5

Build Alpha Factors for Stock Portfolios

Professional traders often construct factor portfolios to target and exploit market inefficiencies, such as anomalies in value, size, or momentum, to generate better risk-adjusted returns. By systematically identifying and weighing securities based on these specific characteristics or factors, investors can create a portfolio that captures the desired exposures while minimizing unintended risks. Factors act as the fundamental building blocks of investing, being the persistent forces that influence returns across various asset classes. A trading edge is a consistent, non-random inefficiency in the market that can be exploited for profit. Factors are the inefficiencies that drive asset prices and form the basis of this edge, allowing traders to capitalize on these persistent anomalies.

Factor analysis is a broad topic but comes down to identifying the factors, determining the sensitivity of a portfolio to those factors, and taking action. That action can be hedging undesirable risk based on the factor exposure or increasing exposure to the factor. In this chapter, we explore the key elements of identifying factors, hedging out unwanted risks, and setting up forward returns to assess the predictive power of factors. We'll use Python libraries for statistical modeling to build a principal component analysis and linear regression. Then, we'll introduce the Zipline Reloaded Pipeline API, which will prepare us for analyzing factors.

In this chapter, we present the following recipes:

- Identifying latent return drivers using principal component analysis
- Finding and hedging portfolio beta using linear regression
- Analyzing portfolio sensitivities to the Fama-French factors
- Assessing market inefficiency based on volatility
- Preparing a factor ranking model using the Zipline Reloaded Pipeline API

Identifying latent return drivers using principal component analysis

Principal component analysis (**PCA**) is a dimensionality reduction technique that is widely used in data science. It transforms the original features into a new set of features, called **principal components**, which reflect the maximum variance in the data. In other words, it transforms a large set of variables into a smaller set of variables, while still containing most of the information from the larger set.

There are various sources of risk in an asset portfolio, including market risk, sector risk, and asset-specific risk. PCA helps identify and quantify these risks by breaking down the returns of the portfolio into components that explain the maximum variance. The first few principal components usually capture most of the variance and they can be analyzed to understand the major sources of risk in the portfolio.

This recipe will use scikit-learn to run PCA on a portfolio of eight stocks made of up mining and technology companies.

Getting ready

For this recipe, we introduce scikit-learn, which provides algorithms for classification, regression, clustering, dimensionality reduction, and more. It's built on NumPy, SciPy, and Matplotlib, which makes it easy to integrate with other scientific Python libraries. We'll use it to conduct our analysis.

You can install scikit-learn using `pip`:

```
pip install scikit-learn
```

How to do it...

Scikit-learn makes it easy to run a PCA. Here's how to do it:

1. Import the libraries we need for the analysis:

```
import numpy as np
import pandas as pd
```

```
from openbb import obb
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
obb.user.preferences.output_type = "dataframe"
```

2. Download data for gold mining stocks and healthcare stocks and compute their daily returns:

```
symbols = ["NEM", "RGLD", "SSRM", "CDE", "LLY", "UNH",
           "JNJ", "MRK"]
data = obb.equity.price.historical(
    symbols,
    start_date="2020-01-01",
    end_date="2022-12-31",
    provider="yfinance",
).pivot(columns="symbol", values="close")
returns = data.pct_change().dropna()Run the PCA using three components and fit the
pca = PCA(n_components=3)
pca.fit(returns)
```

3. Extract the explained variance ratio for each component and extract the principal components:

```
pct = pca.explained_variance_ratio_
pca_components = pca.components_
```

4. Plot the contribution of each principal component and the cumulative percent of explained variance:

```
cum_pct = np.cumsum(pct)
x = np.arange(1, len(pct) + 1, 1)
plt.subplot(1, 2, 1)
plt.bar(x, pct * 100, align="center")
plt.title("Contribution (%)")
plt.xticks(x)
plt.xlim([0, 4])
plt.ylim([0, 100])
plt.subplot(1, 2, 2)
plt.plot(x, cum_pct * 100, "ro-")
plt.title("Cumulative contribution (%)")
plt.xticks(x)
plt.xlim([0, 4])
plt.ylim([0, 100])
```

The last code block results in the following chart:

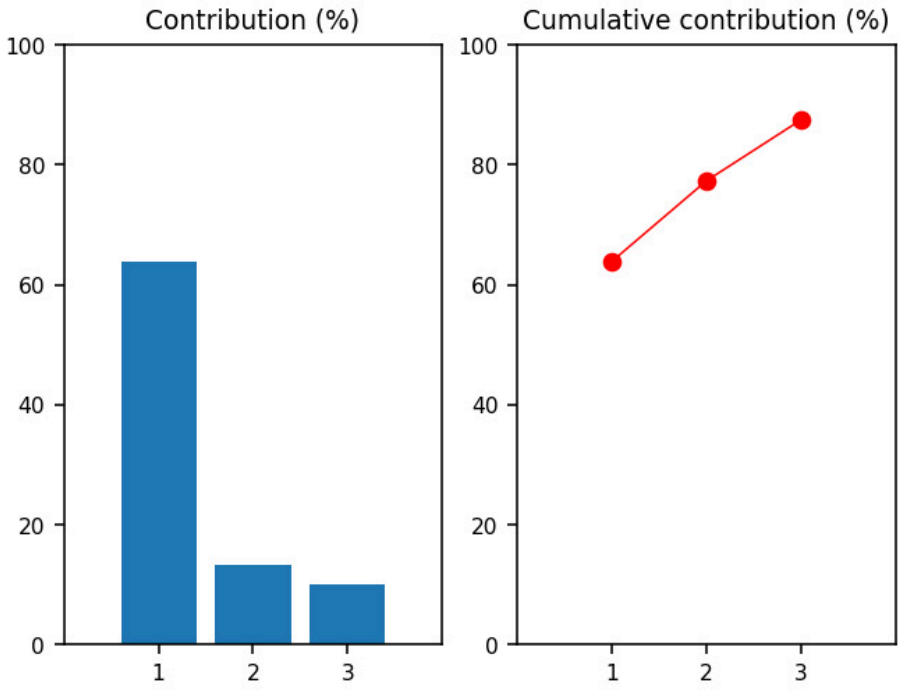

Figure 5.1: Contribution and cumulative contribution of explained variance for the first three principal components

How it works...

There is ample documentation on how PCA works so we will not cover the math behind the process. But in summary, the `PCA(n_components=3)` code creates a PCA object that will compute the first three principal components of the `returns` data. The `n_components=3` parameter specifies that we want to select the top three principal components. The call to `pca.fit` fits the PCA model to the `returns` data, which hides most of the complexity of PCA. Under the hood, PCA standardizes the returns and computes a covariance matrix. Next, the eigen decomposition is performed on the covariance matrix, which computes the eigenvectors and eigenvalues that indicate the directions and magnitude of data variance. Finally, eigenvectors are ordered by their eigenvalues in descending order and the top three are chosen as principal components.

There's more...

From the principal components, we can transform the original `returns` data into a new set of features that represent the statistical risk factors that explain the most variance in the returns, as shown in the following code:

```
X = np.asarray(returns)
factor_returns = X.dot(pca_components.T)
factor_returns = pd.DataFrame(
    columns=["f1", "f2", "f3"],
    index=returns.index,
    data=factor_returns
)
```

The preceding code will return the following DataFrame containing a time series of the statistical risk factors:

Date	f1	f2	f3
2020-01-03	-0.028845	0.007909	-0.013464
2020-01-06	-0.081169	-0.006508	0.058871
2020-01-07	0.003694	0.015644	0.013973
2020-01-08	-0.105891	-0.028187	0.005725
2020-01-09	0.001918	-0.018469	-0.023525

Figure 5.2: DataFrame containing a time series of the statistical risk factors

The statistical risk factors are similar to more conventional risk factors such as the Fama-French factors we'll explore in the *Analyzing portfolio sensitivities to the Fama-French factors* recipe in this chapter. The **`returns`** data is projected onto the principal components obtained from the PCA. This is done by taking the dot product of **`X`** with the transpose of the **`pca_components`** matrix. The **`pca_components`** matrix contains the principal components of the **`returns`** data, which were previously computed using the **`fit`** method.

This step transforms the original **`returns`** data into a new set of features (the principal components or factors) that explain the most variance in the data. These factors should give us an idea of how much of the portfolio's returns come from some unobservable statistical feature.

With the principal components, we can create the exposure of each asset to the three principal components:

```
factor_exposures = pd.DataFrame(
    index=["f1", "f2", "f3"],
    columns=returns.columns,
    data=pca_components
).T
```

The resulting `factor_exposures` DataFrame is shown in the following screenshot. It shows how much each asset in the portfolio is exposed to each of the three factors. The exposure of each asset to each of the three factors (`"f1", "f2", "f3"`) represents how much the returns of that asset are influenced by changes in those factors.

	f1	f2	f3
NEM	0.292250	0.091439	0.394104
RGLD	0.308669	0.125098	0.416761
SSRM	0.427029	0.227828	0.485778
CDE	0.783970	-0.030216	-0.618241
LLY	0.062208	-0.605558	0.131696
UNH	0.101891	-0.535551	0.136475
JNJ	0.061262	-0.338930	0.108649
MRK	0.065923	-0.393423	0.070823

Figure 5.3: DataFrame containing the exposure of each asset to each factor

We can visualize each asset's exposure to the first two principal components on an annotated scatter plot:

```
labels = factor_exposures.index
data = factor_exposures.values
plt.scatter(data[:, 0], data[:, 1])
plt.xlabel("factor exposure of PC1")
plt.ylabel("factor exposure of PC2")
for label, x, y in zip(labels, data[:, 0], data[:, 1]):
    plt.annotate(
        label,
        xy=(x, y),
        xytext=(-20, 20),
        textcoords="offset points",
        arrowprops=dict(
            arrowstyle="->",
            connectionstyle="arc3,rad=0"
        ),
    )
```

The result is the following scatter plot showing the exposure to the factors:

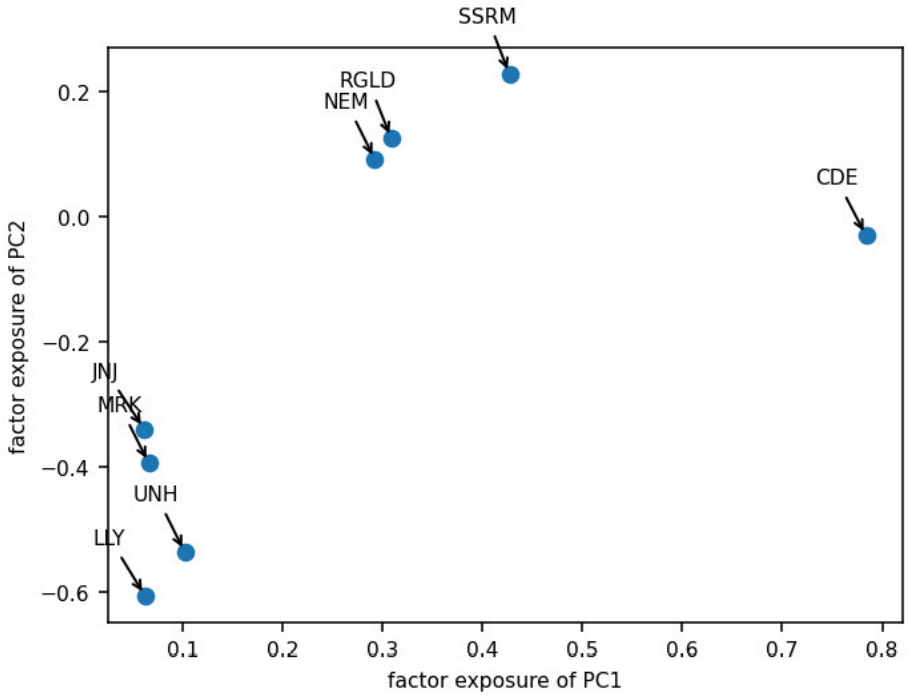

Figure 5.4: Scatter plot showing the stock exposure to the first two principal components

The gold mining stocks have higher factor exposures compared to the healthcare stocks, indicating that they are more sensitive to changes in the underlying factors. The healthcare stocks have lower and negative exposures to `"f2"`, indicating that they may provide some diversification benefits in a portfolio that is highly exposed to this factor. The factors may represent different sources of systematic risk in the market, such as market risk, interest rate risk, or industry-specific risks.

See also

PCA is an important dimensionality reduction technique commonly used in portfolio management. You can visit the following links to learn more about PCA:

- More on PCA: https://en.wikipedia.org/wiki/Principal_component_analysis
- Documentation for scikit-learn: https://scikit-learn.org/stable/
- Documentation for the scikit-learn implementation of PCA used in this recipe: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Finding and hedging portfolio beta using linear regression

Algorithmic traders often seek exposure to specific risks that they believe will yield outsized returns while hedging other risks they deem unfavorable or unnecessary. For instance, a trader might want exposure to stocks with the lowest price-to-earnings ratios, believing they will outperform while hedging against the broader market risk. This selective exposure helps traders maximize returns by capitalizing on perceived opportunities while minimizing the potential downside by hedging against certain risks.

Factor models are a way of explaining the returns of an asset or portfolio through a combination of the returns of another asset, portfolio, or factor. The general form of a factor model using a linear combination is as follows:

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \beta_n X_n$$

The sensitivity of portfolio returns to a risk factor X is described by the beta. It's the beta that must be hedged to concentrate exposure to the risk factors.

In this recipe, we'll construct a portfolio of stocks, compute the beta of the portfolio returns to **`SPY`** returns, and construct a hedging portfolio to neutralize the broader market exposure.

Getting ready

For this recipe, we introduce Statsmodels, which provides classes and functions for estimating many statistical models and for conducting statistical tests and statistical data exploration. We'll use it to conduct our analysis.

You can install Statsmodels using **`pip`**:

```
pip install statsmodels
```

How to do it...

We'll use Statsmodels to measure the sensitivity of a portfolio of stocks to a benchmark, then offset the portfolio with a short position to hedge beta:

1. Import the libraries for the analysis:

```
import numpy as np
import pandas as pd
from openbb import obb
import statsmodels.api as sm
from statsmodels import regression
import matplotlib.pyplot as plt
obb.user.preferences.output_type = "dataframe"
```

2. Download data for the same portfolio we used in the previous recipe. Note the inclusion of the **`SPY` exchange-traded fund (ETF)**, which we'll use to represent the broad market returns:

```
symbols = ["NEM", "RGLD", "SSRM", "CDE", "LLY", "UNH", "JNJ", "MRK", "SPY"]
data = obb.equity.price.historical(
    symbols,
    start_date="2020-01-01",
    end_date="2022-12-31",
    provider="yfinance"
).pivot(columns="symbol", values="close")
```

3. Pop the column with **`SPY`** data off the DataFrame and compute the returns:

```
benchmark_returns = (
    data
    .pop("SPY")
    .pct_change()
    .dropna()
)
```

4. Now, compute the returns for the portfolio:

```
portfolio_returns = (
    data
    .pct_change()
    .dropna()
    .sum(axis=1)
)
```

5. Set the **`name`** property on the series for plotting purposes:

```
portfolio_returns.name = "portfolio"
```

6. Plot the portfolio returns and the benchmark returns to visualize the difference:

```
portfolio_returns.plot()
benchmark_returns.plot()
```

```
plt.ylabel("Daily Return")
plt.legend()
```

7. The result is a plot of the daily returns during the analysis period:

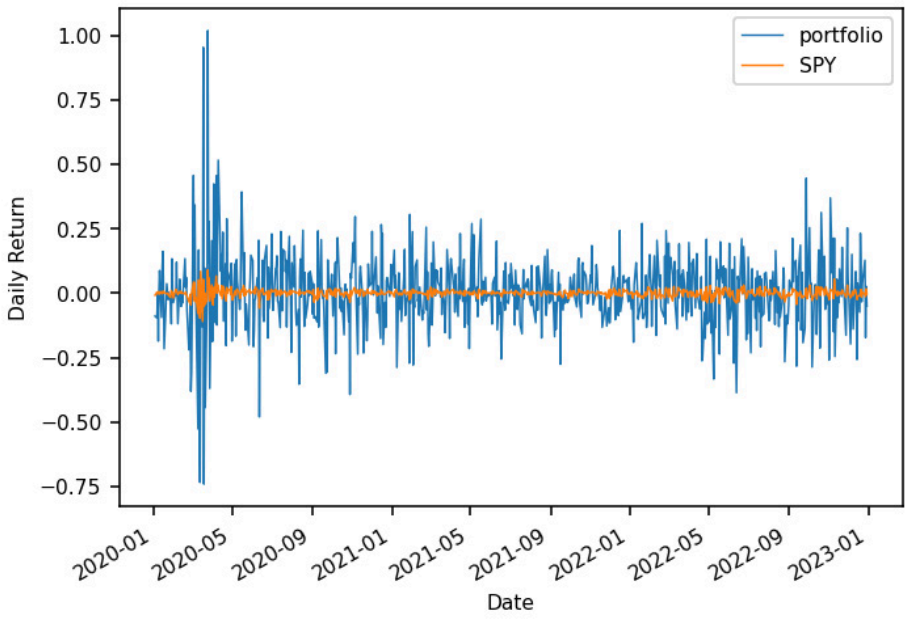

Figure 5.5: Plot of portfolio and benchmark returns

8. Create a function that returns the alpha and beta coefficients from a linear regression:

```
X = benchmark_returns.values
Y = portfolio_returns.values
def linreg(x, y):
    x = sm.add_constant(x)
    model = regression.linear_model.OLS(y, x).fit()
    x = x[:, 1]
    return model.params[0], model.params[1]
```

9. Generate the alpha and beta coefficients from the regression between the portfolio and benchmark returns:

```
alpha, beta = linreg(X, Y)
print(f"Alpha: {alpha}")  # => 0.0028
print(f"Beta: {beta}")  # => 5.5745
```

10. Create a scatter plot that visualizes the linear relationship between the portfolio and benchmark returns:

```
X2 = np.linspace(X.min(), X.max(), 100)
Y_hat = X2 * beta + alpha
plt.scatter(X, Y, alpha=0.3)
plt.xlabel("SPY daily return")
plt.ylabel("Portfolio daily return")
plt.plot(X2, Y_hat, "r", alpha=0.9)
```

The result is the following scatter plot showing the daily returns and their linear relationship:

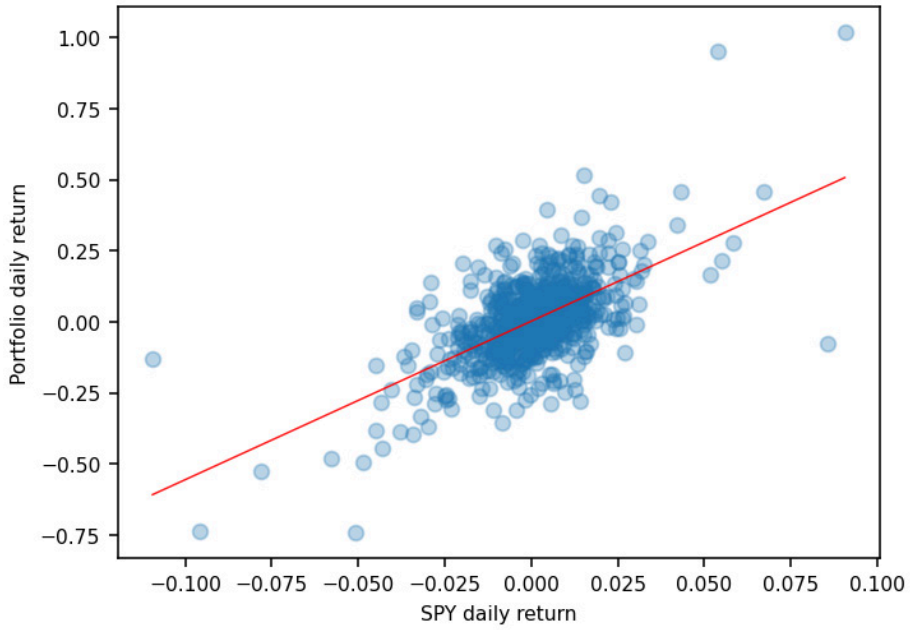

Figure 5.6: Scatter plot showing the linear relationship between the portfolio and benchmark returns

11. Finally, construct a time series of portfolio returns including the beta hedge:

```
hedged_portfolio_returns = -1 * beta * benchmark_returns + portfolio_returns
```

12. Rerun the regression to confirm that the beta is **0**:

```
P = hedged_portfolio_returns.values
_, beta = linreg(X, P)
print(f"Beta: {beta}")  => 0.0000
```

How it works...

We demonstrate the beta hedge by first constructing a portfolio of eight stocks by adding up the daily returns of each stock for each day. We use the S&P 500 tracking ETF as a proxy for the broader market. The ETF symbol is SPY. We download price data for SPY along with our portfolio for convenience since we can use the pandas **pop** method in the SPY column as a separate series. We then compute the daily returns for both the portfolio and the benchmark.

The next step is running a regression that is a few lines of code. **X** is the independent variable representing the benchmark returns, and **Y** is the dependent variable representing the portfolio returns. The **linreg** func-

tion takes these returns, adds a constant term to `X` for the intercept, and then fits a linear model using ordinary least squares regression. It then returns the parameters of the fitted model, which are the intercept (alpha) and the slope (beta).

> *IMPORTANT NOTE*
>
> *Adding a constant term for the intercept is crucial for accurately modeling the relationship between the dependent and independent variables in a linear regression. If we do not include a constant term for the intercept, we are essentially forcing the regression line to pass through the origin (`0, 0`), which may not be an accurate representation of the relationship between the variables.*

After we have our alpha and beta terms, we can build a hedged portfolio. As we learned in the introduction to this recipe, traders hedge exposure to risk that they don't want. If we can determine that our portfolio returns are linked to a risk factor through a linear relationship, then we can initiate a short position in that risk factor – in our case, the broader market represented by SPY. The amount we want to trade is represented by our beta. This is effective because if our returns are made up of $\alpha + \beta_{SPY}$, then going short β_{SPY} will result in our new returns being $\alpha + \beta_{SPY} - \beta_{SPY} = \alpha$. That is, no exposure to the returns of SPY. We validate that the beta of the hedged portfolio is `0` by rerunning the regression.

There's more...

The information ratio is a measure used to evaluate the risk-adjusted performance of a trading strategy. It is calculated by dividing the portfolio's active return (the difference between the portfolio return and the benchmark return) by the active risk (the standard deviation of the active return). The information ratio is a useful way to compare two portfolios against their benchmark. Let's build a function that computes it:

```
def information_ratio(
    portfolio_returns,
    benchmark_returns
):
    active_return = portfolio_returns - benchmark_returns
    tracking_error = active_return.std()
    return active_return.mean() / tracking_error
```

Let's use the information ratio to compare the differences between our hedged and unhedged portfolios:

```
hedged_ir = information_ratio(
    hedged_portfolio_returns,
    benchmark_returns
)
unhedged_ir = information_ratio(
    portfolio_returns,
    benchmark_returns
)
print(f"Hedged information ratio: {hedged_ir}")
print(f"Unhedged information ratio: {unhedged_ir}")
```

Running the preceding code block shows us the unhedged portfolio has a lower risk-adjusted return. That's to be expected since the returns attributable to the benchmark have been hedged.

See also

Hedging is an important part of proper risk management and successful trading. Using a simple linear relationship between the portfolio returns and the factor returns is a great way to neutralize unwanted risk. To learn more about the topic that we covered in this recipe, visit the following links:

- Statsmodels documentation on ordinary least squares: **https://www.statsmodels.org/stable/examples/notebooks/generated/ols.html**
- More about how beta is used in portfolio management and trading: **https://www.investopedia.com/terms/b/beta.asp**
- More about how to use the information ratio: **https://pyquantnews.com/how-to-measure-skill-portfolio-manager/**

Analyzing portfolio sensitivities to the Fama-French factors

The Fama-French factors are a set of factors identified by economists Eugene F. Fama and Kenneth R. French to explain the variation in stock returns. These factors serve as the foundation of the Fama-French three-factor model. In ***Chapter 1***, *Acquire Free Financial Market Data with Cutting-edge Python Libraries*, we learned how to use **`pandas_datareader`** to download Fama-French factor data.

In this recipe, we'll compute the exposure of our portfolio to the size and market value factors.

Getting ready

You should already have **pandas_datareader** installed from ***Chapter 1***, *Acquire Free Financial Market Data with Cutting-edge Python Libraries*. If not, you can install it using **pip**:

```
pip install pandas_datareader
```

We'll also assume you have the **data** DataFrame loaded with historic prices created in the last recipe.

How to do it...

We'll reuse what we learned in the prior recipes about using beta to compute factor sensitivities.

1. Import the libraries we need for the analysis:
```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
from openbb import obb
import statsmodels.api as sm
from statsmodels import regression
from statsmodels.regression.rolling import RollingOLS
obb.user.preferences.output_type = "dataframe"
```
2. Now, resample the daily return data to monthly using **asfreq** and replace the row labels to monthly format using **to_period**. This is the format of the Fama-French factor data. A consistent format lets us align the data in the same DataFrame later:
```
monthly_returns = (
    data
    .asfreq("M")
    .dropna()
    .pct_change(fill_method=None)
    .to_period("M")
)
```
3. The result is a DataFrame with monthly returns suitable for alignment with the Fama-French monthly factor data:

Date	NEM	RGLD	SSRM	CDE	LLY	UNH	JNJ	MRK	SPY
2015-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2015-02	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2015-03	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2015-04	0.220175	0.022500	0.203090	0.108280	-0.010736	-0.058247	-0.013917	0.036187	0.009834
2015-05	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
...	...	...	...	...	...	...	...	...	...
2022-07	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2022-08	-0.306854	-0.139352	-0.188644	-0.092105	-0.067904	0.011098	-0.084986	-0.063727	0.047528
2022-09	0.029778	0.020892	0.090437	0.239130	0.073432	-0.024465	0.012520	0.017037	-0.092446
2022-10	0.006900	0.015798	-0.061863	0.105263	0.119808	0.099220	0.064949	0.175104	0.081275
2022-11	0.121692	0.182919	0.103917	-0.074074	0.027687	-0.013312	0.029769	0.088142	0.055592

Figure 5.7: DataFrame with monthly portfolio returns

4. Next, we compute the active returns and use them as the dependent variable. This will help us understand the sensitivity of the Fama-French factors to our active returns:

```
bench = monthly_returns.pop("SPY")
R = monthly_returns.mean(axis=1)
active = (R - bench).dropna()
```

5. Now, let's download the Fama-French data using **`pandas_datareader`**:

```
factors = pdr.get_data_famafrench(
    'F-F_Research_Data_Factors',
    start="2015-01-01",
    end="2022-12-31"
)[0][1:] / 100
SMB = factors.loc[active.index, "SMB"]
HML = factors.loc[active.index, "HML"]
```

6. Now that we have our returns and factor data, build a pandas DataFrame that aligns the data along the common date index:

```
df = pd.DataFrame(
    {
        "R": active,
        "SMB": SMB,
        "HML": HML,
    },
    index=active.index
).dropna()
```

7. Run the regression and get the beta coefficients for each factor. These coefficients represent the exposure of our active returns to the Fama-French factors:

```
b1, b2 = regression.linear_model.OLS(
    df.R,
```

```
    df[["SMB", "HML"]]
).fit().params
```

8. To see how these sensitivities evolve through time, use the Statsmodels **`RollingOLS`** class:

```
exog = sm.add_constant(df[["SMB", "HML"]])
rols = RollingOLS(active, exog, window=12)
rres = rols.fit()
fig = rres.plot_recursive_coefficient(
    variables=["SMB", "HML"],
    figsize=(5.5, 6.6)
)
```

This results in the following two charts that plot the 12-month rolling regression over time along with the 95% confidence intervals:

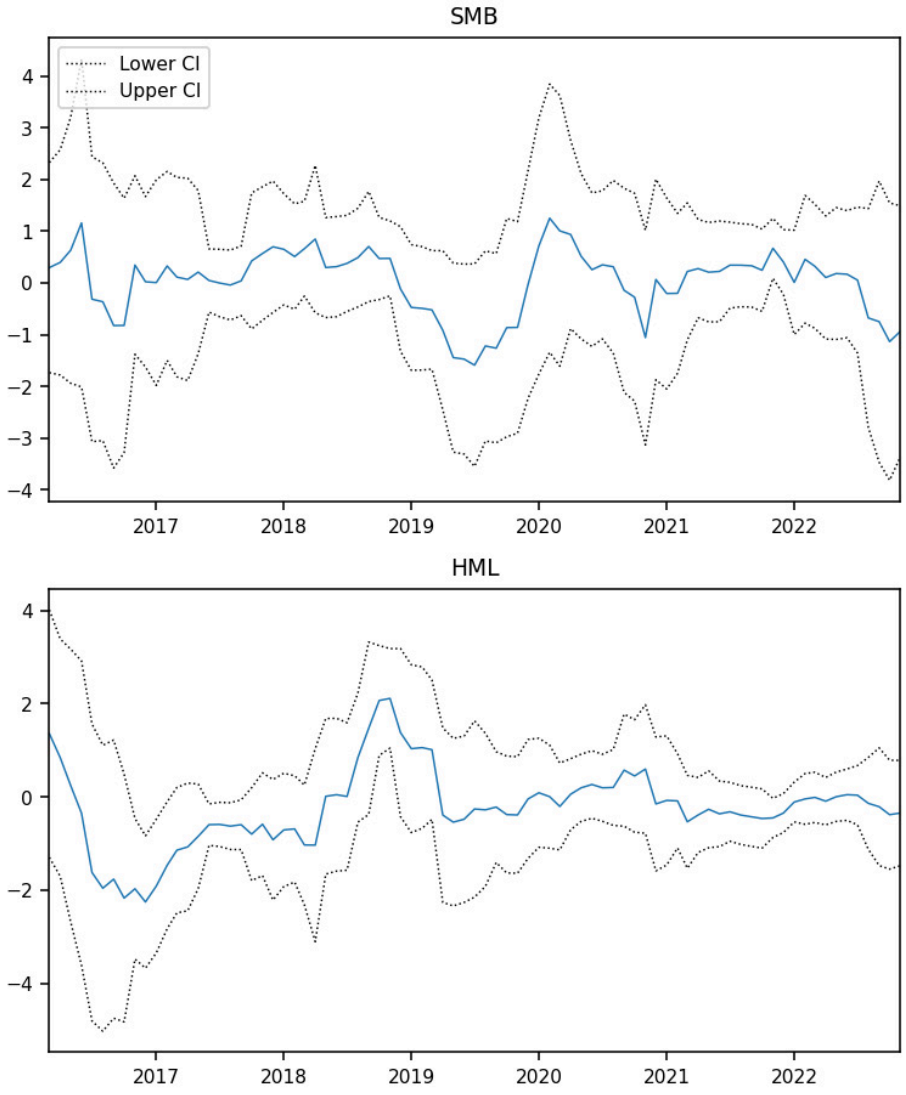

Figure 5.8: Betas for two of the Fama-French factors using a 12-month rolling window

How it works...

We follow the same procedure in downloading financial market data using OpenBB and computing portfolio returns as in the *Finding and hedging portfolio beta using linear regression* recipe in this chapter. The Fama-French factor data is in monthly resolution, so we use **`asfreq`** and **`to_period`** to resample our data price data to monthly, then compute monthly returns.

From there, we compute active returns, which are the portion of portfolio returns that cannot be attributed to the market's overall movement and are instead a result of active portfolio management decisions. As we learned, investment factors are systematic risk factors that explain the differences in returns between different securities. A portfolio's active return can be influenced by its exposure to these factors.

To measure the exposure, we compute the beta coefficients of the factors using the **`OLS`** method from the Statsmodels library to fit a linear regression model. The dependent variable is **`df.R`**, which represents the active returns of our portfolio, while the independent variables are **`df["SMB"]`** and **`df["HML"]`**, representing the **Small Minus Big** (**SMB**) and **High Minus Low** (**HML**) factors of the Fama-French three-factor model, respectively. The **`fit`** method is used to estimate the parameters of the model, which, in this case, are the betas for the **`SMB`** and **`HML`** factors stored in **`b1`** and **`b2`**.

To get an idea of how the exposures evolve through time, we build a rolling regression using the **`RollingOLS`** class. The exogenous variable is created by adding a constant column (for the intercept) to the **`SMB`** and **`HML`** factors. The **`RollingOLS`** class is then initialized with a window size of **`12`**, and the **`fit`** method is called to compute the rolling OLS estimates. Finally, the **`plot_recursive_coefficient`** method is used to create a plot of the rolling estimates of the regression coefficients for **`SMB`** and **`HML`** over the 12-period window.

There's more...

Marginal Contribution to Active Risk (**MCAR**) quantifies the additional active risk each factor brings to your portfolio. To compute the MCAR of the factors, we multiply the factor sensitivity by the covariance between the factors and then divide by the square of the standard deviation of the active returns. This calculation shows the amount of risk incurred by being exposed to each factor, considering the exposures to other factors already in your portfolio. The risk contribution that is not explained is the exposure to factors other than the ones being analyzed.

```
F1 = df.SMB
F2 = df.HML
cov = np.cov(F1, F2)
ar_squared = (active.std()) ** 2
mcar1 = (b1 * (b2 * cov[0, 1] + b1 * cov[0, 0])) / ar_squared
mcar2 = (b2 * (b1 * cov[0, 1] + b2 * cov[1, 1])) / ar_squared
print(f"SMB risk contribution: {mcar1}")
print(f"HML risk contribution: {mcar2}")
print(f"Unexplained risk contribution: {
    1 - (mcar1 + mcar2)}")
```

Running the preceding code shows only about 0.7% of the risk is explained by these two factors. Let's plot the MCAR for these factors over time:

1. Compute the 12-month rolling covariances between the factors:

```
covariances = (
    df[["SMB", "HML"]]
    .rolling(window=12)
    .cov()
).dropna()
```

2. Compute the 12-month rolling active return squared:

```
active_risk_squared = (
    active.rolling(window=12).std()**2
).dropna()
```

3. Combine the rolling factor betas:

```
betas = pd.concat(
    [rres.params.SMB, rres.params.HML],
    axis=1
).dropna()
```

4. Create an empty DataFrame to store the rolling MCAR data:

```
MCAR = pd.DataFrame(
    index=betas.index,
    columns=betas.columns
)
```

5. Loop through each factor and each date computing the MCAR value:

```
for factor in betas.columns:
    for t in betas.index:
        s = np.sum(
            betas.loc[t] * covariances.loc[t][factor])
        b = betas.loc[t][factor]
        AR = active_risk_squared.loc[t]
        MCAR[factor][t] = b * s / AR
```

6. Finally, plot the MCAR for each factor as follows:

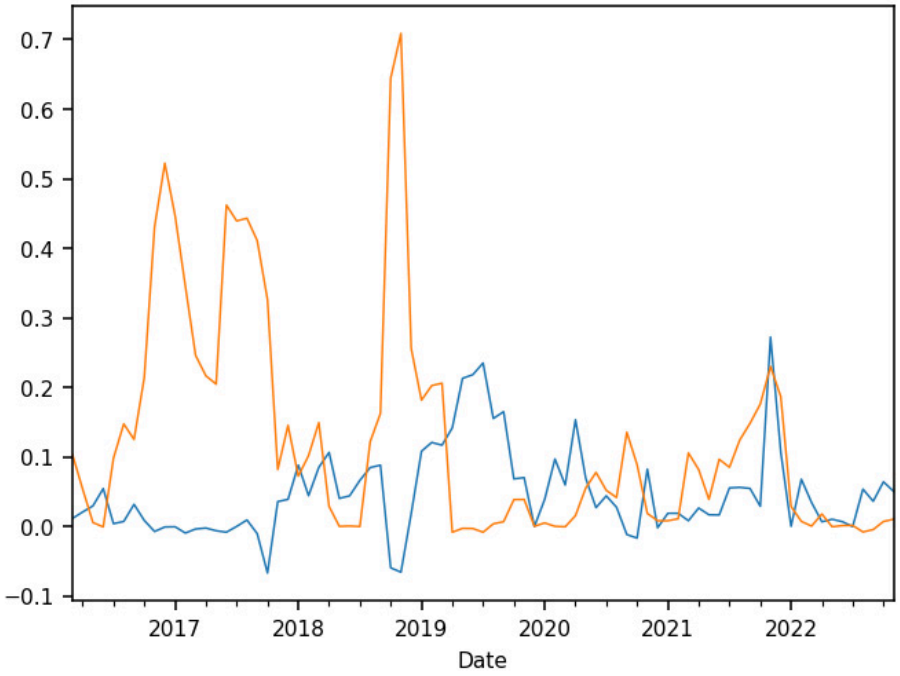

Figure 5.9: Rolling marginal contribution to risk for the two Fama-French factors

See also

By systematically selecting securities with certain factor characteristics, investors aim to achieve better risk-adjusted returns compared to the broader market or a specific benchmark. Factor investing is a broad and deep topic. To learn more about factor investing, visit the following links:

- BlackRock's introduction to factor investing: **https://www.blackrock.com/us/individual/investment-ideas/what-is-factor-investing**
- *Advances in Active Portfolio Management* by Richard Grinold and Ronald Kahn is an advanced guide that delves deep into the quantitative aspects of active portfolio management, including topics such as portfolio construction, risk management, and implementation: **https://amzn.to/44zJeMk**

Assessing market inefficiency based on volatility

Using volatility as a factor reflects the market inefficiency related to the pricing of volatile stocks. Historically, stocks with lower volatility have

tended to outperform those with higher volatility on a risk-adjusted basis. This phenomenon contradicts the traditional finance theory that higher risk should be compensated with higher return and thus represents a market inefficiency.

One of the first steps in analyzing a factor's performance is calculating forward returns. Forward returns are the returns of a security in a future period. Once the forward returns are calculated, we can use the Spearman rank correlation to understand the relationship between the factor and the forward returns. The Spearman rank correlation assesses how well the relationship between two variables can be described using a monotonic function. A high Spearman rank correlation indicates that the factor ranks securities in a way that is closely aligned with the ranking of securities by their forward returns, suggesting that the factor has predictive power. Conversely, a low Spearman rank correlation suggests that the factor is not a good predictor of forward returns.

The Parkinson estimator uses the high and low prices over a period to get a more accurate measure of volatility. The basic idea is that the high and low prices encapsulate more information about the price movement than just the closing prices. In this recipe, we'll build a factor using Parkinson volatility, compute the forward returns, and determine the Spearman rank correlation between the factor and forward returns.

How to do it…

We'll calculate Parkinson volatility and forward returns to measure the predictive power of Parkinson volatility as a tradeable factor:

1. Import the libraries we'll use for the analysis:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from openbb import obb
from scipy.stats import spearmanr
obb.user.preferences.output_type = "dataframe"
```

2. Use OpenBB to download data:

```
symbols = ["NEM", "RGLD", "SSRM", "CDE", "LLY", "UNH",
           "JNJ", "MRK"]
data = obb.equity.price.historical(
    symbols,
    start_date="2015-01-01",
```

```
    end_date="2022-12-31",
    provider="yfinance"
)
prices = data[["high", "low", "close", "volume",
               "symbol"]]
```

3. As a preprocessing step, let's make sure all our tickers have at least two years of data. We'll create a mask and grab the stocks that meet our criteria:

```
nobs = prices.groupby("symbol").size()
mask = nobs[nobs > 2 * 12 * 21].index
prices = prices[prices.symbol.isin(mask)]
```

4. Next, set the symbol column as an index, reorder, and drop duplicates:

```
prices = (
    prices
    .set_index("symbol", append=True)
    .reorder_levels(["symbol", "date"])
    .sort_index(level=0)
).drop_duplicates()
```

   The result is the following MultiIndex DataFrame with **`symbol`** as the first index and **`date`** as the second:

		high	low	close	volume
symbol	**date**				
CDE	**2015-01-02**	5.30	4.96	5.30	2864400
	2015-01-05	5.45	5.14	5.44	2997000
	2015-01-06	5.73	5.45	5.69	3885200
	2015-01-07	5.94	5.47	5.61	4099700
	2015-01-08	5.85	5.51	5.60	2903300

Figure 5.10: MultiIndex DataFrame with symbol and date as indexes

5. Next, we'll create a function that returns the normalized Parkinson volatility estimate:

```
def parkinson(data, window=14, trading_days=252):
    rs = (1.0 / (4.0 * np.log(2.0))) * ((
        data.high / data.low).apply(np.log))**2.0
    def f(v):
        return (trading_days * v.mean())**0.5
    result = rs.rolling(
        window=window,
        center=False
    ).apply(func=f)
    return result.sub(result.mean()).div(result.std())
```

6. We'll apply that function to each group of stocks and add the Parkinson estimator as a new column:

```
prices["vol"] = (
    prices
    .groupby("symbol", group_keys=False)
    .apply(parkinson)
)
prices.dropna(inplace=True)
```

The result is the same MultiIndex DataFrame with each stock's normalized Parkinson volatility added as a new column:

		high	low	close	volume	vol
symbol	date					
CDE	2015-01-22	6.43	6.15	6.31	3404900	0.502344
	2015-01-23	6.25	5.93	5.99	2234000	0.466274
	2015-01-26	6.13	5.69	6.11	2149800	0.512970
	2015-01-27	6.42	6.12	6.32	3697700	0.508157
	2015-01-28	6.36	5.89	5.99	2668300	0.488355

Figure 5.11: Updated MultiIndex DataFrame with per-stock normalized Parkinson volatility

7. Now that we have the normalized Parkinson volatility, we can compute historic and forward returns. First, compute the historic returns over **1**, **5**, **10**, **21**, **42**, and **63** periods representing one day through three months:

```
lags = [1, 5, 10, 21, 42, 63]
for lag in lags:
    prices[f"return_{lag}d"] = (
        prices
        .groupby(level="symbol")
        .close
        .pct_change(lag)
    )
```

8. Compute the forward returns for the same periods:

```
for t in [1, 5, 10, 21, 42, 63]:
    prices[f"target_{t}d"] = (
        prices
        .groupby(level="symbol")[f"return_{t}d"]
        .shift(-t)
    )
```

The result is new columns in the prices DataFrame representing the historic and forward returns, as shown in the following screenshot:

date	high	low	close	volume	vol	return_1d	return_5d	return_10d	return_21d	return_42d	return_63d	target_1d	target_5d	target_10d	target_21d	target_42d	target_63d
2015-01-22	24.69	23.98	24.29	13132700	1.021514	NaN	NaN	NaN	NaN	NaN	NaN	-0.005764	-0.004940	0.021820	0.068753	-0.058872	-0.034582
2015-01-23	24.43	23.67	24.15	12056800	0.976772	-0.005764	NaN	NaN	NaN	NaN	NaN	0.014079	0.041408	-0.004141	0.068737	-0.069151	0.034369
2015-01-26	24.63	23.35	24.49	11189200	1.059908	0.014079	NaN	NaN	NaN	NaN	NaN	0.026541	0.025316	0.004900	0.063699	-0.095141	0.044916
2015-01-27	25.25	24.53	25.14	10774500	0.948081	0.026541	NaN	NaN	NaN	NaN	NaN	-0.035004	-0.020684	-0.029037	0.046539	-0.115752	0.048130
2015-01-28	25.05	24.03	24.26	11251300	0.962112	-0.035004	NaN	NaN	NaN	NaN	NaN	-0.003710	0.023083	-0.004946	0.085326	-0.092333	0.091509
2015-01-29	24.34	23.42	24.17	8850000	1.016856	-0.003710	-0.004940	NaN	NaN	NaN	NaN	0.040546	0.026893	0.014067	0.071163	-0.101779	0.095987
2015-01-30	25.22	23.92	25.15	11594400	1.139382	0.040546	0.041408	NaN	NaN	NaN	NaN	-0.001590	-0.043738	-0.014712	0.018688	-0.096620	0.053280
2015-02-02	25.20	24.56	25.11	7497000	1.109351	-0.001590	0.025316	NaN	NaN	NaN	NaN	-0.019514	-0.019912	-0.036639	0.003186	-0.110315	0.037435
2015-02-03	25.07	24.14	24.62	9038100	0.957277	-0.019514	-0.020684	NaN	NaN	NaN	NaN	0.008123	-0.008530	0.004874	0.027620	-0.082859	0.054021
2015-02-04	25.03	24.50	24.82	7071100	0.708258	0.008123	0.023083	NaN	NaN	NaN	NaN	0.000000	-0.027397	-0.014504	-0.061241	-0.106769	0.033441

Figure 5.12: DataFrame with historic and forward returns for each stock in our portfolio

9. Print the first 10 rows for the first symbol to inspect the historic returns for each period:

date	high	low	close	volume	vol	return_1d	return_5d	return_10d	return_21d	return_42d	return_63d	target_1d	target_5d	target_10d	target_21d	target_42d	target_63d
2015-01-22	24.69	23.98	24.29	13132700	1.021514	NaN	NaN	NaN	NaN	NaN	NaN	-0.005764	-0.004940	0.021820	0.068753	-0.058872	-0.034582
2015-01-23	24.43	23.67	24.15	12056800	0.976772	-0.005764	NaN	NaN	NaN	NaN	NaN	0.014079	0.041408	-0.004141	0.068737	-0.069151	0.034369
2015-01-26	24.63	23.35	24.49	11189200	1.059908	0.014079	NaN	NaN	NaN	NaN	NaN	0.026541	0.025316	0.004900	0.063699	-0.095141	0.044916
2015-01-27	25.25	24.53	25.14	10774500	0.948081	0.026541	NaN	NaN	NaN	NaN	NaN	-0.035004	-0.020684	-0.029037	0.046539	-0.115752	0.048130
2015-01-28	25.05	24.03	24.26	11251300	0.962112	-0.035004	NaN	NaN	NaN	NaN	NaN	-0.003710	0.023083	-0.004946	0.085326	-0.092333	0.091509
2015-01-29	24.34	23.42	24.17	8850000	1.016856	-0.003710	-0.004940	NaN	NaN	NaN	NaN	0.040546	0.026893	0.014067	0.071163	-0.101779	0.095987
2015-01-30	25.22	23.92	25.15	11594400	1.139382	0.040546	0.041408	NaN	NaN	NaN	NaN	-0.001590	-0.043738	-0.014712	0.018688	-0.096620	0.053280
2015-02-02	25.20	24.56	25.11	7497000	1.109351	-0.001590	0.025316	NaN	NaN	NaN	NaN	-0.019514	-0.019912	-0.036639	0.003186	-0.110315	0.037435
2015-02-03	25.07	24.14	24.62	9038100	0.957277	-0.019514	-0.020684	NaN	NaN	NaN	NaN	0.008123	-0.008530	0.004874	0.027620	-0.082859	0.054021
2015-02-04	25.03	24.50	24.82	7071100	0.708258	0.008123	0.023083	NaN	NaN	NaN	NaN	0.000000	-0.027397	-0.014504	-0.061241	-0.106769	0.033441

Figure 5.13: Slice of the prices DataFrame showing historic returns for stock NEM

10. Finally, use the Seaborn plotting library to visualize how the one-day forward return is related to the normalized Parkinson volatility factor:

```
target = "target_1d"
metric = "vol"
j = sns.jointplot(x=metric, y=target, data=prices)
plt.tight_layout()
df = prices[[metric, target]].dropna()
r, p = spearmanr(df[metric], df[target])
```

The result is the following joint plot, which shows the relationship along with distributions of the values:

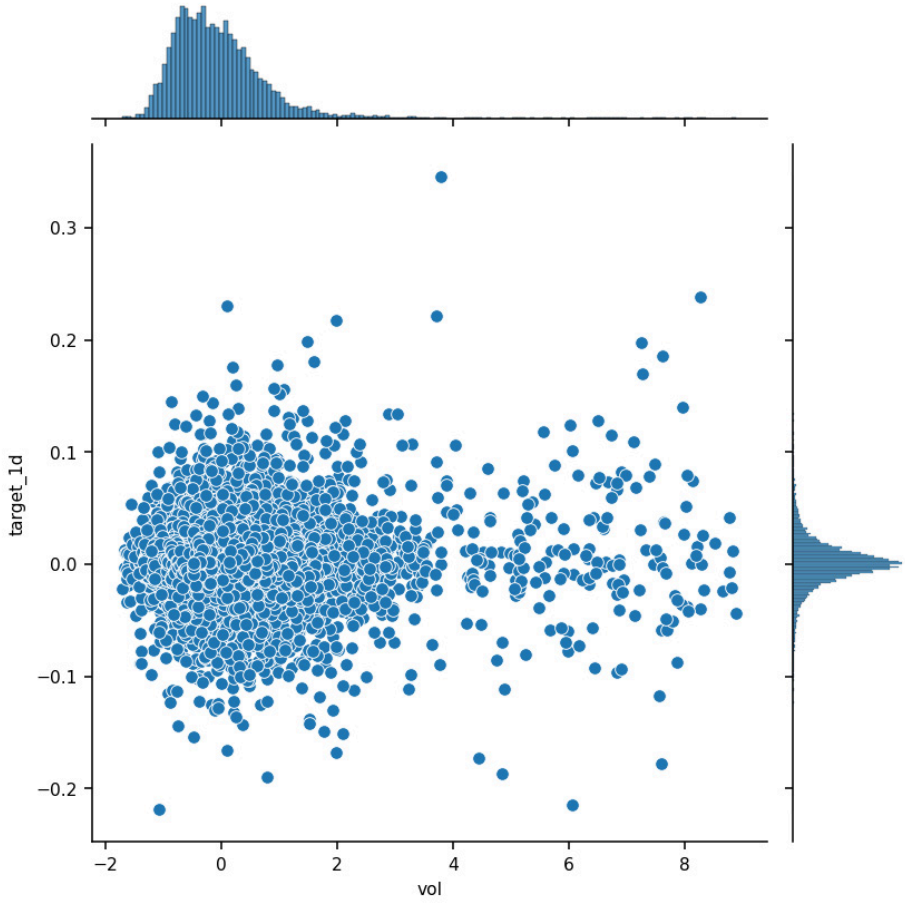

Figure 5.14: Joint plot of normalized Parkinson volatility versus one-day forward return

How it works...

In this recipe, we begin to touch on the data preprocessing required when preparing data for factor analysis.

We modify the DataFrame with the historical data by setting **`symbol`** as a part of a MultiIndex, reordering the levels of the index to have **`symbol`** first, and then removing any duplicate rows. This allows us to apply the normalized Parkinson volatility to each group in the next step. To do it, we group by **`symbol`** and use the **`apply`** method to apply our custom Parkinson volatility function to each chunk of data.

Finally, we compute the historic and forward returns. In the first loop, the code computes historical returns over several periods specified in the **`lags`** list. It does this by grouping the DataFrame by **`symbol`** and then computing the percentage change over each of the specified lags. These historical returns are then added to the DataFrame as new columns with the names **`return_1d`**, **`return_5d`**, and so on.

In the second loop, the code computes future returns for several periods specified in the list. It does this by taking the already computed historical returns (**`return_1d`**, **`return_5d`**, and so on) and shifting them up by the specified number of periods. This essentially assigns to each row the return that will occur **`t`** days in the future. These future returns are then added to the DataFrame as new columns with the names **`target_1d`**, **`target_5d`**, and so on.

There's more...

The **`spearmanr`** function computes the Spearman rank-order correlation coefficient between two data arrays. We can get the rank correlation and *p*-value using the **`spearmanr`** SciPy method:

```
stat, pvalue = spearmanr(df[metric], df[target])
```

The **`stat`** coefficient measures the strength and direction of the relationship between the two variables, with a value between **`-1`** and **`1`**. A value of **`0`** indicates no correlation, **`1`** indicates a perfect positive correlation, and **`-1`** indicates a perfect negative correlation.

In this context, **`stat`** is the correlation between Parkinson volatility and the forward returns which is **`0.0378`**, which is a weak positive correlation.

The **`pvalue`** tests the null hypothesis that the data is uncorrelated. A small **`pvalue`** (typically $\leq$ **`0.05`**) indicates that you can reject the null hypothesis. In this case, the **`pvalue`** is **`0.0158`**, so we reject the null hypothesis, which suggests there is a statistically significant correlation between the Parkinson volatility and the returns of this portfolio.

See also

For more resources on the tools and techniques used in this recipe, visit the following links:

- Walk-through of Parkinson volatility: **https://www.ivolatility.com/help/3.html**
- Average true range: **https://en.wikipedia.org/wiki/Average_true_range**
- Spearman rank correlation: **https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient**

- SciPy's implementation of the Spearman rank correlation coefficient documentation: **https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html**

Preparing a factor ranking model using Zipline Pipelines

Zipline Reloaded is an open source Python library that provides a comprehensive backtesting framework for algorithmic trading. The library provides tools to manage the algorithm's capital, set trading commissions, and simulate orders.

One of the most powerful features of Zipline Reloaded is the Pipeline API. Pipelines let us define factors from columns of data from bundles. Pipelines efficiently process large amounts of data in a single pass while also filtering down to a smaller set of assets of interest. Pipelines allow us to rank a universe of stocks based on the computed factor and output those data in a format suitable for the Alphalens library, which analyzes factor performance and risk. We'll look at Alphalens in ***Chapter 8***, *Evaluate Factor Risk and Performance with Alphalens Reloaded*.

In this recipe, we'll build a data bundle using free stock market data, define a momentum factor, and rank a universe of stocks based on the performance in that factor.

Getting ready

Zipline Reloaded has a built in process to download and bundle free historical market data from Nasdaq Data Link. You can use the Nasdaq Data Link client to acquire data without an account or an API key, but you're limited in the number of API calls you can make. Without a key, you can make 20 API calls in 10 minutes or up to 50 API calls per day. Zipline Reloaded requires an API key to be set before using it.

Since an account is free, it makes sense to sign up. This is something you can do via the Nasdaq Data Link website (https://data.nasdaq.com/). Your API key is in your profile (https:// data.nasdaq.com/account/profile).

After you get your API key set up, install Zipline Reloaded. The steps differ depending on your operating system.

For Windows, Unix/Linux, and Mac Intel users

If you're running on an Intel x86 chip, you can use **conda**:

```
conda install -c conda-forge zipline-reloaded pyfolio-reloaded alphalens-reloaded -y
```

For Apple Silicon users

If you have a Mac with an M-series chip, you need to install some dependencies first. The easiest way is to use Homebrew (**https://brew.sh**).

Install the dependencies with Homebrew:

```
brew install freetype pkg-config gcc openssl hdf5 ta-lib
```

Install the Python dependencies with **conda**:

```
conda install -c conda-forge pytables h5py -y
```

Install the Zipline Reloaded ecosystem:

```
pip install zipline-reloaded pyfolio-reloaded alphalens-reloaded
```

In this example, we'll use the free data bundle provided by Nasdaq Data Link. This dataset has market price data on 3,000 stocks through 2018.

The free dataset is great for getting up and running but it is limited. Once you start hitting the limitations of the free data, you can either pay for a premium dataset or ingest your own data. Make sure you have your Nasdaq Data Link API key handy.

How to do it...

To build the analysis, we'll import the required Zipline Reloaded modules.

1. Import the libraries we need for the analysis:

```
import os
import numpy as np
import pandas as pd
from zipline.data.bundles.core import load
from zipline.pipeline import Pipeline
from zipline.pipeline.data import USEquityPricing
```

```
from zipline.pipeline.engine import SimplePipelineEngine
from zipline.pipeline.factors import AverageDollarVolume, CustomFactor, Returns
from zipline.pipeline.loaders import USEquityPricingLoader
```

2. Set your API key as an environment variable and load the free Nasdaq data into a bundle:

```
os.environ["QUANDL_API_KEY"] = "YOUR_API_KEY"
bundle_data = load("quandl", os.environ, None)
```

3. Build a US equity pricing loader:

```
pipeline_loader = USEquityPricingLoader(
    bundle_data.equity_daily_bar_reader,
    bundle_data.adjustment_reader,
    fx_reader=None
)
```

4. Use the pricing loader to create a Pipeline engine:

```
engine = SimplePipelineEngine(
    get_loader=lambda col: pipeline_loader,
    asset_finder=bundle_data.asset_finder
)
```

5. Implement a custom momentum factor that returns a measure of price momentum:

```
class MomentumFactor(CustomFactor):
    inputs = [USEquityPricing.close, Returns(window_length=126)]
    window_length = 252
    def compute(self, today, assets, out, prices, returns):
        out[:] = (
            (prices[-21] -prices[-252]) / prices[-252]
            - (prices[-1] - prices[-21]) / prices[-21]
        ) / np.nanstd(returns, axis=0)
```

6. Create a function that instantiates the custom momentum factor, builds a filter for average dollar volume over the last 30 days, and returns a Pipeline:

```
def make_pipeline():
    momentum = MomentumFactor()
    dollar_volume = AverageDollarVolume(
        window_length=30)
    return Pipeline(
        columns={
            "factor": momentum,
            "longs": momentum.top(50),
            "shorts": momentum.bottom(50),
            "rank": momentum.rank()
        },
```

```
            screen=dollar_volume.top(100)
        )
```

7. Run the Pipeline:

```
results = engine.run_pipeline(
    make_pipeline(),
    pd.to_datetime("2012-01-04"),
    pd.to_datetime("2012-03-01")
)
```

8. Clean up the resulting DataFrame by removing records with no factor data, adding names to the MultiIndex and sorting the values first by date and then by factor value:

```
results.dropna(subset="factor", inplace=True)
results.index.names = ["date", "symbol"]
results.sort_values(by=["date", "factor"], inplace=True)
```

The result is a MultiIndex DataFrame including the raw factor value, Boolean values indicating a short or long position, and how the stock's factor value is ranked among the universe:

		factor	longs	shorts	rank
date	**symbol**				
2012-01-04	**Equity(300 [BAC])**	-2.522045	False	False	165.0
	Equity(1264 [GS])	-2.215784	False	False	220.0
	Equity(1888 [MS])	-2.204802	False	False	225.0
	Equity(1894 [MSFT])	-1.949654	False	False	295.0
	Equity(457 [C])	-1.830819	False	False	345.0
...	**...**	...	...	...	...
2012-03-01	**Equity(3105 [WMT])**	3.409414	False	False	2607.0
	Equity(1690 [LLY])	3.809608	False	False	2642.0
	Equity(399 [BMY])	4.689588	True	False	2685.0
	Equity(1770 [MCD])	4.816880	True	False	2691.0
	Equity(1789 [MDLZ])	5.680276	True	False	2706.0

Figure 5.15: MultiIndex DataFrame with factor information and trading indicators

How it works...

The first step is to initialize a Pipeline loader for US equities. **`USEquityPricingLoader`** is responsible for loading pricing and adjustment data for US equities from the specified data sources. The **`SimplePipelineEngine`** class is used to run a pipeline. The **`get_loader`** pa-

rameter is a function that returns a loader to be used to load the data needed for the pipeline. In this case, it's a lambda function that always returns the previously defined **`pipeline_loader`**. The **`asset_finder`** parameter is used to do asset lookups. Here, it is set to the **`asset_finder`** of **`bundle_data`**, which is an object that knows how to look up asset metadata for the bundle.

Our code defines a custom factor, **`MomentumFactor`**. This factor computes momentum as the percentage change in price over the first 126 days of the 252-day window minus the percentage change in price over the last 21 days, divided by the standard deviation of the returns over the 252-day window. **`inputs`** is the data that the factor needs: the close prices of the US equities and the returns over a 126-day window. **`window_length`** is the number of days of data that the **`compute`** method will receive. The **`compute`** method is where the actual computation of the factor is done. The **`out`** array is where the computed factor values are stored. The **`[:]`** after **`out`** is used to modify the **`out`** array in place.

Next, we implement a function called **`make_pipeline`** that brings the factor, the universe screener, and Pipeline together. The **`MomentumFactor`** class is instantiated to compute momentum, and the **`AverageDollarVolume`** function is used with a 30-day window to calculate average dollar volume. The pipeline includes columns for the momentum factor, Boolean masks for selecting the top and bottom 50 securities based on momentum, and the momentum rank across all securities. Additionally, the pipeline employs a screen to focus on the top 100 securities by dollar volume.

There's more...

Zipline Reloaded comes with many built-in factor classes that can be used within trading algorithms. Some of the more commonly used algorithms are as follows:

- **`ExponentialWeightedMovingAverage`**: Computes the exponential weighted moving average over a specified window length
- **`BollingerBands`**: Computes the Bollinger Bands over a specified window length
- **`VWAP`**: Computes the volume-weighted average price over a specified window length
- **`AnnualizedVolatility`**: Computes the annualized volatility of an asset over a specified window length

- **`MaxDrawdown`**: Computes the maximum drawdown of an asset over a specified window length

See also

To learn more about Zipline Reloaded, see the documentation here: **https://zipline.ml4trading.io**.