less likely to be based on learning from similar training features. Put differently, the number of possible unique rows grows exponentially as the number of features increases, which makes it so much harder to efficiently sample the space.

Similarly, the complexity of the functions learned by flexible algorithms that make fewer assumptions about the actual relationship grows exponentially with the number of dimensions.

Flexible algorithms include the tree-based models we saw in `Chapter 10`, *Decision Trees and Random Forests*, and `Chapter 11`, *Gradient Boosting Machines*, and the deep neural networks that we will cover from `Chapter 17`, *Deep Learning* onward. The variance of these algorithms increases as they get more opportunity to overfit to noise in more dimensions, resulting in poor generalization performance.

In practice, features are correlated, often substantially so, or do not exhibit much variation. For these reasons, dimensionality reduction helps to compress the data without losing much of the signal, and combat the curse while also economizing on memory. In these cases, it complements the use of regularization to manage prediction error due to variance and model complexity.

The critical question that we take on in the following section then becomes: what are the best ways to find a lower-dimensional representation of the data that loses as little information as possible?

Linear dimensionality reduction

Linear dimensionality reduction algorithms compute linear combinations that translate, rotate, and rescale the original features to capture significant variation in the data, subject to constraints on the characteristics of the new features.

Principal Component Analysis (**PCA**), invented in 1901 by Karl Pearson, finds new features that reflect directions of maximal variance in the data while being mutually uncorrelated, or orthogonal.

Independent Component Analysis (**ICA**), in contrast, originated in signal processing in the 1980s, with the goal of separating different signals while imposing the stronger constraint of statistical independence.

This section introduces these two algorithms and then illustrates how to apply PCA to asset returns to learn risk factors from the data, and to build so-called eigen portfolios for systematic trading strategies.

Principal Component Analysis

PCA finds principal components as linear combinations of the existing features and uses these components to represent the original data. The number of components is a hyperparameter that determines the target dimensionality and needs to be equal to or smaller than the number of observations or columns, whichever is smaller.

PCA aims to capture most of the variance in the data, to make it easy to recover the original features and so that each component adds information. It reduces dimensionality by projecting the original data into the principal component space.

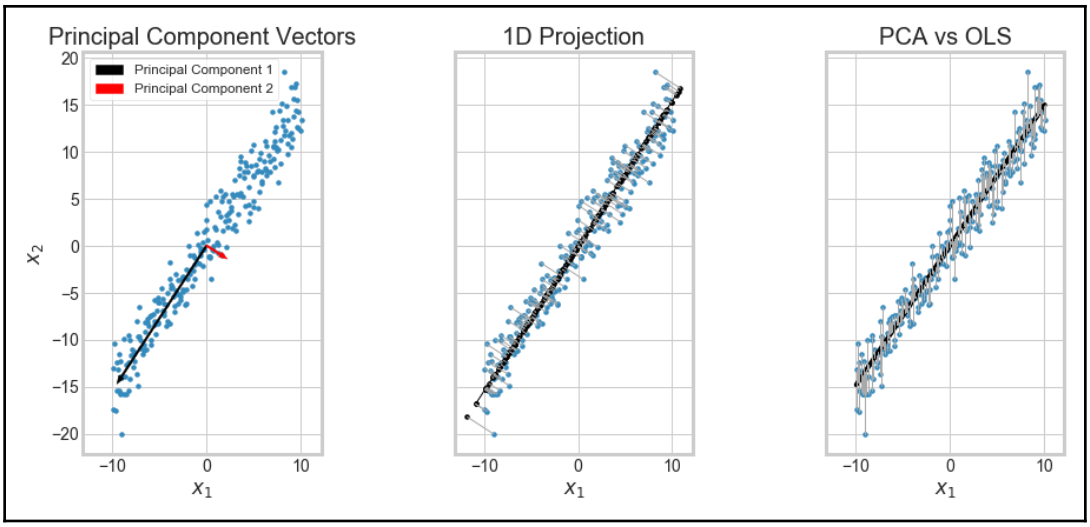
The PCA algorithm works by identifying a sequence of principal components, each of which aligns with the direction of maximum variance in the data after accounting for variation captured by previously-computed components. The sequential optimization also ensures that new components are not correlated with existing components so that the resulting set constitutes an orthogonal basis for a vector space.

This new basis corresponds to a rotated version of the original basis so that the new axis point in the direction of successively decreasing variance. The decline in the amount of variance of the original data explained by each principal component reflects the extent of correlation among the original features.

The number of components that capture, for example, 95% of the original variation relative to the total number of features provides an insight into the linearly-independent information in the original data.

Visualizing PCA in 2D

The following screenshot illustrates several aspects of PCA for a two-dimensional random dataset (see the `pca_key_ideas` notebook):



- The left panel shows how the first and second principal components align with the directions of maximum variance while being orthogonal.
- The central panel shows how the first principal component minimizes the reconstruction error, measured as the sum of the distances between the data points and the new axis.
- Finally, the right panel illustrates supervised OLS, which approximates the outcome variable (here we choose x_2) by a (one-dimensional) hyperplane computed from the (single) feature. The vertical lines highlight how OLS minimizes the distance along the outcome axis, in contrast with PCA, which minimizes the distances orthogonal to the hyperplane.

The assumptions made by PCA

PCA makes several assumptions that are important to keep in mind. These include the following:

- High variance implies a high signal-to-noise ratio
- The data is standardized so that the variance is comparable across features
- Linear transformations capture the relevant aspects of the data
- Higher-order statistics beyond the first and second moment do not matter, which implies that the data has a normal distribution

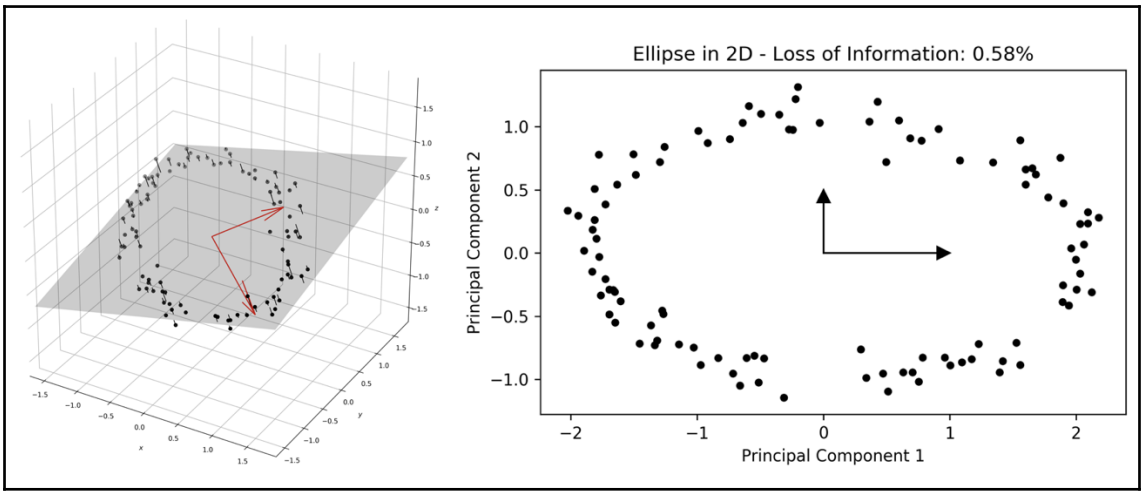
The emphasis on the first and second moments aligns with standard risk/return metrics, but the normality assumption may conflict with the characteristics of market data.

How the PCA algorithm works

The algorithm finds vectors to create a hyperplane of target dimensionality that minimizes the reconstruction error, measured as the sum of the squared distances of the data points to the plane. As illustrated above, this goal corresponds to finding a sequence of vectors that align with directions of maximum retained variance given the other components while ensuring all principal components are mutually orthogonal.

In practice, the algorithm solves the problem either by computing the eigenvectors of the covariance matrix or using the singular value decomposition.

We illustrate the computation using a randomly generated three-dimensional ellipse with 100 data points, shown in the left panel of the following screenshot, including the two-dimensional hyperplane defined by the first two principal components (see the `the_math_behind_pca` notebook for the following code samples):



Three-dimensional ellipse and two-dimensional hyperplane

PCA based on the covariance matrix

We first compute the principal components using the square covariance matrix with the pairwise sample covariances for the features x_i, x_j, $i, j = 1, ..., n$ as entries in row i and column j:

$$cov_{i,j} = \frac{\sum_{k=1}^{N}(x_{ik} - \bar{x_i})(x_{jk} - \bar{x_j})}{N-1}$$

For a square matrix M of n dimensions, we define the eigenvectors ω_i and eigenvalues λ_i, i=1, ..., n as follows:

$$Mw_i = \lambda_i w_i$$

Hence, we can represent the matrix M using eigenvectors and eigenvalues, where W is a matrix that contains the eigenvectors as column vectors, and L is a matrix that contains the λ_i as diagonal entries (and 0s otherwise). We define the eigendecomposition as follows:

$$M = WLW^{-1}$$

Using NumPy, we implement this as follows, where the pandas DataFrame contains the 100 data points of the ellipse:

```
# compute covariance matrix:
cov = np.cov(data, rowvar=False) # expects variables in rows by default
cov.shape
(3, 3)
```

Next, we calculate the eigenvectors and eigenvalues of the covariance matrix. The eigenvectors contain the principal components (where the sign is arbitrary):

```
eigen_values, eigen_vectors = eig(cov)
eigen_vectors
array([[ 0.71409739, -0.66929454, -0.20520656],
[-0.70000234, -0.68597301, -0.1985894 ],
[ 0.00785136, -0.28545725, 0.95835928]])
```

We can compare the result with the result obtained from sklearn, and find that they match in absolute terms:

```
pca = PCA()
pca.fit(data)
C = pca.components_.T # columns = principal components
C
array([[ 0.71409739, 0.66929454, 0.20520656],
[-0.70000234, 0.68597301, 0.1985894 ],
[ 0.00785136, 0.28545725, -0.95835928]])
np.allclose(np.abs(C), np.abs(eigen_vectors))
True
```

We can also verify the eigendecomposition, starting with the diagonal matrix *L* that contains the eigenvalues:

```
# eigenvalue matrix
ev = np.zeros((3, 3))
np.fill_diagonal(ev, eigen_values)
ev # diagonal matrix
array([[1.92923132, 0. , 0. ],
[0. , 0.55811089, 0. ],
[0. , 0. , 0.00581353]])
```

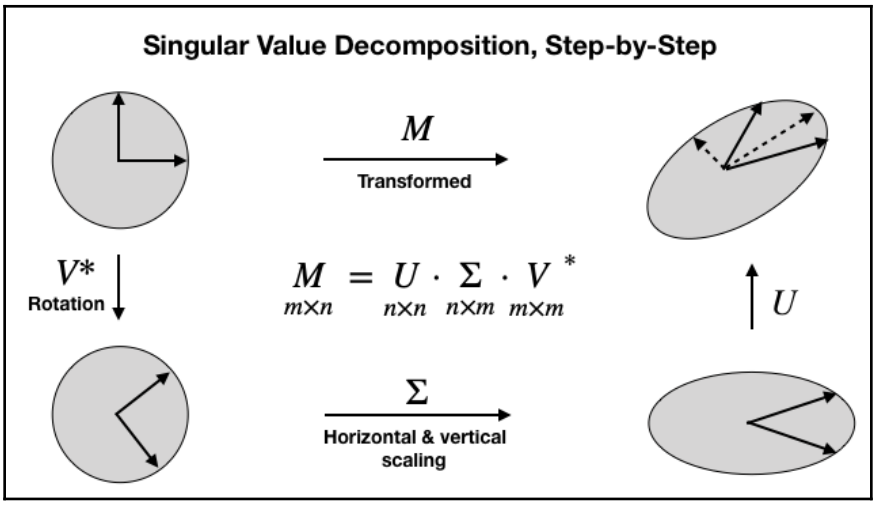
We find that the result does indeed hold:

```
decomposition = eigen_vectors.dot(ev).dot(inv(eigen_vectors))
np.allclose(cov, decomposition)
```

PCA using Singular Value Decomposition

Next, we'll look at the alternative computation using **Singular Value Decomposition** (**SVD**). This algorithm is slower when the number of observations is greater than the number of features (the typical case), but yields better numerical stability, especially when some of the features are strongly correlated (often the reason to use PCA in the first place).

SVD generalizes the eigendecomposition that we just applied to the square and symmetric covariance matrix to the more general case of *m* x *n* rectangular matrices. It has the form shown at the center of the following diagram. The diagonal values of Σ are the singular values, and the transpose of V^* contains the principal components as column vectors:



In this case, we need to make sure our data is centered with mean zero (the computation of the covariance preceding took care of this):

```
n_features = data.shape[1]
data_ = data - data.mean(axis=0
Using the centered data, we compute the singular value decomposition:
U, s, Vt = svd(data_)
U.shape, s.shape, Vt.shape
((100, 100), (3,), (3, 3))
We can convert the vector s that only contains the singular values into an
nxm matrix and show that the decomposition works:
S = np.zeros_like(data_)
S[:n_features, :n_features] = np.diag(s)
S.shape
(100, 3)
```

We find that the decomposition does indeed reproduce the standardized data:

```
np.allclose(data_, U.dot(S).dot(Vt))
True
```

Lastly, we confirm that the columns of the transpose of V^* contain the principal components:

```
np.allclose(np.abs(C), np.abs(Vt.T))
True
```

In the next section, we show how sklearn implements PCA.

PCA with sklearn

The `sklearn.decomposition.PCA` implementation follows the standard API based on the `fit()` and `transform()` methods, which compute the desired number of principal components and project the data into the component space, respectively. The convenience method `fit_transform()` accomplishes this in a single step.

PCA offers three different algorithms that can be specified using the `svd_solver` parameter:

- `Full` computes the exact SVD using the LAPACK solver provided by SciPy
- `Arpack` runs a truncated version suitable for computing less than the full number of components
- `Randomized` uses a sampling-based algorithm that is more efficient when the dataset has more than 500 observations and features, and the goal is to compute less than 80% of the components
- `Auto` uses randomized where most efficient, otherwise, it uses the full SVD

See references on GitHub for algorithmic implementation details.

Other key configuration parameters of the PCA object are as follows:

- `n_components`: These compute all principal components by passing `None` (the default), or limit the number to `int`. For `svd_solver=full`, there are two additional options: a float in the interval [0, 1] computes the number of components required to retain the corresponding share of the variance in the data, and the `mle` option estimates the number of dimensions using maximum likelihood.
- `whiten`: If `True`, it standardizes the component vectors to unit variance that, in some cases, can be useful for use in a predictive model (the default is `False`).

To compute the first two principal components of the three-dimensional ellipsis and project the data into the new space, use `fit_transform()` as follows:

```
pca = PCA(n_components=2)
projected_data = pca.fit_transform(data)
projected_data.shape
(100, 2)
```

The explained variance of the first two components is very close to 100%:

```
pca2.explained_variance_ratio_
array([0.77381099, 0.22385721])
```

The screenshot at the beginning of this section shows the projection of the data into the new two-dimensional space.

Independent Component Analysis

Independent Component Analysis (**ICA**) is another linear algorithm that identifies a new basis on which to represent the original data, but pursues a different objective to PCA.

ICA emerged in signal processing, and the problem it aims to solve is called **blind source separation**. It is typically framed as the cocktail party problem, in which a given number of guests are speaking at the same time so that a single microphone would record overlapping signals. ICA assumes there are as many different microphones as there are speakers, each placed at different locations so as to record a different mix of the signals. ICA then aims to recover the individual signals from the different recordings.

In other words, there are n original signals and an unknown square mixing matrix A that produces an n-dimensional set of m observations, so that:

$$\underset{n\times m}{X} = \underset{n\times n}{A} \quad \underset{n\times m}{s}$$

The goal is to find the matrix $W=A^{-1}$ that untangles the mixed signals to recover the sources.

The ability to uniquely determine the matrix W hinges on the non-Gaussian distribution of the data. Otherwise, W could be rotated arbitrarily given the multivariate normal distribution's symmetry under rotation.

Furthermore, ICA assumes the mixed signal is the sum of its components and is unable to identify Gaussian components because their sum is also normally distributed.

ICA assumptions

ICA makes the following critical assumptions:

- The sources of the signals are statistically independent
- Linear transformations are sufficient to capture the relevant information
- The independent components do not have a normal distribution
- The mixing matrix A can be inverted

ICA also requires the data to be centered and whitened; that is, to be mutually uncorrelated with unit variance. Preprocessing the data using PCA as outlined above achieves the required transformations.

The ICA algorithm

FastICA, used by sklearn, is a fixed-point algorithm that uses higher-order statistics to recover the independent sources. In particular, it maximizes the distance to a normal distribution for each component as a proxy for independence.

An alternative algorithm called **InfoMax** minimizes the mutual information between components as a measure of statistical independence.

ICA with sklearn

The ICA implementation by sklearn uses the same interface as PCA, so there is little to add. Note that there is no measure of explained variance because ICA does not compute components successively. Instead, each component aims to capture independent aspects of the data.

PCA for algorithmic trading

PCA is useful for algorithmic trading in several respects. These include the data-driven derivation of risk factors by applying PCA to asset returns, and the construction of uncorrelated portfolios based on the principal components of the correlation matrix of asset returns.

Data-driven risk factors

In `Chapter 7`, *Linear Models*, we explored risk factor models used in quantitative finance to capture the main drivers of returns. These models explain differences in returns on assets based on their exposure to systematic risk factors and the rewards associated with these factors.

In particular, we explored the Fama-French approach, which specifies factors based on prior knowledge about the empirical behavior of average returns, treats these factors as observable, and then estimates risk model coefficients using linear regression. An alternative approach treats risk factors as latent variables and uses factor analytic techniques such as PCA to simultaneously estimate the factors and how they drive returns from historical returns.

In this section, we will review how this method derives factors in a purely statistical or data-driven way, with the advantage of not requiring ex-ante knowledge of the behavior of asset returns (see the `pca` and `risk_factor` notebook models for details).

We will use the Quandl stock price data and select the daily adjusted close prices of the 500 stocks with the largest market capitalization and data for the 2010-18 period. We then compute the daily returns as follows:

```
idx = pd.IndexSlice
with pd.HDFStore('../../data/assets.h5') as store:
stocks = store['us_equities/stocks'].marketcap.nlargest(500)
returns = (store['quandl/wiki/prices']
.loc[idx['2010': '2018', stocks.index], 'adj_close']
.unstack('ticker')
.pct_change())
```

We obtain `351` stocks and returns for over 2,000 trading days:

```
returns.info()
DatetimeIndex: 2072 entries, 2010-01-04 to 2018-03-27
Columns: 351 entries, A to ZTS
```

PCA is sensitive to outliers, so we winsorize the data at the 2.5% and 97.5% quantiles:

```
returns = returns.clip(lower=returns.quantile(q=.025),
upper=returns.quantile(q=.975),
axis=1)
```

PCA does not permit missing data, so we will remove stocks that do not have data for at least 95% of the time period, and in a second step, remove trading days that do not have observations on at least 95% of the remaining stocks:

```
returns = returns.dropna(thresh=int(returns.shape[0] * .95), axis=1)
returns = returns.dropna(thresh=int(returns.shape[1] * .95))
```

We are left with `314` equity return series covering a similar period:

```
returns.info()
DatetimeIndex: 2070 entries, 2010-01-05 to 2018-03-27
Columns: 314 entries, A to ZBH
```

We impute any remaining missing values using the average return for any given trading day:

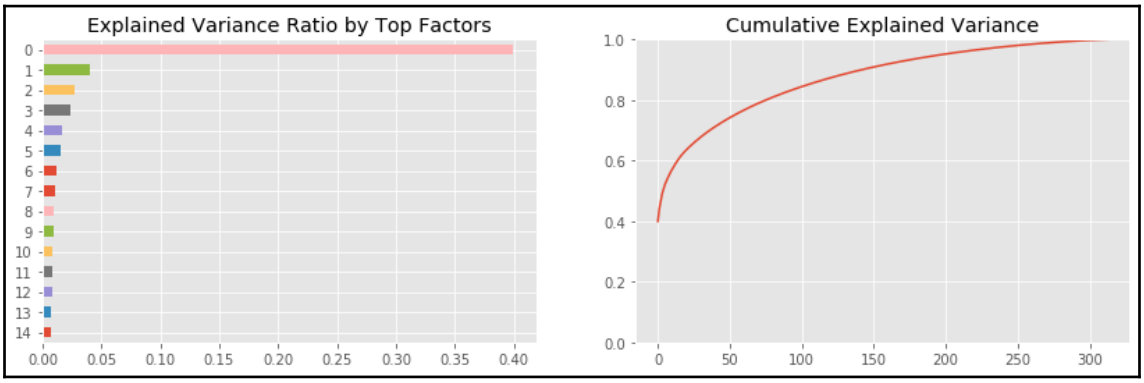
```
daily_avg = returns.mean(1)
returns = returns.apply(lambda x: x.fillna(daily_avg))
```

Now we are ready to fit the principal components model to the asset returns using default parameters to compute all components using the full SVD algorithm:

```
pca = PCA()
pca.fit(returns)
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
svd_solver='auto', tol=0.0, whiten=False)
```

We find that the most important factor explains around 40% of the daily return variation. The dominant factor is usually interpreted as the market, whereas the remaining factors can be interpreted as industry or style factors, in line with our discussion in `Chapter 5`, *Strategy Evaluation*, and `Chapter 7`, *Linear Models*, depending on the results of closer inspection (see the next example).

The plot on the right shows the cumulative explained variance, and indicates that around 10 factors explain 60% of the returns of this large cross-section of stocks:



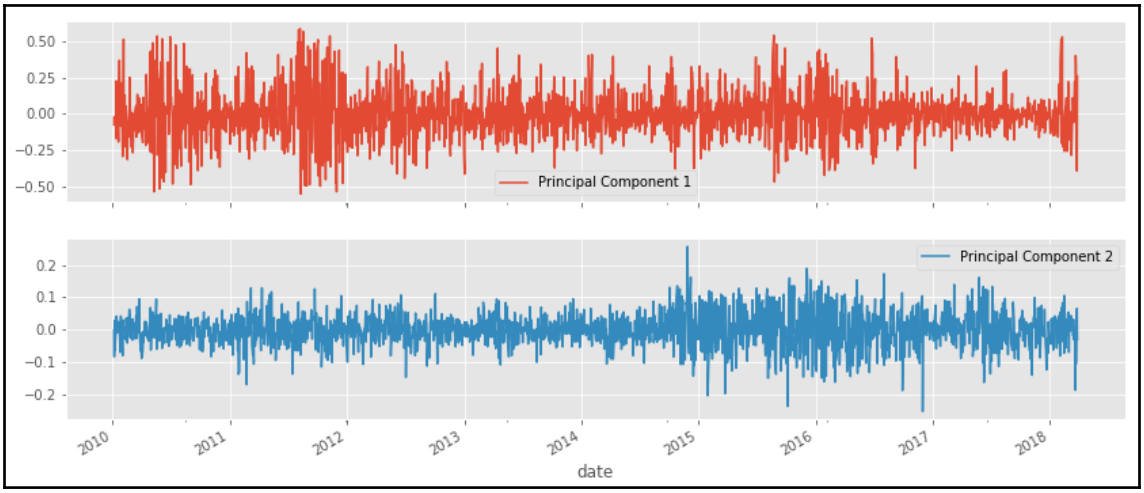
The notebook contains a simulation for a broader cross-section of stocks and the longer 2000-18 time period. It finds that, on average, the first three components explained 25%, 10%, and 5% of 500 randomly selected stocks.

The cumulative plot shows a typical elbow pattern that can help identify a suitable target dimensionality because it indicates that additional components add less explanatory value.

We can select the top two principal components to verify that they are indeed uncorrelated:

```
risk_factors = pd.DataFrame(pca.transform(returns)[:, :2],
columns=['Principal Component 1', 'Principal Component 2'],
index=returns.index)
risk_factors['Principal Component 1'].corr(risk_factors['Principal
Component 2'])
7.773256996252084e-15
```

Moreover, we can plot the time series to highlight how each factor captures different volatility patterns:



A risk factor model would employ a subset of the principal components as features to predict future returns, similar to our approach in `Chapter 7`, *Linear Models – Regression and Classification*.

Eigen portfolios

Another application of PCA involves the covariance matrix of the normalized returns. The principal components of the correlation matrix capture most of the covariation among assets in descending order and are mutually uncorrelated. Moreover, we can use standardized principal components as portfolio weights.

Let's use the 30 largest stocks with data for the 2010-2018 period to facilitate the exposition:

```
idx = pd.IndexSlice
with pd.HDFStore('../../data/assets.h5') as store:
stocks = store['us_equities/stocks'].marketcap.nlargest(30)
returns = (store['quandl/wiki/prices']
.loc[idx['2010': '2018', stocks.index], 'adj_close']
.unstack('ticker')
.pct_change())
```

We again winsorize and also normalize the returns:

```
normed_returns = scale(returns
       .clip(lower=returns.quantile(q=.025),
        upper=returns.quantile(q=.975),
        axis=1)
.apply(lambda x: x.sub(x.mean()).div(x.std())))
```

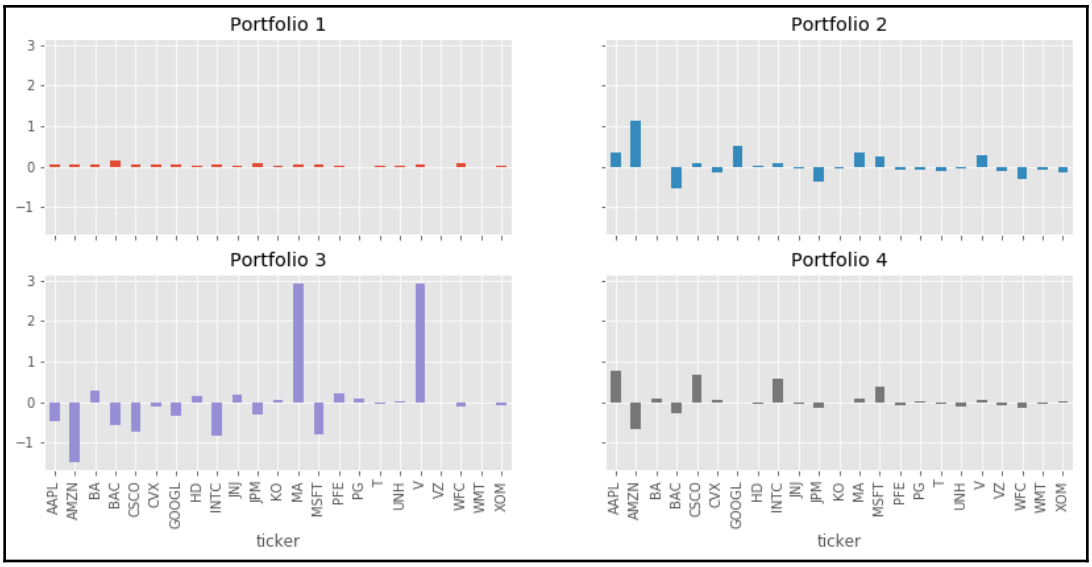
After dropping assets and trading days as in the previous example, we are left with 23 assets and over 2,000 trading days. We estimate all principal components, and find that the two largest explain 55.9% and 15.5% of the covariation, respectively:

```
pca.fit(cov)
pd.Series(pca.explained_variance_ratio_).head()
0 55.91%
1 15.52%
2 5.36%
3 4.85%
4 3.32%
```

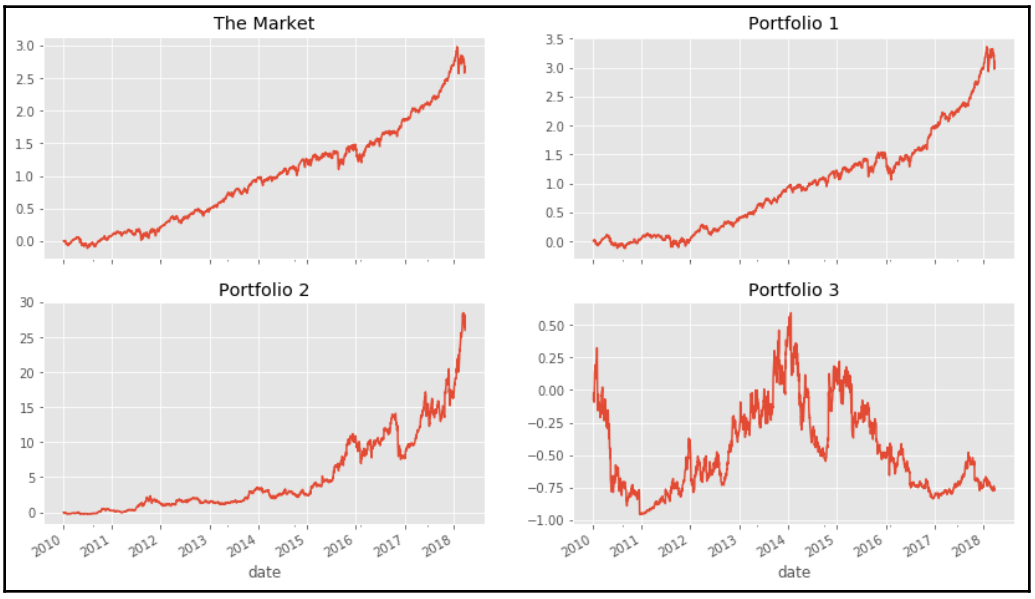
Next, we select and normalize the four largest components so that they sum to `1` and we can use them as weights for portfolios that we can compare to an equal-weighted portfolio formed from all stocks:

```
top4 = pd.DataFrame(pca.components_[:4], columns=cov.columns)
eigen_portfolios = top4.div(top4.sum(1), axis=0)
eigen_portfolios.index = [f'Portfolio {i}' for i in range(1, 5)]
```

The weights show distinct emphasis—for example, **Portfolio 3** puts large weights on Mastercard and Visa, the two payment processors in the sample, whereas **Portfolio 2** has more exposure to technology companies:



When comparing the performance of each portfolio over the sample period to **The Market** consisting of our small sample, we find that portfolio 1 performs very similarly, whereas the other portfolios capture different return patterns:



Comparing performances of each portfolio

Manifold learning

Linear dimensionality reduction projects the original data onto a lower-dimensional hyperplane that aligns with informative directions in the data. The focus on linear transformations simplifies the computation and echoes common financial metrics, such as PCA's goal to capture the maximum variance.

However, linear approaches will naturally ignore signal reflected in non-linear relationships in the data. Such relationships are very important in alternative datasets containing, for example, image or text data. Detecting such relationships during exploratory analysis can provide important clues about the data's potential signal content.

In contrast, the manifold hypothesis emphasizes that high-dimensional data often lies on or near a lower-dimensional non-linear manifold that is embedded in the higher dimensional space. The two-dimensional swiss roll displayed in the screenshot at the beginning of this section illustrates such a topological structure.

Manifold learning aims to find the manifold of intrinsic dimensionality and then represent the data in this subspace. A simplified example uses a road as one-dimensional manifolds in a three-dimensional space and identifies data points using house numbers as local coordinates.

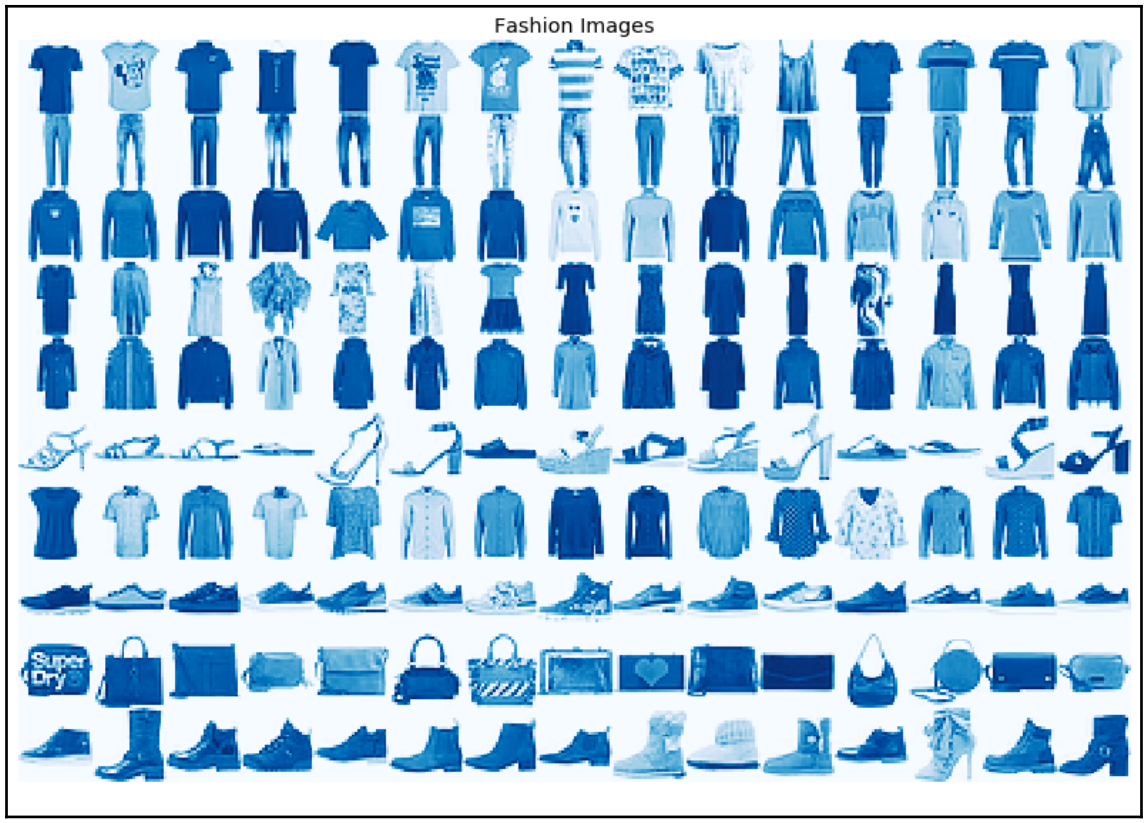
Several techniques approximate a lower dimensional manifold. One example is **locally-linear embedding** (**LLE**), which was developed in 2000 by Sam Roweis and Lawrence Saul and used to unroll the swiss roll in the previous screenshot (see examples in the `manifold_learning_lle` notebook).

For each data point, LLE identifies a given number of nearest neighbors and computes weights that represent each point as a linear combination of its neighbors. It finds a lower-dimensional embedding by linearly projecting each neighborhood onto global internal coordinates on the lower-dimensional manifold, and can be thought of as a sequence of PCA applications.

Visualization requires the reduction to at least three dimensions, possibly below the intrinsic dimensionality, and poses the challenge of faithfully representing local and global structure. This challenge relates to the increasing distance associated with the curse of dimensionality. While the volume of a sphere expands exponentially with the number of dimensions, the space in lower dimensions available to represent high-dimensional data is much more limited.

For example, in 12 dimensions, there can be 13 equidistant points, but in two dimensions there can only be three that form a triangle with sides of equal length. Hence, accurately reflecting the distance of one point to its high-dimensional neighbors in lower dimensions risks distorting the relations among all other points. The result is the crowding problem: to maintain global distances, local points may need to be placed too closely together, and vice versa.

The following two sections cover techniques that have made progress in addressing the crowding problem for the visualization of complex datasets. We will use the fashion `MNIST` dataset, a more sophisticated alternative to the classic handwritten digit MNIST benchmark data used for computer vision. It contains 60,000 train and 10,000 test images of fashion objects in 10 classes (see following samples):



The goal of a manifold learning algorithm for this data is to detect whether the classes lie on distinct manifolds, to facilitate their recognition and differentiation.

t-SNE

The t-distributed stochastic neighbor embedding is an award-winning algorithm developed in 2010 by Laurens van der Maaten and Geoff Hinton to detect patterns in high-dimensional data. It takes a probabilistic, non-linear approach to locating data on several different but related low-dimensional manifolds.

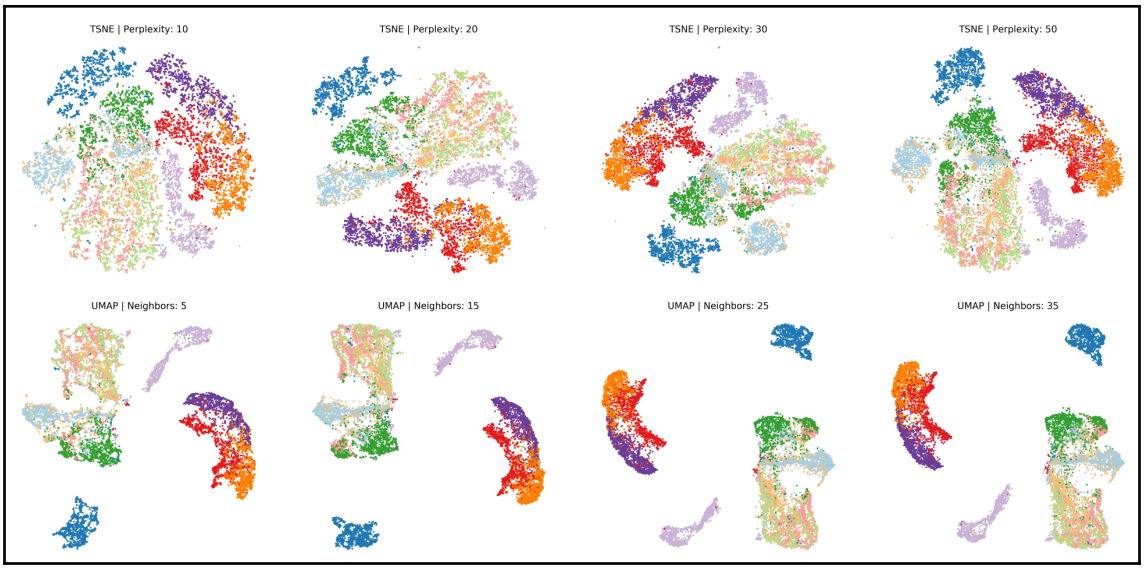
The algorithm emphasizes keeping similar points together in low dimensions, as opposed to maintaining the distance between points that are apart in high dimensions, which results from algorithms such as PCA that minimize squared distances.

The algorithm proceeds by converting high-dimensional distances to (conditional) probabilities, where high probabilities imply low distance and reflect the likelihood of sampling two points based on similarity. It accomplishes this by positioning a normal distribution over each point and computing the density for a point and each neighbor, where the perplexity parameter controls the effective number of neighbors.

In a second step, it arranges points in low dimensions and uses similarly computed low-dimensional probabilities to match the high-dimensional distribution. It measures the difference between the distributions using the Kullback-Leibler divergence, which puts a high penalty on misplacing similar points in low dimensions.

The low-dimensional probabilities use a Student's t-distribution with one degree of freedom, as it has fatter tails that reduce the penalty of misplacing points that are more distant in high dimensions, to manage the crowding problem.

The upper panels of the following chart show how t-SNE is able to differentiate between the image classes. A higher perplexity value increases the number of neighbors used to compute local structure, and gradually results in more emphasis on global relationships:



t-SNE is currently the state-of-the-art in high-dimensional data visualization. Weaknesses include the computational complexity that scales quadratically in the number n of points because it evaluates all pairwise distances, but a subsequent tree-based implementation has reduced the cost to $n \log n$.

t-SNE does not facilitate the projection of new data points into the low-dimensional space. The compressed output is not a very useful input for distance-based or density-based cluster algorithms, because t-SNE treats small and large distances differently.

UMAP

Uniform Manifold Approximation and Projection is a more recent algorithm for visualization and general dimensionality reduction. It assumes the data is uniformly distributed on a locally-connected manifold and looks for the closest low-dimensional equivalent using fuzzy topology. It uses a neighbors parameter that impacts the result similarly as perplexity above.

It is faster, and hence scales better to large datasets than t-SNE, and sometimes preserves global structure than better than t-SNE. It can also work with different distance functions, including, for example, cosine similarity, which is used to measure the distance between word count vectors.

The four charts in the bottom row of the previous figure illustrates how UMAP does indeed move the different clusters further apart, whereas t-SNE provides more granular insight into the local structure.

The notebook also contains interactive Plotly visualizations for each algorithm, which permit the exploration of the labels and identify which objects are placed close to each other.

Clustering

Both clustering and dimensionality reduction summarize the data. As just discussed in detail, dimensionality reduction compresses the data by representing it using new, fewer features that capture the most relevant information. Clustering algorithms, by contrast, assign existing observations to subgroups that consist of similar data points.

Clustering can serve to better understand the data through the lens of categories learned from continuous variables. It also permits automatically categorizing new objects according to the learned criteria. Examples of related applications include hierarchical taxonomies, medical diagnostics, and customer segmentation.

Alternatively, clusters can be used to represent groups as prototypes, using (for example) the midpoint of a cluster as the best representative of learned grouping. An example application includes image compression.

Clustering algorithms differ with respect to their strategies for identifying groupings:

- Combinatorial algorithms select the most coherent of different groupings of observations
- Probabilistic modeling estimates distributions that most likely generated the clusters
- Hierarchical clustering finds a sequence of nested clusters that optimizes coherence at any given stage

Algorithms also differ in their notion of what constitutes a useful collection of objects, which needs to match the data characteristics, domain, and the goal of the applications. Types of groupings include the following:

- Clearly separated groups of various shapes
- Prototype-based or center-based compact clusters
- Density-based clusters of arbitrary shape
- Connectivity-based or graph-based clusters

Important additional aspects of a clustering algorithm include the following:

- Whether it requires exclusive cluster membership
- Whether it makes hard (binary) or soft (probabilistic) assignment
- Whether it is complete and assigns all data points to clusters

The following sections introduce key algorithms, including k-Means, hierarchical, and density-based clustering, as well as Gaussian mixture models. The `clustering_algos` notebook compares the performance of these algorithms on different, labeled datasets to highlight their strengths and weaknesses. It uses mutual information (see `Chapter 6`, *The Machine Learning Process*) to measure the congruence of cluster assignments and labels.

k-Means clustering

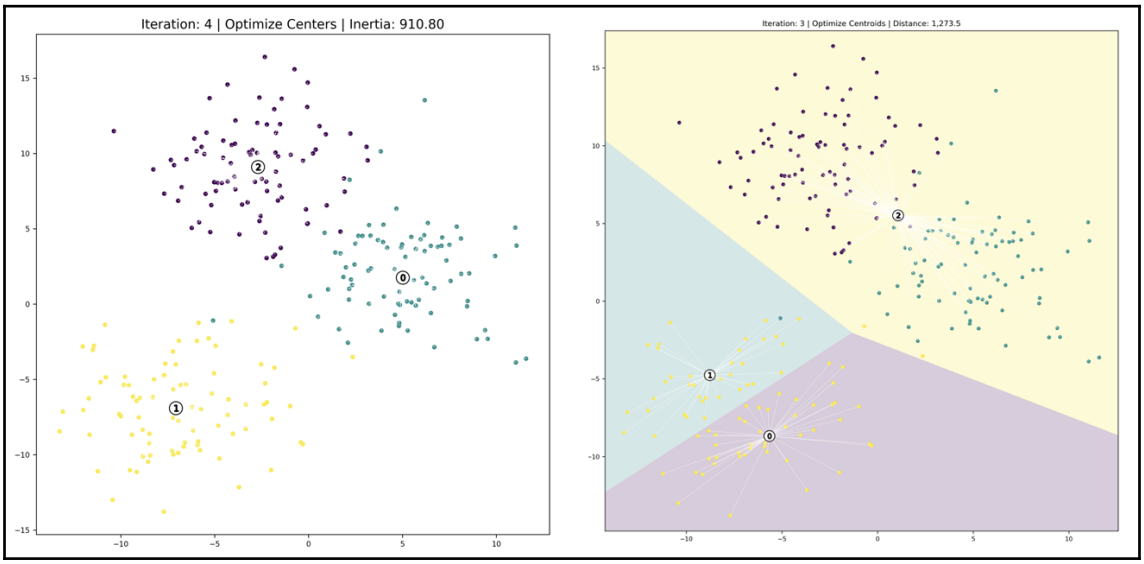
k-Means is the most well-known clustering algorithm and was first proposed by Stuart Lloyd at Bell Labs in 1957.

The algorithm finds *K* centroids and assigns each data point to exactly one cluster with the goal of minimizing the within-cluster variance (called inertia). It typically uses Euclidean distance, but other metrics can also be used. k-Means assumes that clusters are spherical and of equal size, and ignores the covariance among features.

The problem is computationally difficult (np-hard) because there are K^N ways to partition the *N* observations into *K* clusters. The standard iterative algorithm delivers a local optimum for a given *K* and proceeds as follows:

1. Randomly define *K* cluster centers and assign points to nearest centroid.
2. Repeat as follows:
    - For each cluster, compute the centroid as the average of the features
    - Assign each observation to the closest centroid
3. Convergence: assignments (or within-cluster variation) don't change.

The `kmeans_implementation` notebook shows how to code the algorithm using Python, and visualizes the algorithm's iterative optimization. The following screenshot highlights how the resulting centroids partition the feature space into areas called **Voronoi** which delineate the clusters:



The result is optimal for the given initialization, but alternative starting positions will produce different results. Hence, we compute multiple clusterings from different initial values and select the solution that minimizes within-cluster variance.

k-Means requires continuous or one-hot encoded categorical variables. Distance metrics are typically sensitive to scale so that standardizing features is necessary to make sure they have equal weight.

The strengths of k-Means include its wide range of applicability, fast convergence, and linear scalability to large data while producing clusters of even size.

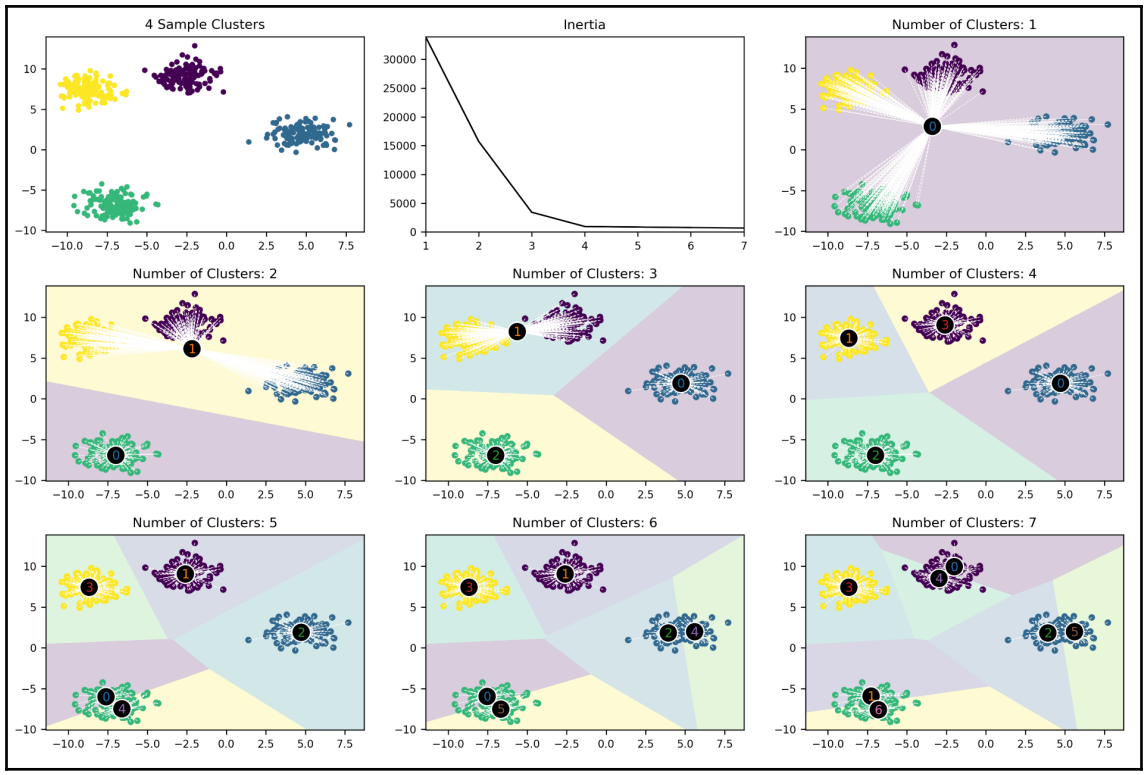
The weaknesses include:

- The need to tune the hyperparameter *k*
- The lack of a guarantee to find a global optimum
- Restrictive assumption that clusters are spheres and features are not correlated
- Sensitivity to outliers

Evaluating cluster quality

Cluster quality metrics help select among alternative clustering results.
The `kmeans_evaluation` notebook illustrates the following options:

1. The k-Means objective function suggests we compare the evolution of the inertia or within-cluster variance.
2. Initially, additional centroids decrease the inertia sharply because new clusters improve the overall fit.
3. Once an appropriate number of clusters has been found (assuming it exists), new centroids reduce the within-cluster variance by much less as they tend to split natural groupings.
4. Hence, when k-Means finds a good cluster representation of the data, the inertia tends to follow an elbow-shaped path similar to the explained variance ratio for PCA, as shown in the following screenshot (see notebook for implementation details):



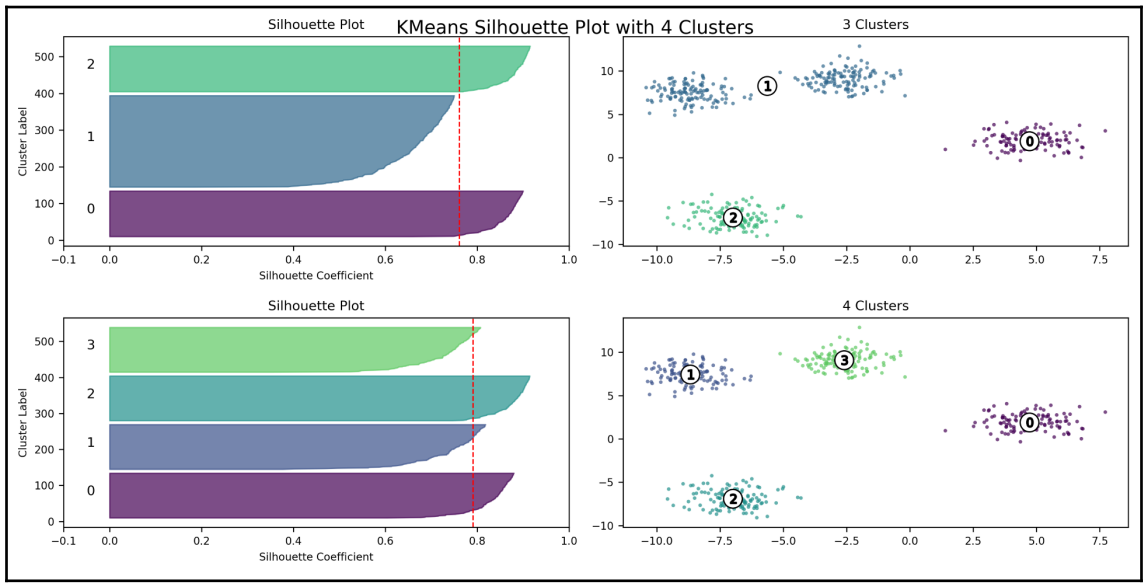
The silhouette coefficient provides a more detailed picture of cluster quality. It answers the question: how far are the points in the nearest cluster, relative to the points in the assigned cluster?

To this end, it compares the mean intra-cluster distance (a) to the mean distance of the nearest cluster (b) and computes the following score s:

$$s = \frac{b - a}{max(a, b)} \in [-1, 1]$$

The score can vary from between *-1* and *1*, but negative values are unlikely in practice because they imply that the majority of points are assigned to the wrong cluster. A useful visualization of the silhouette score compares the values for each data point to the global average because it highlights the coherence of each cluster relative to the global configuration. The rule of thumb is to avoid clusters with mean scores below the average for all samples.

The following screenshot shows an excerpt from the silhouette plot for three and four clusters, where the former highlights the poor fit of cluster *1* by sub-par contributions to the global silhouette score, whereas all of the four clusters have some values that exhibit above average scores:



In sum, given the usually unsupervised nature, it is necessary to vary the hyperparameters of the cluster algorithms and evaluate the different results. It is also important to calibrate the scale of the features, in particular when some should be given a higher weight and should thus be measured on a larger scale.

Finally, to validate the robustness of the results, use subsets of data to identify whether particular patterns emerge consistently.

Hierarchical clustering

Hierarchical clustering avoids the need to specify a target number of clusters because it assumes that data can successively be merged into increasingly dissimilar clusters. It does not pursue a global objective but decides incrementally how to produce a sequence of nested clusters that range from a single cluster to clusters consisting of the individual data points.

There are two approaches:

1. **Agglomerative clustering** proceeds bottom-up, sequentially merging two of the remaining groups based on similarity
2. **Divisive clustering** works top-down and sequentially splits the remaining clusters to produce the most distinct subgroups

Both groups produce *N*-1 hierarchical levels and facilitate the selection of a clustering at the level that best partitions data into homogenous groups. We will focus on the more common agglomerative clustering approach.

The agglomerative clustering algorithm departs from the individual data points and computes a similarity matrix containing all mutual distances. It then takes *N*-1 steps until there are no more distinct clusters, and each time updates the similarity matrix to substitute elements that have been merged by the new cluster so that the matrix progressively shrinks.

While hierarchical clustering does not have hyperparameters like k-Means, the measure of dissimilarity between clusters (as opposed to individual data points) has an important impact on the clustering result. The options differ as follows:

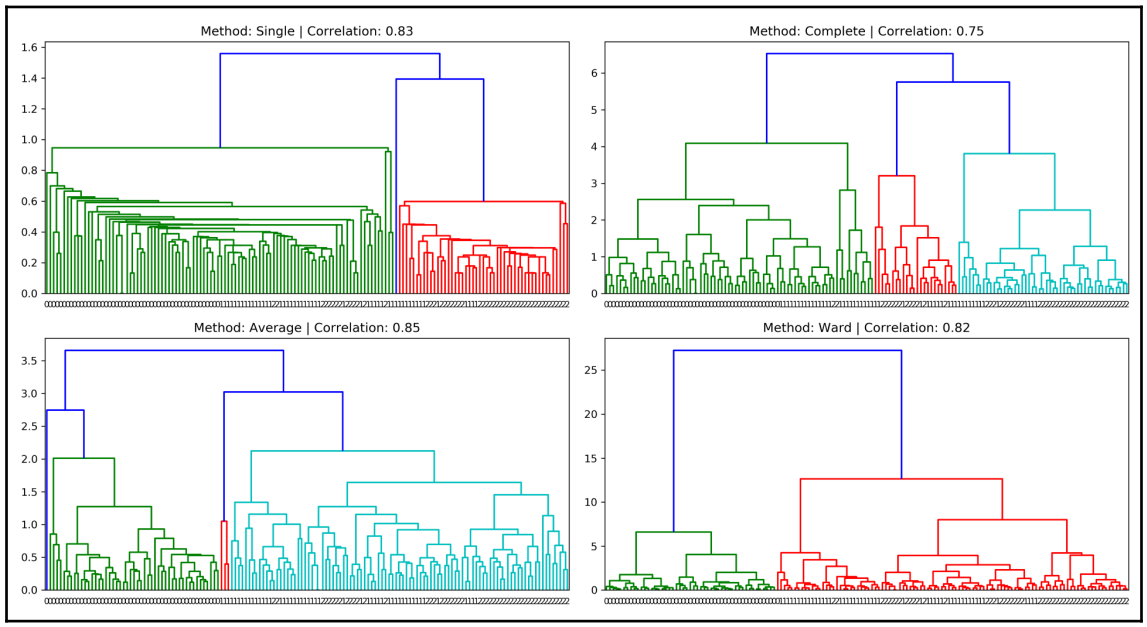
- **Single-link**: the distance between nearest neighbors of two clusters
- **Complete link**: the maximum distance between respective cluster members
- **Group average:** the distance between averages for each group
- **Ward's method**: minimizes within-cluster variance

Visualization – dendrograms

Hierarchical clustering provides insight into degrees of similarity among observations as it continues to merge data. A significant change in the similarity metric from one merge to the next suggests a natural clustering existed prior to this point.

The dendrogram visualizes the successive merges as a binary tree, displaying the individual data points as leaves and the final merge as the root of the tree. It also shows how the similarity monotonically decreases from bottom to top. Hence, it is natural to select a clustering by cutting the dendrogram.

The following screenshot (see the `hierarchical_clustering` notebook for implementation details) illustrates the dendrogram for the classic Iris dataset with four classes and three features, using the four different distance metrics introduced precedingly:



It evaluates the fit of the hierarchical clustering using the cophenetic correlation coefficient, which compares the pairwise distances among points and the cluster similarity metric at which a pairwise merge occurred. A coefficient of 1 implies that closer points always merge earlier.

Different linkage methods produce dendrograms of different appearance, so we cannot use this visualization to compare results across methods. In addition, Ward's method, which minimizes within-cluster variance, may not properly reflect the change in variance, but rather the total variance, which may be misleading. Instead, other quality metrics such as cophenetic correlation, or measures such as inertia (if aligned with the overall goal), may be more appropriate.

The strengths of clustering include:

- You do not need to specify the number of clusters
- It offers insight about potential clustering by means of an intuitive visualization
- It produces a hierarchy of clusters that can serve as taxonomy
- It can be combined with k-Means to reduce the number of items at the start of the agglomerative process

Weaknesses of hierarchical clustering include:

- The high cost in terms of computation and memory because of the numerous similarity matrix updates
- It does not achieve the global optimum because all merges are final
- The curse of dimensionality leads to difficulties with noisy, high-dimensional data

Density-based clustering

Density-based clustering algorithms assign cluster membership based on proximity to other cluster members. They pursue the goal of identifying dense regions of arbitrary shapes and sizes. They do not require the specification of a certain number of clusters but instead rely on parameters that define the size of a neighborhood and a density threshold (see the `density_based_clustering` notebook for relevant code samples).

DBSCAN

Density-based spatial clustering of applications with noise (**DBSCAN**) was developed in 1996, and received the *Test of Time* award at the 2014 KDD conference because of the attention it has received in both theory and practice.

It aims to identify core and non-core samples, where the former extend a cluster and the latter are part of a cluster but do not have sufficient nearby neighbors to further grow the cluster. Other samples are outliers and not assigned to any cluster.

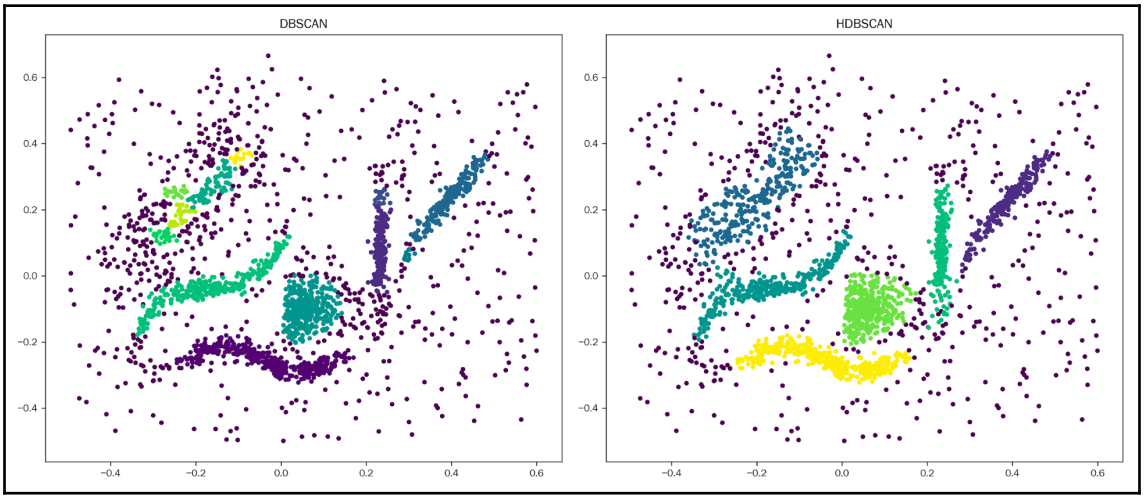
It uses an `eps` parameter for the radius of the neighborhood and `min_samples` for the number of members required for core samples. It is deterministic and exclusive and has difficulties with clusters of different density and high-dimensional data. It can be challenging to tune the parameters to the requisite density, especially as it is often not constant.

Hierarchical DBSCAN

Hierarchical DBSCAN is a more recent development that assumes clusters are islands of potentially differing density, to overcome the DBSCAN challenges just mentioned. It also aims to identify the core and non-core samples. It uses the `min_cluster_ size` and `min_samples` parameters to select a neighborhood and extend a cluster. The algorithm iterates over multiple `eps` values and chooses the most stable clustering.

In addition to identifying clusters of varying density, it provides insight into the density and hierarchical structure of the data.

The following screenshots show how DBSCAN and HDBSCAN are able to identify very differently shaped clusters:



Gaussian mixture models

A **Gaussian mixture model** (**GMM**) is a generative model that assumes the data has been generated by a mix of various multivariate normal distributions. The algorithm aims to estimate the mean and covariance matrices of these distributions.

It generalizes the k-Means algorithm: it adds covariance among features so that clusters can be ellipsoids rather than spheres, while the centroids are represented by the means of each distribution. The GMM algorithm performs soft assignments because each point has the probability to be a member of any cluster.

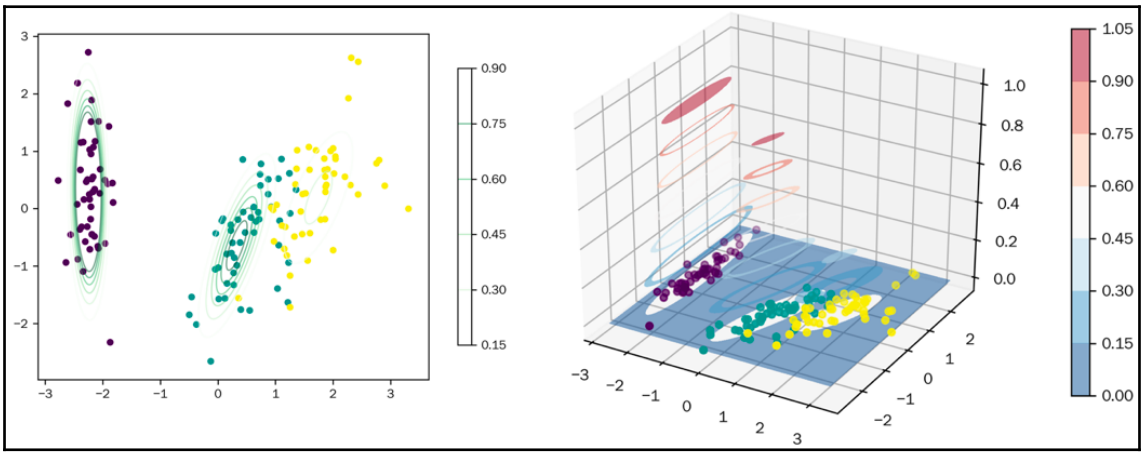
The expectation-maximization algorithm

GMM uses the expectation-maximization algorithm to identify the components of the mixture of Gaussian distributions. The goal is to learn the probability distribution parameters from unlabeled data.

The algorithm proceeds iteratively as follows:

1. Initialization—Assume random centroids (for example, using k-Means)
2. Repeat the following steps until convergence (that is, changes in assignments drop below the threshold):
    - **Expectation step**: Soft assignment—compute probabilities for each point from each distribution
    - **Maximization step**: Adjust normal-distribution parameters to make data points most likely

The following screenshot shows the GMM cluster membership probabilities for the Iris dataset as contour lines:



Hierarchical risk parity

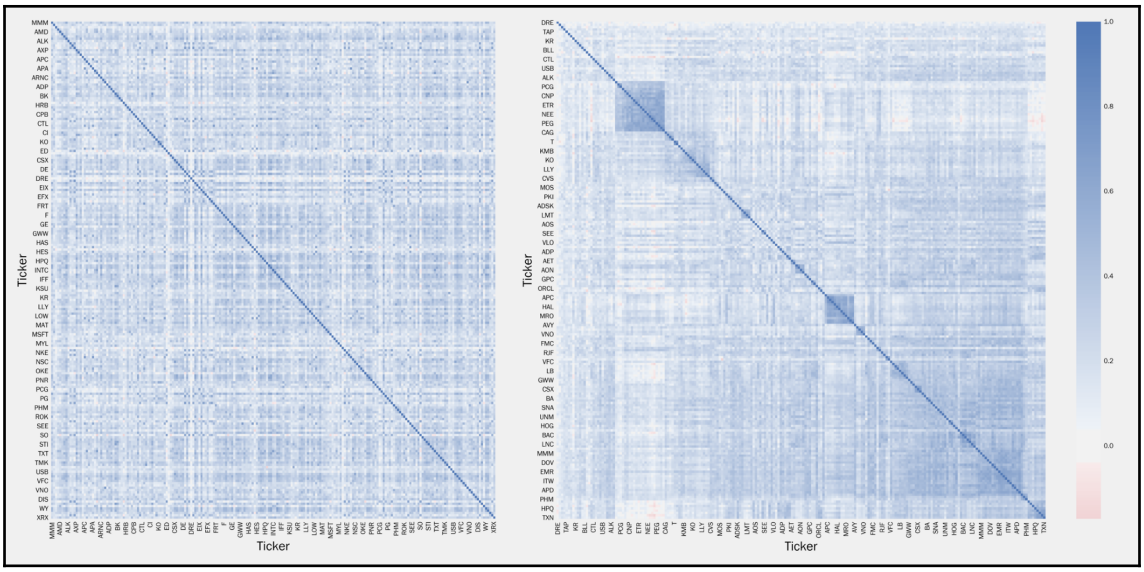
The key idea of hierarchical risk parity is to use hierarchical clustering on the covariance matrix in order to be able to group assets with similar correlations together, and reduce the number of degrees of freedom by only considering similar assets as substitutes when constructing the portfolio (see notebook and Python files in the `hierarchical_risk_parity` subfolder for details).

The first step is to compute a distance matrix that represents proximity for correlated assets and meets distance metric requirements. The resulting matrix becomes an input to the SciPy hierarchical clustering function which computes the successive clusters using one of several available methods discussed so far:

```
def get_distance_matrix(corr):
"""Compute distance matrix from correlation;
0 <= d[i,j] <= 1"""
return np.sqrt((1 - corr) / 2)
distance_matrix = get_distance_matrix(corr)
linkage_matrix = linkage(squareform(distance_matrix), 'single')
```

The `linkage_matrix` can be used as input to the `seaborn.clustermap` function to visualize the resulting hierarchical clustering. The dendrogram displayed by `seaborn` shows how individual assets and clusters of assets are merged based on their relative distances:

```
clustergrid = sns.clustermap(distance_matrix,
method='single',
row_linkage=linkage_matrix,
col_linkage=linkage_matrix,
cmap=cmap, center=0)
sorted_idx = clustergrid.dendrogram_row.reordered_ind
sorted_tickers = corr.index[sorted_idx].tolist()
```



Heatmap

Compared to a `seaborn.heatmap` of the original correlation matrix, there is now significantly more structure in the sorted data (right panel).

Using the tickers sorted according to the hierarchy induced by the clustering algorithm, HRP now proceeds to compute a top-down inverse-variance allocation that successively adjusts weights depending on the variance of the subclusters further down the tree:

```
def get_cluster_var(cov, cluster_items):
    """Compute variance per cluster"""
    cov_ = cov.loc[cluster_items, cluster_items]  # matrix slice
    w_ = get_inverse_var_pf(cov_)
    return (w_ @ cov_ @ w_).item()
```

To this end, the algorithm uses bisectional search to allocate the variance of a cluster to its elements based on their relative riskiness:

```
def get_hrp_allocation(cov, tickers):
    """Compute top-down HRP weights"""

    weights = pd.Series(1, index=tickers)
    clusters = [tickers]  # initialize one cluster with all assets

    while len(clusters) > 0:
        # run bisectional search:
        clusters = [c[start:stop] for c in clusters
```

```
                    for start, stop in ((0, int(len(c) / 2)),
                                        (int(len(c) / 2), len(c)))
                    if len(c) > 1]
        for i in range(0, len(clusters), 2):  # parse in pairs
            cluster0 = clusters[i]
            cluster1 = clusters[i + 1]

            cluster0_var = get_cluster_var(cov, cluster0)
            cluster1_var = get_cluster_var(cov, cluster1)

            weight_scaler = 1 - cluster0_var / (cluster0_var +
cluster1_var)
            weights[cluster0] *= weight_scaler
            weights[cluster1] *= 1 - weight_scaler
    return weights
```

The resulting portfolio allocation produces weights that sum to `1` and reflect the structure present in the correlation matrix (see notebook for details).

Summary

In this chapter, we explored unsupervised learning methods that allow us to extract valuable signal from our data, without relying on the help of outcome information provided by labels.

We saw how we can use linear dimensionality reduction methods, such as PCA and ICA, to extract uncorrelated or independent components from the data that can serve as risk factors or portfolio weights. We also covered advanced non-linear manifold learning techniques that produce state-of-the-art visualizations of complex alternative datasets.

In the second part, we covered several clustering methods that produce data-driven groupings under various assumptions. These groupings can be useful, for example, to construct portfolios that apply risk-parity principles to assets that have been clustered hierarchically.

In the next three chapters, we will learn about various ML techniques for a key source of alternative data, namely, natural language processing for text documents.