13
Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning

Chapter 6, *The Machine Learning Process*, introduced how unsupervised learning adds value by uncovering structures in data without the need for an outcome variable to guide the search process. This contrasts with supervised learning, which was the focus of the last several chapters: instead of predicting future outcomes, unsupervised learning aims to learn an informative representation of the data that helps explore new data, discover useful insights, or solve some other task more effectively.

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. Algorithms differ by how they measure the loss of information, whether they apply linear or nonlinear transformations or which 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.

These unsupervised algorithms are useful when a **dataset does not contain an outcome**. For instance, we may want to extract tradeable information from a large body of financial reports or news articles. In *Chapter 14*, *Text Data for Trading – Sentiment Analysis*, we'll use topic modeling to discover hidden themes that allow us to explore and summarize content more effectively, and identify meaningful relationships that can help us to derive signals.

The algorithms are also useful when we want to **extract information independently from an outcome**. For example, rather than using third-party industry classifications, clustering allows us to identify synthetic groupings based on the attributes of assets useful for our purposes, such as returns over a certain time horizon, exposure to risk factors, or similar fundamentals. In this chapter, we will learn how to use clustering to manage portfolio risks by identifying hierarchical relationships among asset returns.

More specifically, after reading this chapter, you will understand:

- How **principal component analysis** (**PCA**) and **independent component analysis** (**ICA**) perform linear dimensionality reduction
- Identifying data-driven risk factors and eigenportfolios from asset returns using PCA
- Effectively visualizing nonlinear, high-dimensional data using manifold learning
- Using T-SNE and UMAP to explore high-dimensional image data
- How k-means, hierarchical, and density-based clustering algorithms work
- Using agglomerative clustering to build robust portfolios with hierarchical risk parity

> You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Dimensionality reduction

In linear algebra terms, the features of a dataset create a **vector space** whose dimensionality corresponds to the number of linearly independent rows or columns, whichever is larger. 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 direction rather than different ones in the data and thus 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 and does not add a new dimension of its own.

The number of dimensions of a dataset matters 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, with dramatic implications for **machine learning** (**ML**). **Prediction becomes much harder** when observations are more distant, that is, different from each other. Alternative data sources, like text and images, typically are of high dimensionality, but they generally affect models that rely on a large number of features. The next section addresses the resulting challenges.

Dimensionality reduction seeks to **represent the data more efficiently** by using fewer features. To this end, algorithms project the data to a

lower-dimensional space while discarding any variation that is not informative, or by identifying a lower-dimensional subspace or manifold on or near to where the data lives.

A **manifold** is a space that locally resembles Euclidean space. One-dimensional manifolds include a line or a circle, but not the visual representation of the number eight due to the crossing point.

The manifold hypothesis maintains that high-dimensional data often resides in a lower-dimensional space, which, if identified, permits a faithful representation of the data in this subspace. Refer to Fefferman, Mitter, and Narayanan (2016) for background information and the description of an algorithm that tests this hypothesis.

Dimensionality reduction, therefore, 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.

Dimensionality reduction algorithms differ by the constraints they impose on the new variables and how they aim to minimize the loss of information (see Burges 2010 for an excellent overview):

- **Linear algorithms** like PCA and ICA constrain the new variables to be linear combinations of the original features; for example, hyperplanes in a lower-dimensional space. Whereas PCA requires the new features to be uncorrelated, ICA goes further and imposes statistical independence, implying the absence of both linear and nonlinear relationships.
- **Nonlinear algorithms** are not restricted to hyperplanes and can capture a more complex structure in the data. However, given the infinite number of options, the algorithms still need to make assumptions in order to arrive at a solution. Later in this section, we will explain how **t-distributed Stochastic Neighbor Embedding (t-SNE)** and **Uniform Manifold Approximation and Projection (UMAP)** are very useful to visualize higher-dimensional data. *Figure 13.1* illustrates how manifold learning identifies a two-dimensional subspace in the three-dimensional feature space. (The notebook `manifold_learning` illustrates the use of additional algorithms, including local linear embedding.)

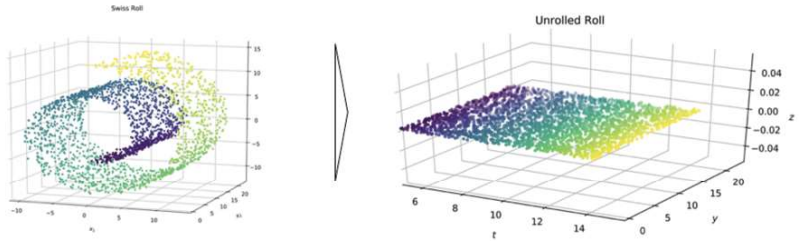

Figure 13.1: Nonlinear dimensionality reduction

The curse of dimensionality

An increase in the number of dimensions of a dataset means that 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 the Euclidean distance, also known as the L^2 norm, which we applied to the vector of linear regression coefficients to train a regularized ridge regression.

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:

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

Therefore, 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 the data points the same.

Figure 13.2 illustrates the exponential growth in the number of data points needed to maintain the average distance among observations as the number of dimensions increases. 10 points uniformly distributed on a line correspond to 10^2 points in two dimensions and 10^3 points in three dimensions in order to keep the density constant.

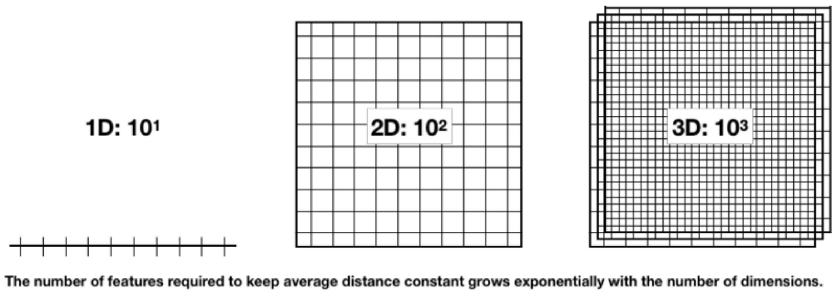

Figure 13.2: The number of features required to keep the average distance constant grows exponentially with the number of dimensions

The notebook `the_curse_of_dimensionality` in the GitHub repository folder for this section simulates how the average and minimum distances between data points increase as the number of dimensions grows (see *Figure 13.3*).

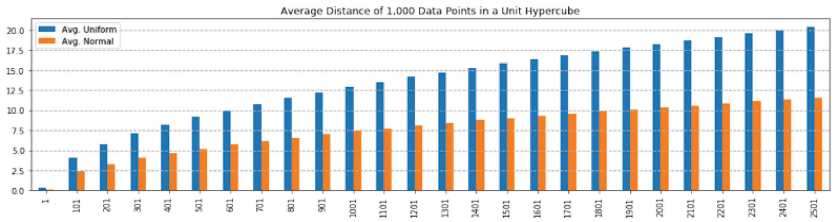

Figure 13.3: Average distance of 1,000 data points in a unit hypercube

The **simulation** randomly samples up to 2,500 features in the range [0, 1] from an uncorrelated uniform or a correlated normal distribution. The average distance between data points increases to over 11 times the unitary feature range for the normal distribution, and to over 20 times in the (extreme) case of an uncorrelated uniform distribution.

When the **distance between observations** grows, supervised ML becomes more difficult because predictions for new samples are less likely to be based on learning from similar training features. Put simply, the number of possible unique rows grows exponentially as the number of features increases, making it 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 11*, *Random Forests – A Long-Short Strategy for Japanese Stocks*, and *Chapter 12*, *Boosting Your Trading Strategy*. They also include the deep neural networks that we will cover later in the book, starting with *Chapter 16*, *Word Embeddings for Earnings Calls and SEC Filings*. The variance of these algorithms increases as **more dimensions add opportunities to overfit** to noise, resulting in poor generalization performance.

Dimensionality reduction leverages the fact that, in practice, features are often correlated or exhibit little variation. If so, it can compress data without losing much of the signal and 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?

Linear dimensionality reduction

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

PCA, invented in 1901 by Karl Pearson, finds new features that reflect directions of maximal variance in the data while being mutually uncorrelated. 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 in order to learn risk factors from the data, and build so-called eigenportfolios for systematic trading strategies.

Principal component analysis

PCA finds linear combinations of the existing features and uses these principal components to represent the original data. The number of components is a hyperparameter that determines the target dimensionality and can be, at most, equal to the lesser of the number of rows or columns.

PCA aims to capture most of the variance in the data to make it easy to recover the original features and ensures 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 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 ensures that new components are not correlated with existing components and produces an orthogonal basis for a vector space.

This new basis is a rotation of the original basis, such that the new axes 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. In other words, the share of components that captures, for example, 95 percent of the original variation provides insight into the linearly independent information in the original data.

Visualizing PCA in 2D

Figure 13.4 illustrates several aspects of PCA for a two-dimensional random dataset (refer to the notebook `pca_key_ideas`):

- 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.
- The right panel illustrates **supervised OLS** (refer to *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*), which approximates the outcome (x_2) by a line computed from the single feature x_1. The vertical lines highlight how OLS minimizes the distance along the outcome axis, whereas PCA minimizes the distances that are orthogonal to the hyperplane.

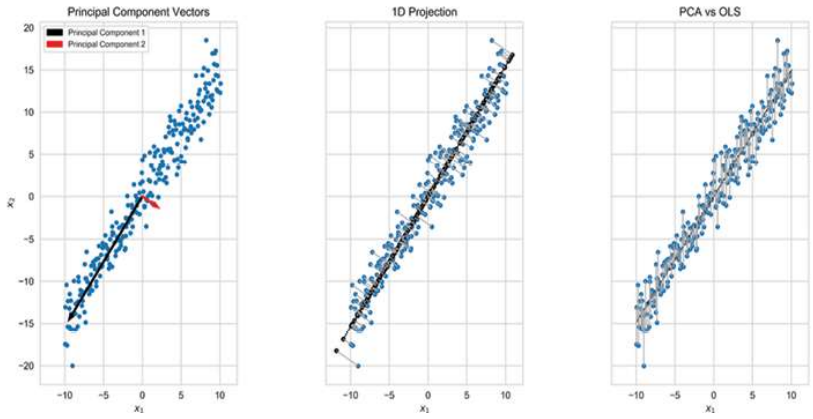

Figure 13.4: PCA in 2D from various perspectives

Key assumptions made by PCA

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

- 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 moments 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. Market data often exhibits skew or kurtosis (fat tails) that differ from those of the normal distribution and will not be taken into account by PCA.

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 previously, 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 by using the **singular value decomposition** (**SVD**).

We illustrate the computation using a randomly generated three-dimensional ellipse with 100 data points, as shown in the left panel of *Figure 13.5*, including the two-dimensional hyperplane defined by the first two principal components. (Refer to the notebook `the_math_behind_pca` for the code samples in the following three sections.)

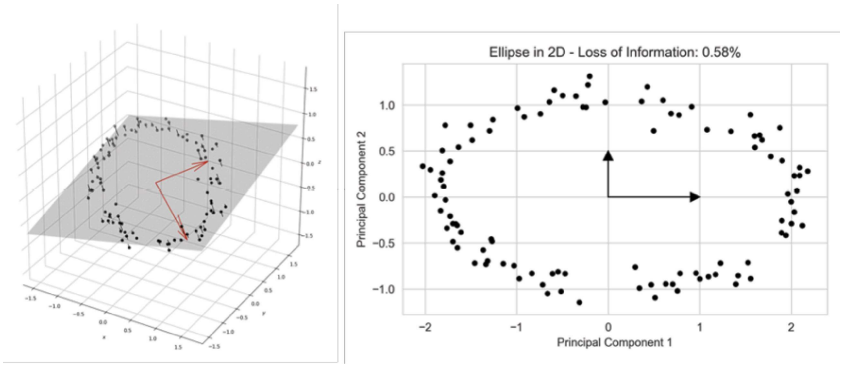

Figure 13.5: Visual representation of dimensionality reduction from 3D to 2D

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, $i, j = 1, \ldots, n$ as entries in row i and column j:

$$cov_{i,j} = \frac{\sum_{k=1}^{N}(x_{ik} - \bar{x}_i)\left(x_{jk} - \bar{x}_j\right)}{N-1}$$

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

$$Mw_i = \lambda_i w_i$$

Therefore, 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 λ_i as diagonal entries (and 0s otherwise). We define the **eigendecomposition** as:

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

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

```
# compute covariance matrix:
cov = np.cov(data.T) # 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 the singular value decomposition

Next, we'll take a look at the alternative computation using the SVD. This algorithm is slower when the number of observations is greater than the number of features (which is the typical case) but yields better **numerical stability**, especially when some of the features are strongly correlated (which is 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 figure. The diagonal values of Σ are the singular values, and the transpose of V^* contains the principal components as column vectors.

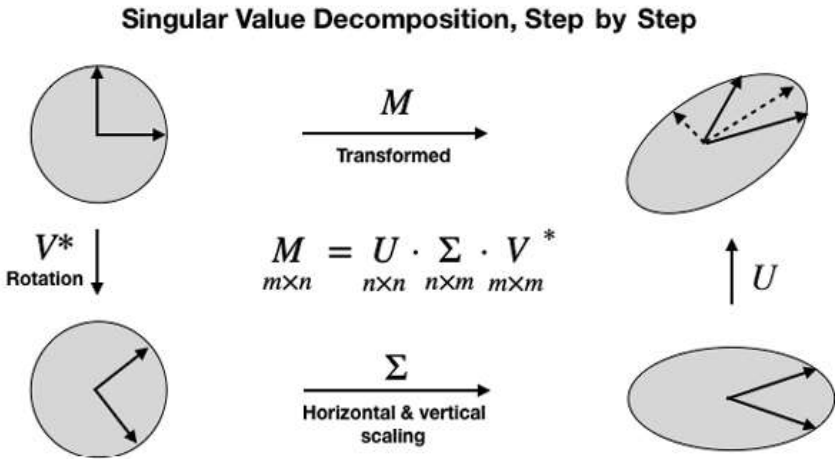

Figure 13.6: The SVD decomposed

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

```
n_features = data.shape[1]
data_ = data - data.mean(axis=0)
```

Using the centered data, we compute the SVD:

```
U, s, Vt = svd(data_)
U.shape, s.shape, Vt.shape
((100, 100), (3,), (3, 3))
```

We can convert the vector `s`, which contains only singular values, into an *n* x *m* 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))
```

In the next section, we will demonstrate how sklearn implements PCA.

PCA with sklearn

The `sklearn.decomposition.PCA` implementation follows the standard API based on the `fit()` and `transform()` methods that 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 percent of the components.
- **auto** also randomizes where it is most efficient; otherwise, it uses the full SVD.

Please view the references on GitHub for algorithmic implementation details.

Other key configuration parameters of the PCA object are:

- **n_components**: 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 option `mle` estimates the number of dimensions using the maximum likelihood.
- **whiten**: If `True`, it standardizes the component vectors to unit variance, which, in some cases, can be useful 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()`:

```
pca2 = PCA(n_components=2)
projected_data  = pca2.fit_transform(data)
projected_data.shape
(100, 2)
```

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

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

Figure 13.5 shows the projection of the data into the new two-dimensional space.

Independent component analysis

ICA is another linear algorithm that identifies a new basis to represent the original data but pursues a different objective than PCA. Refer to Hyvärinen and Oja (2000) for a detailed introduction.

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, where a given number of guests are speaking at the same

time so that a single microphone records overlapping signals. ICA assumes there are as many different microphones as there are speakers, each placed at different locations so that they record a different mix of signals. ICA then aims to recover the individual signals from these 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

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

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, therefore, 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 earlier, 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 the independent aspects of the data.

Manifold learning – nonlinear dimensionality reduction

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 signals reflected in nonlinear 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 nonlinear manifold that is embedded in the higher-dimensional space. The two-dimensional Swiss roll displayed in *Figure 13.1* (at the beginning of this chapter) 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 a one-dimensional manifold 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 invented by Lawrence Saul and Sam Roweis (2000) and used to "unroll" the Swiss roll shown in *Figure 13.1* (view the 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 on global internal coordinates on the lower-dimensional manifold and can be thought of as a sequence of PCA applications.

Visualization requires that the reduction is at least three dimensions, possibly below the intrinsic dimensionality, and poses the **challenge of faithfully representing both the local and global structure**. This challenge

relates to the curse of dimensionality; that is, while the volume of a sphere expands exponentially with the number of dimensions, the lower-dimensional space available to represent high-dimensional data is much more limited. For instance, in 12 dimensions, there can be 13 equidistant points; however, in two dimensions, there can only be 3 that form a triangle with sides of equal length. Therefore, accurately reflecting the distance of one point to its high-dimensional neighbors in lower dimensions risks distorting the relationships among all other points. The result is the **crowding problem**: to maintain global distances, local points may need to be placed too closely together.

The next two sections cover techniques that have allowed us to make progress in addressing the crowding problem for the visualization of complex datasets. We will use the fashion MNIST dataset, which is a more sophisticated alternative to the classic handwritten digit MNIST benchmark data used for computer vision. It contains 60,000 training and 10,000 test images of fashion objects in 10 classes (take a look at the sample images in the notebook `manifold_learning_intro`). 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-distributed Stochastic Neighbor Embedding

t-SNE is an award-winning algorithm, developed by Laurens van der Maaten and Geoff Hinton in 2008, to detect patterns in high-dimensional data. It takes a probabilistic, nonlinear approach to locate 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 like PCA that minimize squared distances.

The algorithm proceeds by **converting high-dimensional distances into (conditional) probabilities**, where high probabilities imply low distance and reflect the likelihood of sampling two points based on similarity. It accomplishes this by, first, 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 the 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 because 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 in *Figure 13.7* show how t-SNE is able to differentiate between the FashionMNIST image classes. A higher perplexity value increases the number of neighbors used to compute the local structure and gradually results in more emphasis on global relationships. (Refer to the repository for a high-resolution color version of this figure.)

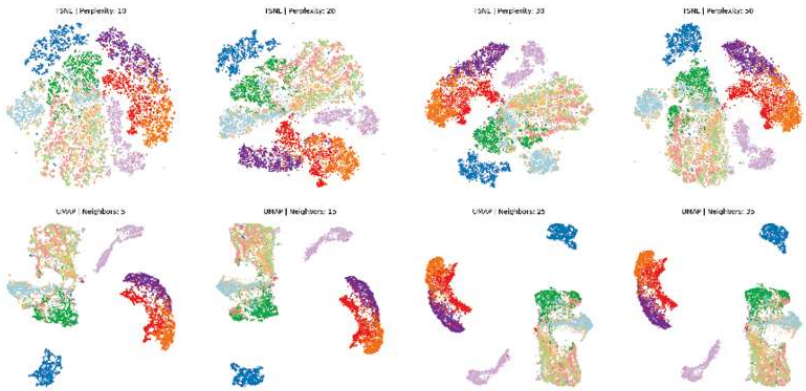

Figure 13.7: t-SNE and UMAP visualization of Fashion MNIST image data for different hyperparameters

t-SNE is the current 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 $n \log n$.

Unfortunately, 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- or density-based cluster algorithms because t-SNE treats small and large distances differently.

Uniform Manifold Approximation and Projection

UMAP 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, which impacts the result in a similar way to `perplexity` in the preceding section.

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

The preceding 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 of the algorithms that permit the exploration of the labels and identify which objects are placed close to each other.

PCA for trading

PCA is useful for algorithmic trading in several respects, including:

- The data-driven derivation of risk factors by applying PCA to asset returns
- The construction of uncorrelated portfolios based on the principal components of the correlation matrix of asset returns

We will illustrate both of these applications in this section.

Data-driven risk factors

In *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, 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 like PCA to simultaneously learn the factors from data and estimate how they drive returns. In this section, we will demonstrate 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 notebook `pca_and_risk_factor_models` for more details).

Preparing the data – top 350 US stocks

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 2010-2018 period. We will 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 percent and 97.5 percent quantiles, respectively:

PCA does not permit missing data, so we will remove any stocks that do not have data for at least 95 percent of the time period. Then, in a second step, we will remove trading days that do not have observations on at least 95 percent 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 315 equity return series covering a similar period:

```
returns.info()
DatetimeIndex: 2071 entries, 2010-01-05 to 2018-03-27
Columns: 315 entries, A to LYB
```

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))
```

Running PCA to identify the key return drivers

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

```
pca = PCA(n_components='mle')
pca.fit(returns)
```

We find that the most important factor explains around 55 percent 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 discussions in *Chapter 5*, *Portfolio Optimization and Performance Evaluation*, and *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, depending on the results of a closer inspection (please refer to the next example).

The plot on the right of *Figure 13.8* shows the cumulative explained variance and indicates that around 10 factors explain 60 percent of the returns of this cross-section of stocks.

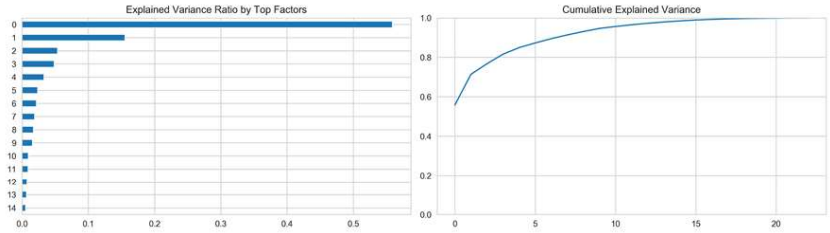

Figure 13.8: (Cumulative) explained return variance by PCA-based risk factors

The notebook contains a **simulation** for a broader cross-section of stocks and the longer 2000-2018 time period. It finds that, on average, the first three components explained 40 percent, 10 percent, and 5 percent of 500 randomly selected stocks, as shown in *Figure 13.9*:

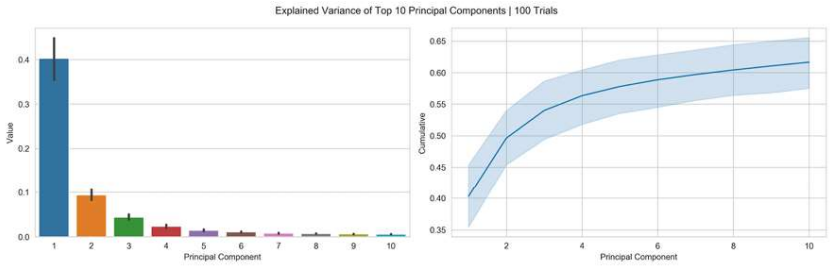

Figure 13.9: Explained variance of the top 10 principal components—100 trials

The cumulative plot shows a typical "elbow" pattern that can help to identify a suitable target dimensionality as the number of components beyond which additional components add less incremental 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, as shown in the following figure:

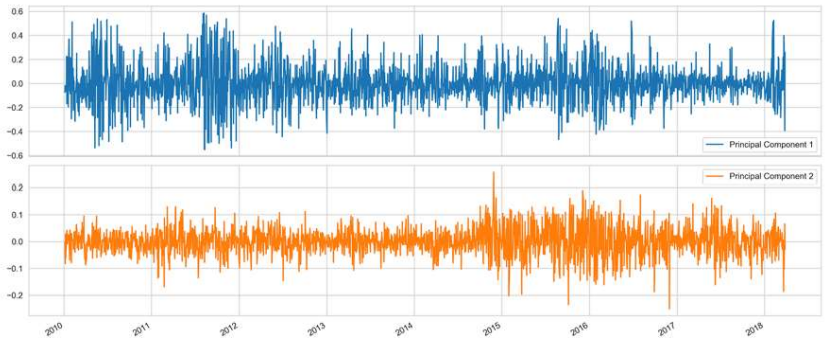

Figure 13.10: Return volatility patterns captured by the first two principal components

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 – From Risk Factors to Return Forecasts*.

Eigenportfolios

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. You can find the code example for this section in the notebook `pca_and_eigen_portfolios` .

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 like in the previous example, we are left with 23 assets and over 2,000 trading days. We compute the return covariance and estimate all of the principal components to find that the two largest explain 55.9 percent and 15.5 percent of the covariation, respectively:

```
cov = returns.cov()
pca = PCA()
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 EW portfolio formed from all of the 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, as you can see in *Figure 13.11*. 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:

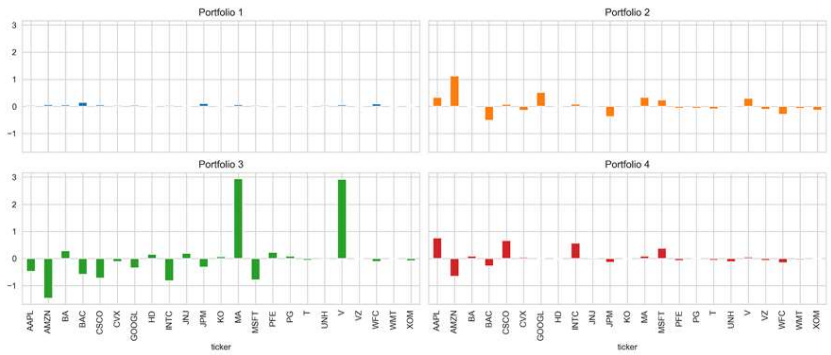

Figure 13.11: Eigenportfolio weights

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 (see *Figure 13.12*).

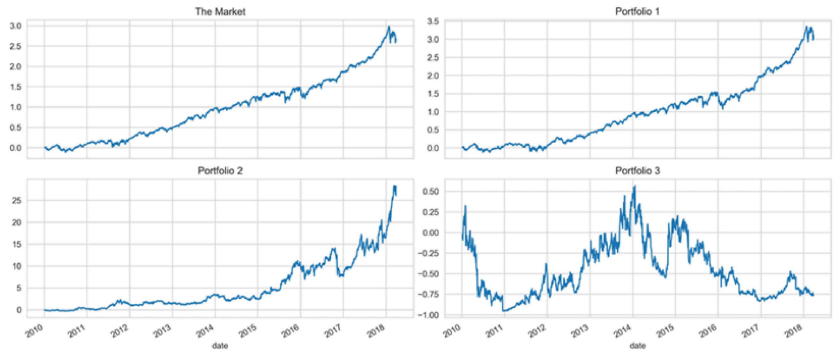

Figure 13.12: Cumulative eigenportfolio returns

Clustering

Both clustering and dimensionality reduction summarize the data. As we have just discussed, dimensionality reduction compresses the data by representing it using new, fewer features that capture the most relevant information. Clustering algorithms, in 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 you to automatically categorize 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 representatives of learned grouping. An example application includes image compression.

Clustering algorithms differ with respect to their strategy of 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 by the notion of what constitutes a useful collection of objects that needs to match the data characteristics, domain, and goal

of the applications. Types of groupings include:

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

Important additional aspects of a clustering algorithm include whether it:

- Requires exclusive cluster membership
- Makes hard, that is, binary, or soft, probabilistic assignments
- 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** (**GMMs**). The notebook `clustering_algos` compares the performance of these algorithms on different, labeled datasets to highlight strengths and weaknesses. It uses mutual information (refer to *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 it was first proposed by Stuart Lloyd at Bell Labs in 1957. It 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 the 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.

Assigning observations to clusters

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 the nearest centroid
2. Repeat:
    1. For each cluster, compute the centroid as the average of the features
    2. Assign each observation to the closest centroid
3. Convergence: assignments (or within-cluster variation) don't change

The notebook `kmeans_implementation` shows you how to code the algorithm using Python. It visualizes the algorithm's iterative optimization

and demonstrates how the resulting centroids partition the feature space into areas called Voronoi that delineate the clusters. The result is optimal for the given initialization, but alternative starting positions will produce different results. Therefore, 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, making it necessary to standardize features to ensure 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*, no guarantee of finding a global optimum, the restrictive assumption that clusters are spheres, and features not being correlated. It is also sensitive to outliers.

Evaluating cluster quality

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

The **k-means objective** function suggests we compare the evolution of the inertia or within-cluster variance. Initially, additional centroids decrease the inertia sharply because new clusters improve the overall fit. 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.

Therefore, 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 (take a look at the 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 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.

Figure 13.13 shows an excerpt from the silhouette plot for three and four clusters, where the former highlights the poor fit of cluster 1 by subpar contributions to the global silhouette score, whereas all of the four clusters have some values that exhibit above-average scores.

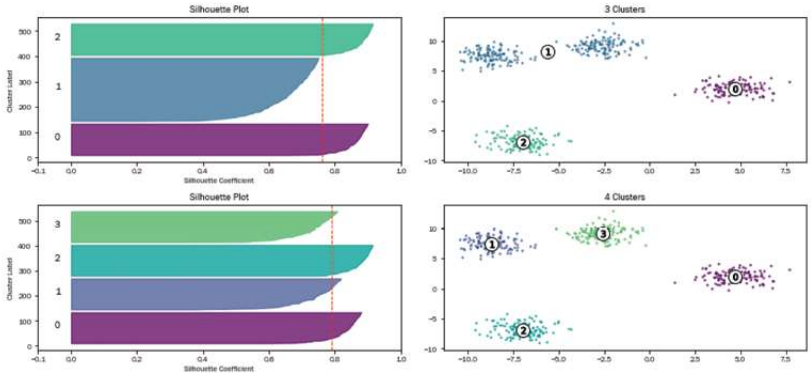

Figure 13.13: Silhouette plots for three and four clusters

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, particularly when some should be given a higher weight and thus be measured on a larger scale. Finally, to validate the robustness of the results, use subsets of data to identify whether particular cluster 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.

Different strategies and dissimilarity measures

There are two approaches to hierarchical clustering:

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 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**: Distance between the nearest neighbors of two clusters
- **Complete link**: Maximum distance between the respective cluster members
- **Ward's method**: Minimize within-cluster variance
- **Group average**: Uses the cluster midpoint as a reference distance

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 that 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 the bottom to the top. Therefore, it is natural to select a clustering by cutting the dendrogram. Refer to the notebook `hierarchical_clustering` for implementation details.

Figure 13.14 illustrates the dendrogram for the classic Iris dataset with four classes and three features using the four different distance metrics introduced in the preceding section. It evaluates the fit of the hierarchical clustering using the **cophenetic correlation** coefficient that 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.

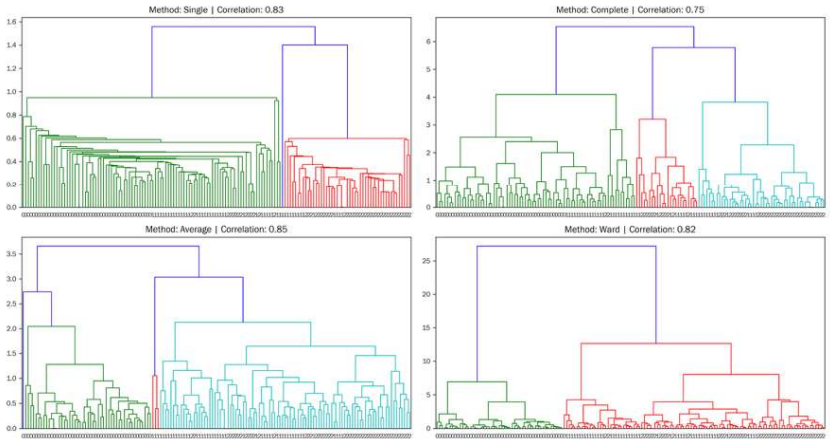

Figure 13.14: Dendrograms and cophenetic correlation for different dissimilarity measures

Different linkage methods produce different dendrogram "looks" so that we cannot use this visualization to compare results across methods. In addition, the Ward method, which minimizes the within-cluster variance, may not properly reflect the change in variance from one level to the next. Instead, the dendrogram can reflect the total within-cluster variance at different levels, which may be misleading. Alternative quality metrics are more appropriate, such as the **cophenetic correlation** or measures like **inertia** if aligned with the overall goal.

The **strengths** of hierarchical clustering include:

- The algorithm does not need the specific number of clusters but, instead, provides insight about potential clustering by means of an intuitive visualization.
- It produces a hierarchy of clusters that can serve as a taxonomy.
- It can be combined with k-means to reduce the number of items at the start of the agglomerative process.

On the other hand, its **weaknesses** include:

- The high cost in terms of computation and memory due to the numerous similarity matrix updates.
- All merges are final so that it does not achieve the global optimum.
- 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 spec-

ification of a certain number of clusters but instead rely on parameters that define the size of a neighborhood and a density threshold.

We'll outline the two popular algorithms: DBSCAN and its newer hierarchical refinement. Refer to the notebook `density_based_clustering` for the relevant code samples and the link in this chapter's `README` on GitHub to a Quantopian example by Jonathan Larking that uses DBSCAN for a pairs trading strategy.

DBSCAN

Density-based spatial clustering of applications with noise (**DBSCAN**) was developed in 1996 and awarded the KDD Test of Time award at the 2014 KDD conference because of the attention it has received in 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 are not assigned to any cluster.

It uses a parameter `eps` 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 (**HDBSCAN**) 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 parameters `min_cluster_size` and `min_samples` 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.

Figure 13.15 shows how DBSCAN and HDBSCAN, respectively, are able to identify clusters that differ in shape significantly from those discovered by k-means, for example. The selection of the clustering algorithm is a function of the structure of your data; refer to the pairs trading strategy that was referenced earlier in this section for a practical example.

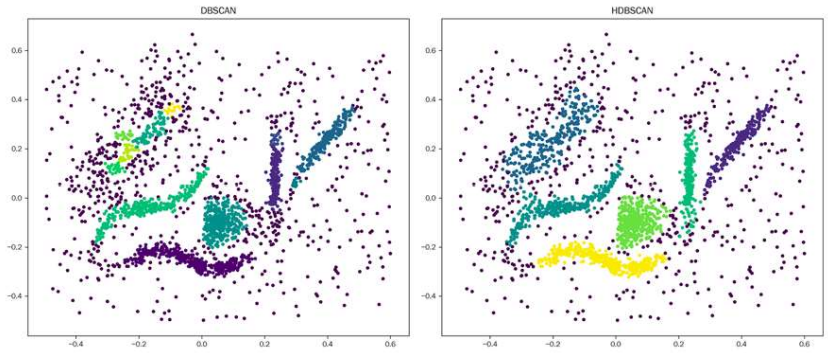

Figure 13.15: Comparing the DBSCAN and HDBSCAN clustering algorithms

Gaussian mixture models

GMMs are generative models that assume 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.

A GMM 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 a probability of being a member of any cluster.

The notebook `gaussian_mixture_models` demonstrates the implementation and visualizes the resulting cluster. You are likely to prefer GMM over other clustering algorithms when the k-means assumption of spherical clusters is too constraining; GMM often needs fewer clusters to produce a good fit given its greater flexibility. The GMM algorithm is also preferable when you need a generative model; because GMM estimates the probability distributions that generated the samples, it is easy to generate new samples based on the result.

Hierarchical clustering for optimal portfolios

In *Chapter 5*, *Portfolio Optimization and Performance Evaluation*, we discussed several methods that aim to choose portfolio weights for a given set of assets to optimize the risk and return profile of the resulting portfolio. These included the mean-variance optimization of Markowitz's modern portfolio theory, the Kelly criterion, and risk parity. In this section, we cover **hierarchical risk parity** (**HRP**), a more recent innovation (Prado 2016) that leverages hierarchical clustering to assign position sizes to assets based on the risk characteristics of subgroups.

We will first present how HRP works and then compare its performance against alternatives using a long-only strategy driven by the gradient boosting models we developed in the last chapter.

How hierarchical risk parity works

The key ideas of hierarchical risk parity are to do the following:

- Use hierarchical clustering of the covariance matrix to group assets with a similar correlation structure together
- Reduce the number of degrees of freedom by only considering similar assets as substitutes when constructing the portfolio

Refer to the notebook and Python files in the subfolder `hierarchical_risk_parity` for implementation 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 that computes the successive clusters using one of several available methods, as discussed previously in this chapter.

```
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 `sns.clustermap` function to visualize the resulting hierarchical clustering. The dendrogram displayed by seaborn shows how individual assets and clusters of assets merged based on their relative distances (see the left panel of *Figure 13.16*).

```
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()
```

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

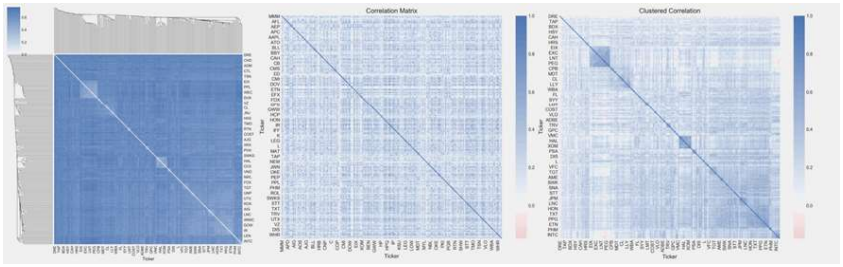

Figure 13.16: Original and clustered correlation matrix

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_inverse_var_pf(cov):
    """Compute the inverse-variance portfolio"""
    ivp = 1 / np.diag(cov)
    return ivp / ivp.sum()
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 a 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 (refer to the notebook for details).

Backtesting HRP using an ML trading strategy

Now that we know how HRP works, we would like to test how it performs in practice compared to some alternatives, namely a simple equal-weighted portfolio and a mean-variance optimized portfolio. You can find the code samples for this section and additional details and analyses in the notebook `pf_optimization_with_hrp_zipline_benchmark`.

To this end, we'll build on the gradient boosting models developed in the last chapter. We will backtest a strategy for 2015-2017 with a universe of the 1,000 most liquid US stocks. The strategy relies on the model predictions to enter long positions in the 25 stocks with the highest positive return prediction for the next day. On a daily basis, we rebalance our holdings so that the weights for our target positions match the values suggested by HRP.

Ensembling the gradient boosting model predictions

We begin by averaging the predictions of the 10 models that performed best during the 2015-16 cross-validation period (refer to *Chapter 12*, *Boosting Your Trading Strategy*, for details), as shown in the following code excerpt:

```
def load_predictions(bundle):
    path = Path('../../12_gradient_boosting_machines/data')
    predictions = (pd.read_hdf(path / 'predictions.h5', 'lgb/train/01')
                   .append(pd.read_hdf(path / 'predictions.h5', 'lgb/test/01').drop('y_test
    predictions = (predictions.loc[~predictions.index.duplicated()]
                   .iloc[:, :10]
                   .mean(1)
                   .sort_index()
                   .dropna()
                  .to_frame('prediction'))
```

On a daily basis, we obtain the model predictions and select the top 25 tickers. If there are at least 20 tickers with positive forecasts, we enter the long positions and close all of the other holdings:

```
def before_trading_start(context, data):
    """
    Called every day before market open.
    """
    output = pipeline_output('signals')['longs'].astype(int)
    context.longs = output[output!=0].index
```

```
    if len(context.longs) < MIN_POSITIONS:
        context.divest = set(context.portfolio.positions.keys())
    else:
        context.divest = context.portfolio.positions.keys() - context.longs
```

Using PyPortfolioOpt to compute HRP weights

PyPortfolioOpt, which we used in *Chapter 5*, *Portfolio Optimization and Performance Evaluation*, to compute mean-variance optimized weights, also implements HRP. We'll run it as part of the scheduled rebalancing that takes place every morning. It needs the return history for the target assets and returns a dictionary of ticker-weight pairs that we use to place orders:

```
def rebalance_hierarchical_risk_parity(context, data):
    """Execute orders according to schedule_function()"""
    for symbol, open_orders in get_open_orders().items():
        for open_order in open_orders:
            cancel_order(open_order)
    for asset in context.divest:
        order_target(asset, target=0)

    if len(context.longs) > context.min_positions:
        returns = (data.history(context.longs, fields='price',
                                bar_count=252+1, # for 1 year of returns
                                frequency='1d')
                   .pct_change()
                   .dropna(how='all'))
        hrp_weights = HRPOpt(returns=returns).hrp_portfolio()
        for asset, target in hrp_weights.items():
            order_target_percent(asset=asset, target=target)
```

Markowitz rebalancing follows a similar process, as outlined in *Chapter 5*, *Portfolio Optimization and Performance Evaluation*, and is included in the notebook.

Performance comparison with pyfolio

The following charts show the cumulative returns for the in- and out-of-sample (with respect to the ML model selection process) of the **equal-weighted** (**EW**), the HRP, and the **mean-variance** (**MV**) optimized portfolios.

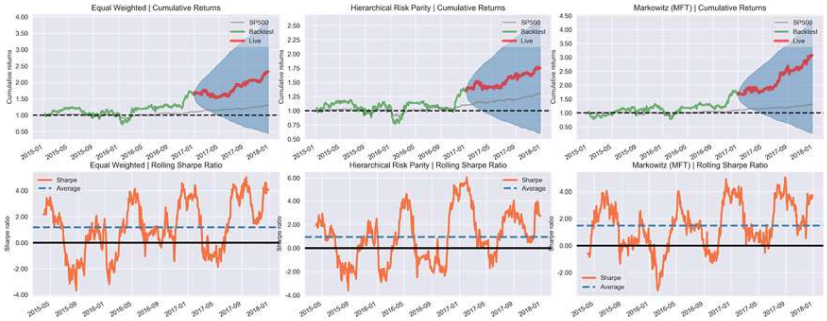

Figure 13.17: Cumulative returns for the different portfolios

The cumulative returns are 207.3 percent for MV, 133 percent for EW, and 75.1 percent for HRP. The Sharpe ratios are 1.16, 1.01, and 0.83, respectively. Alpha returns are 0.28 for MV, 0.16 for EW, and 0.16 for HRP, with betas of 1.77, 1.87, and 1.67, respectively.

Therefore, it turns out that, in this particular context, the often-criticized MV approach does best, while HRP comes up last. However, be aware that the results are quite sensitive to the number of stocks traded, the time period, and other factors.

Try it out for yourself, and learn which technique performs best under the circumstances most relevant for you!

Summary

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

We learned how to use linear dimensionality reduction methods like PCA and ICA to extract uncorrelated or independent components from data that can serve as risk factors or portfolio weights. We also covered advanced nonlinear manifold learning techniques that produce state-of-the-art visualizations of complex, alternative datasets. In the second part of the chapter, 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 machine learning techniques for a key source of alternative data, namely natural language processing for text documents.