3

Visualize Financial Market Data with Matplotlib, Seaborn, and Plotly Dash

The first step when working with data is to visualize and explore it. This is especially true when dealing with financial market data we rely on for trading. This chapter sets the stage by introducing five powerful data visualization techniques: pandas, Matplotlib, Seaborn, Plotly, and Plotly Dash.

Each tool has pros and cons and should be selected depending on the use case. pandas has built-in plotting functionality using both Matplotlib and Plotly to render the charts. Matplotlib offers advanced functionality for building 3-dimensional surfaces and animated charts. Seaborn offers an array of statistical data visualizations. Plotly works with JavaScript for interactive charting. Plotly Dash is a framework for building interactive web apps with Python.

By the end of the chapter, you'll have a wide range of tools and chart types to visually inspect the financial market data required to research and build algorithmic trading applications.

In this chapter, we present the following recipes:

- Quickly visualizing data using pandas
- Animating the evolution of the yield curve with Matplotlib

- Plotting options implied volatility surfaces with Matplotlib
- Visualizing statistical relationships with Seaborn
- Creating an interactive PCA analytics dashboard with Plotly Dash

Quickly visualizing data using pandas

pandas is an all-purpose data manipulation library. Not only can you use it for data acquisition and manipulation as we saw in ***Chapter 1***, *Acquire Free Financial Market Data with Cutting-edge Python Libraries* and ***Chapter 2***, *Analyze and Transform Financial Market Data with pandas*, but you can use it for plotting too. pandas offers various "backends" that are used while plotting through a common method. In this recipe, you'll learn how to use the default backend, Matplotlib, to quickly plot financial market data using a line plot, bar chart, histogram, and others.

How to do it...

You can use the Matplotlib plots through pandas by importing them.

1. Import the libraries:

```
import matplotlib as plt
import pandas as pd
from openbb import obb
from pandas.plotting import bootstrap_plot, scatter_matrix
obb.user.preferences.output_type = "dataframe"
```

2. Download stock price data:

```
df = obb.equity.price.historical("AAPL", provider="yfinance")
```

3. Create a line chart that plots the closing price:

```
df.close.plot()
```

   The result is the following chart:

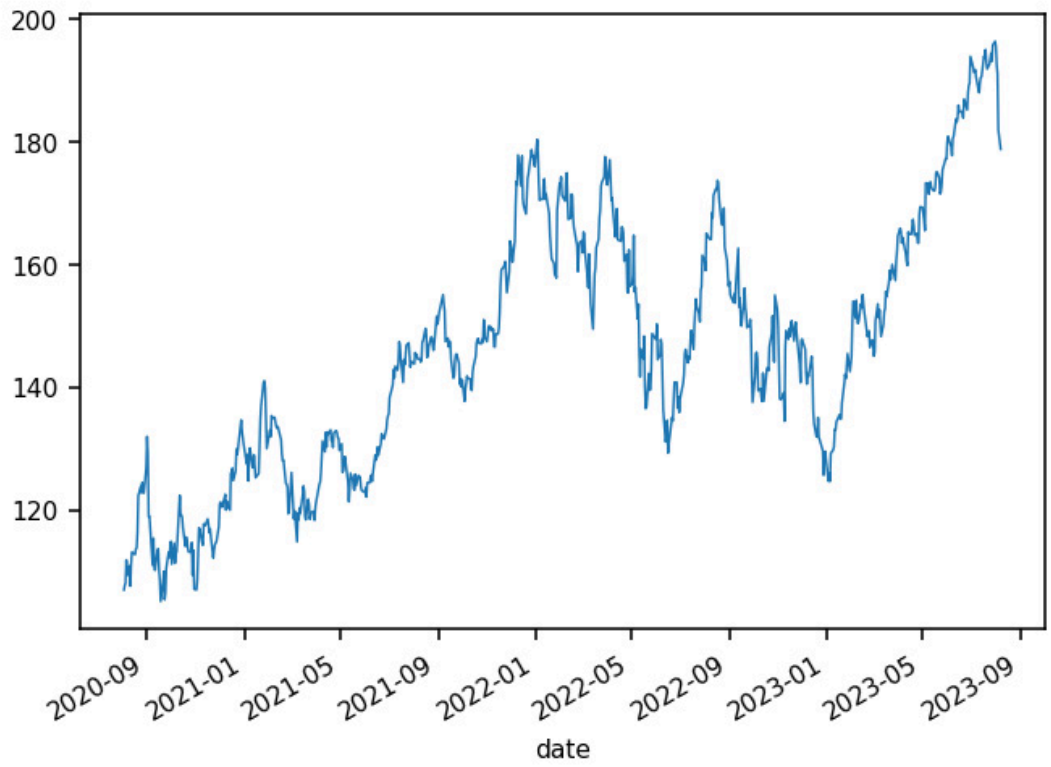

Figure 3.1: Line chart plotting AAPL's unadjusted closing price between 2020 and 2023.

4. Plot the daily returns as a bar chart using additional options to style the chart:

```
returns = df.close.pct_change()
returns.name = "return"
returns.plot.bar(
    title="AAPL returns",
    grid=False,
    legend=True,
    xticks=[]
)
```

The result is the following chart:

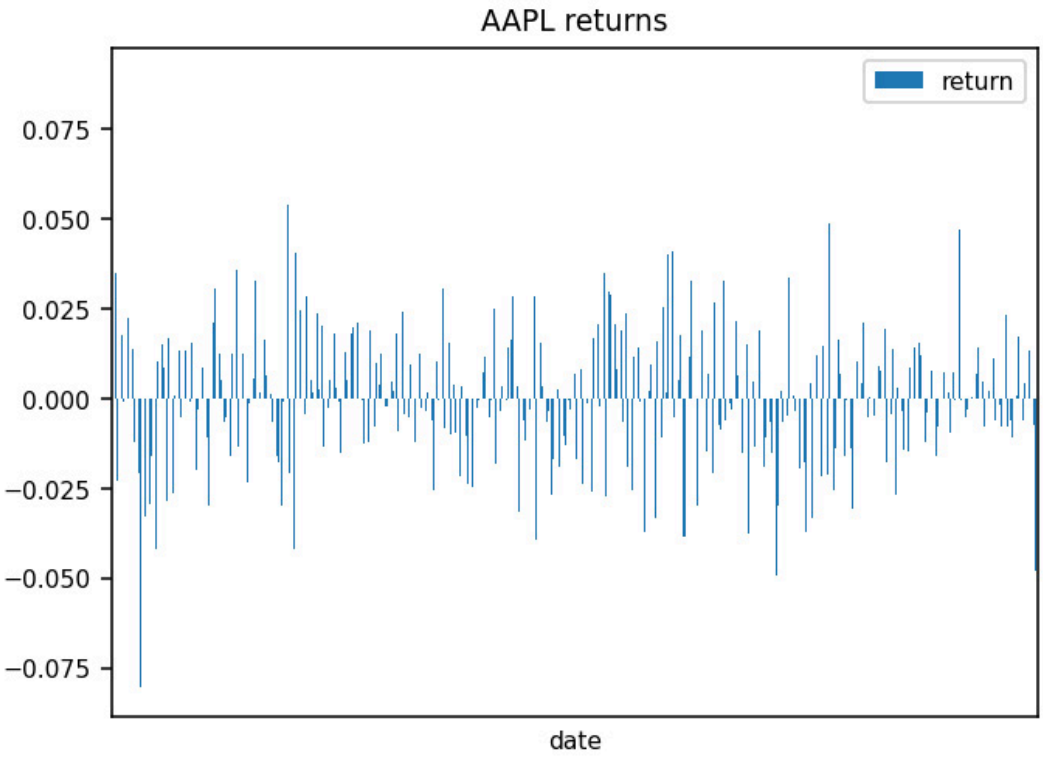

Figure 3.2: Bar chart plotting AAPL's daily returns without the x-axis labels, grid, or legend.

5. Create a histogram of returns with 50 bins:

```
returns.plot.hist(bins=50)
```

The result is the following chart:

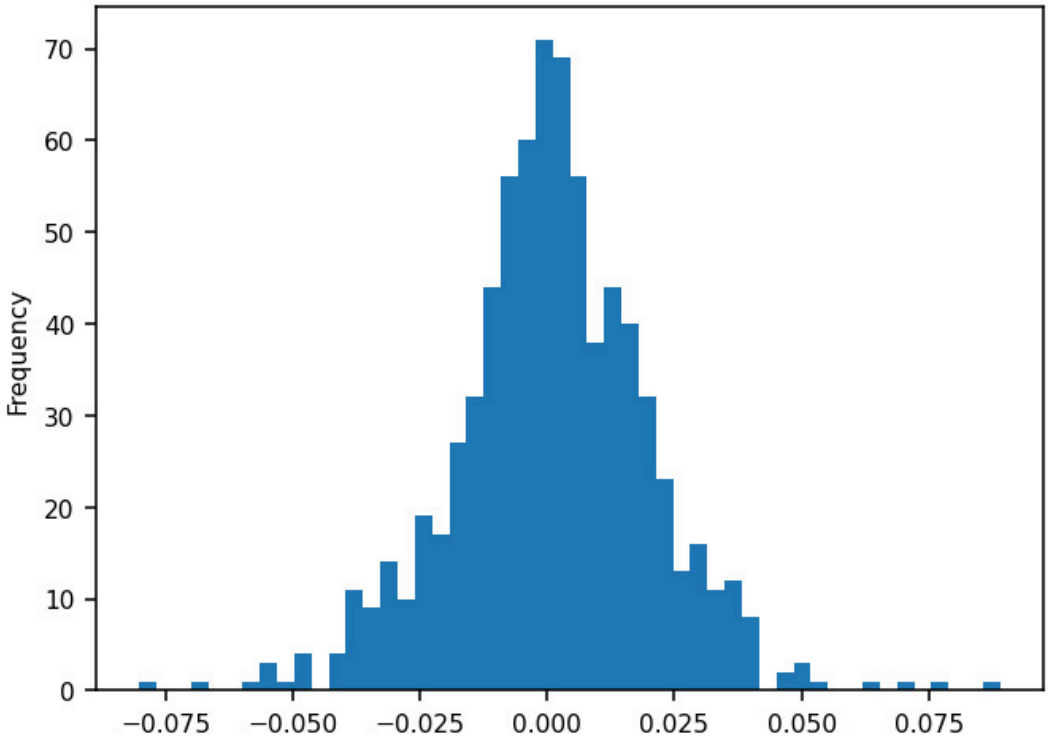

Figure 3.3: Histogram of AAPL daily returns with 50 bins.

6. Create a box-and-whisker plot:

```
returns.plot.box()
```

The result is the following chart:

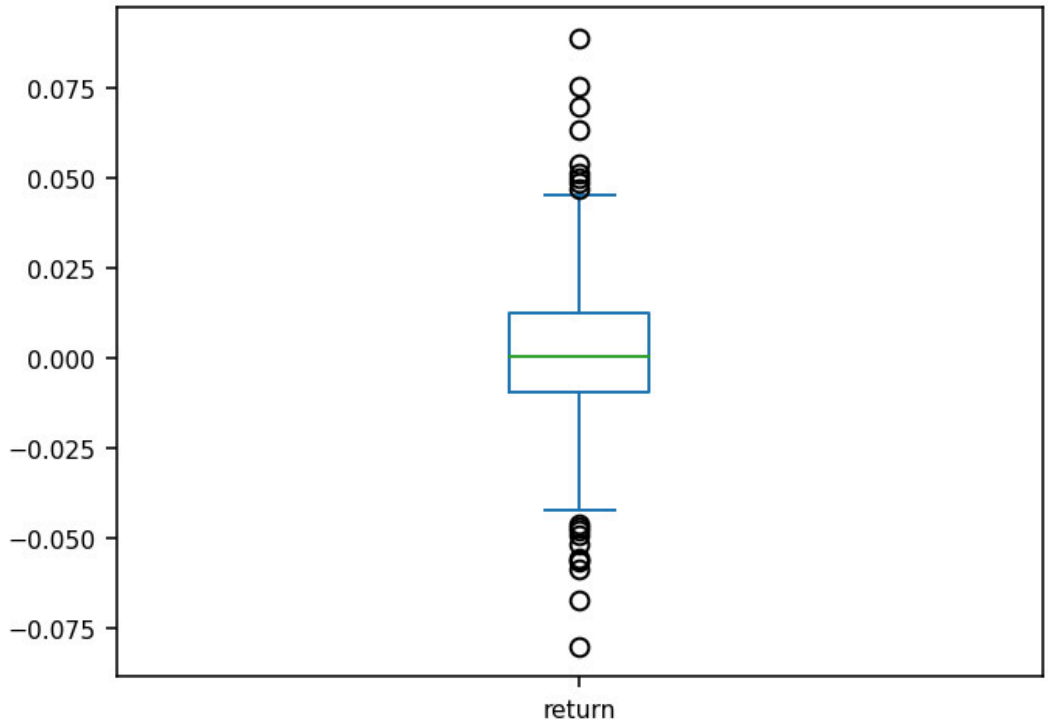

Figure 3.4: Box-and-whisker plot showing the median, quartiles, and outliers of AAPL daily returns.

> *IMPORTANT*

> *A box plot serves as a graphical representation for displaying numerical data distributions via their quartiles. The box itself spans from the first quartile (Q1) to the third quartile (Q3), with a line indicating the median (Q2). Whiskers extend from the box boundaries to illustrate the data's range, and their default position is determined by 1.5 times the interquartile range (IQR), where IQR equals Q3 minus Q1. Data points beyond the whiskers are considered outliers.*

How it works…

The pandas **`plot`** method lets you create various types of plots using DataFrames and Series. Under the hood, pandas use the defined backend (Matplotlib by default) to generate these visualizations. To use this method, you first create a DataFrame or Series, then call the **`plot`** method on the object, optionally specifying the type of plot and other parameters that control the plot's appearance such as color, size, title, and axes labels.

> *IMPORTANT*

> *Using the* **`plot`** *method, we can generate bar plots, density plots, scatter plots, and many others. We can define the type of plot using the* **`kind`** *parameter. Not all backends support all plot types. For example, the plot type* **`hexbin`** *does not work with the Plotly backend. See the pandas documentation for details.*

There's more…

A common step in quantitative portfolio construction and risk management is analyzing the relationship between two or more assets.

Scatter plots are a type of visual representation that can be used to explore the relationship between two stocks. Each dot on the scatter plot represents a pair of values for the two stocks at a specific point in time. The x-coordinate of the dot represents the value of one stock, and the y-coordinate represents the value of the other stock.

If the dots form a pattern that slants upwards from left to right, it suggests a positive correlation between the stocks, meaning as one stock's price increases, the other's tends to as well. If the plot forms a pattern slanting downwards, it suggests a negative correlation, meaning as one stock's price increases, the other's tends to decrease. If the dots appear randomly scattered with no discernible pattern, it indicates no or a weak correlation.

Compare AAPL with the Nasdaq tracking ETF, QQQ.

```
qqq = obb.equity.price.historical("QQQ", provider="yfinance")
qqq_returns = qqq.close.pct_change()
asset_bench = pd.concat([returns, qqq_returns], axis=1)
asset_bench.columns = ["AAPL", "QQQ"]
asset_bench.plot.scatter(x="QQQ", y="AAPL", s=0.25)
```

The result is a scatter plot that shows the seemingly positive relationship between AAPL returns and QQQ returns.

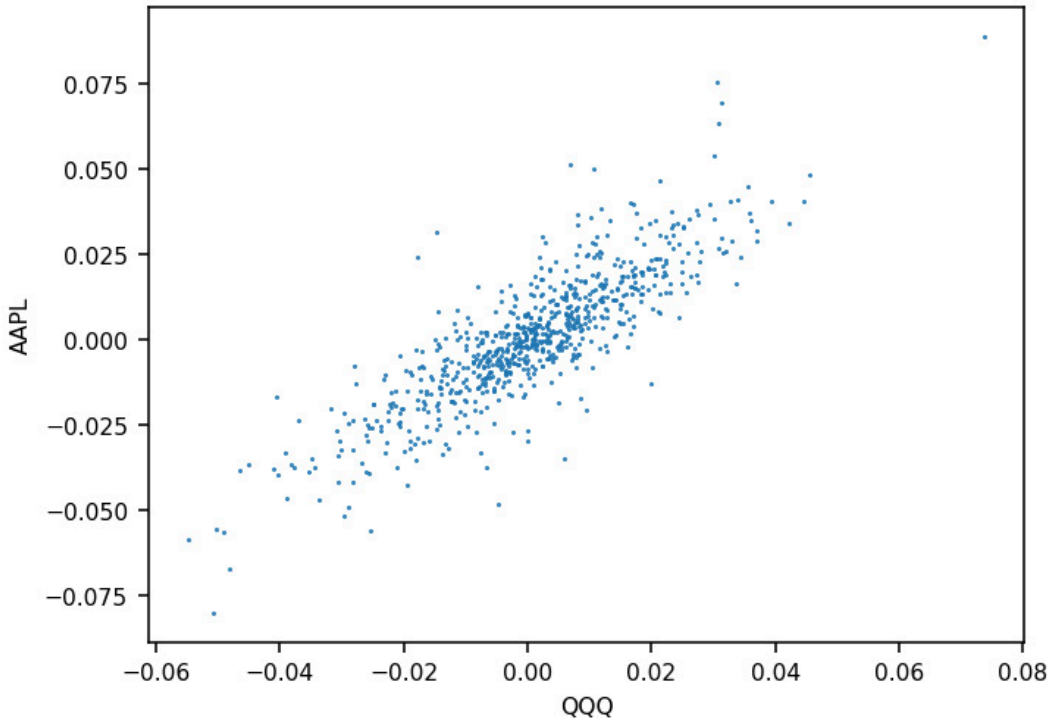

Figure 3.5: Scatter plot of AAPL daily returns and QQQ daily returns showing a positive linear relationship.

The pandas **`scatter_matrix`** function visualizes pairwise relationships. It generates a matrix of scatter plots, each plotting a pair of columns against each other. This allows for a quick visual inspection of potential correlations or patterns within your data. Additionally, the main diagonal (from top left to bottom right) shows the histogram of each column, which helps to visualize data distribution.

```
scatter_matrix(asset_bench)
```

The result is the scatter matrix.

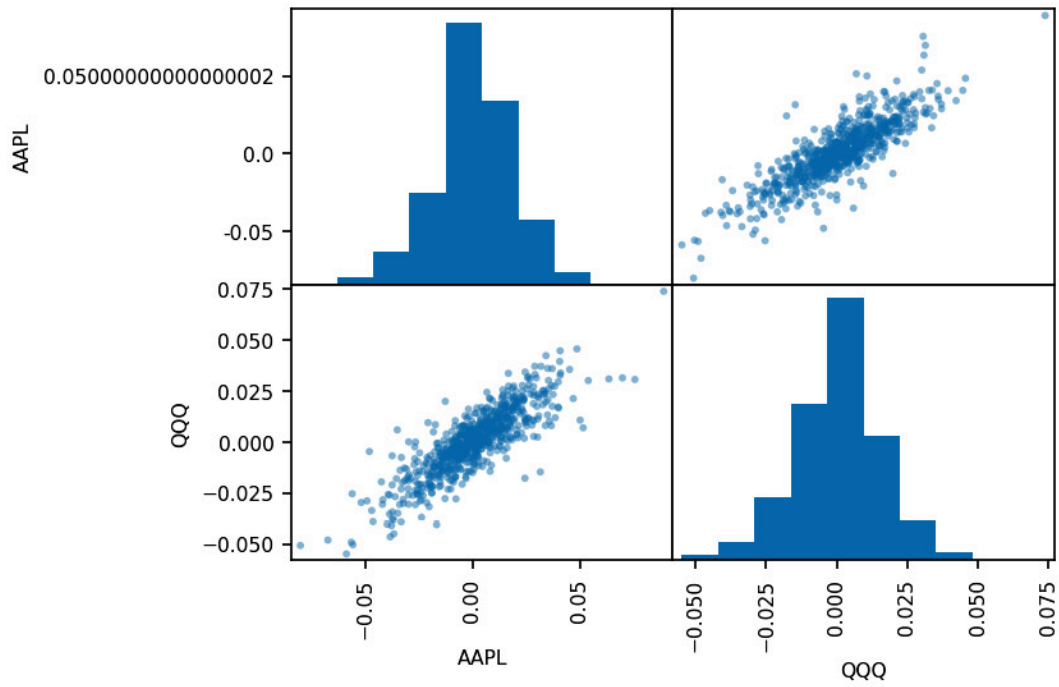

Figure 3.6: A scatter matrix plot summarizing AAPL and QQQ returns in one chart.

Bootstrap plots serve to graphically evaluate the variability associated with a particular statistic, including the mean, median, and midrange. A designated subset size is randomly sampled from the data set, and the target statistic is calculated for this subset. This procedure is iteratively performed a predetermined number of times. The plots and histograms collectively form the bootstrap plot.

```
bootstrap_plot(returns)
```

By default, **`bootstrap_plot`** selects 50 data points to consider during each sampling. The result is a **`bootstrap_plot`**.

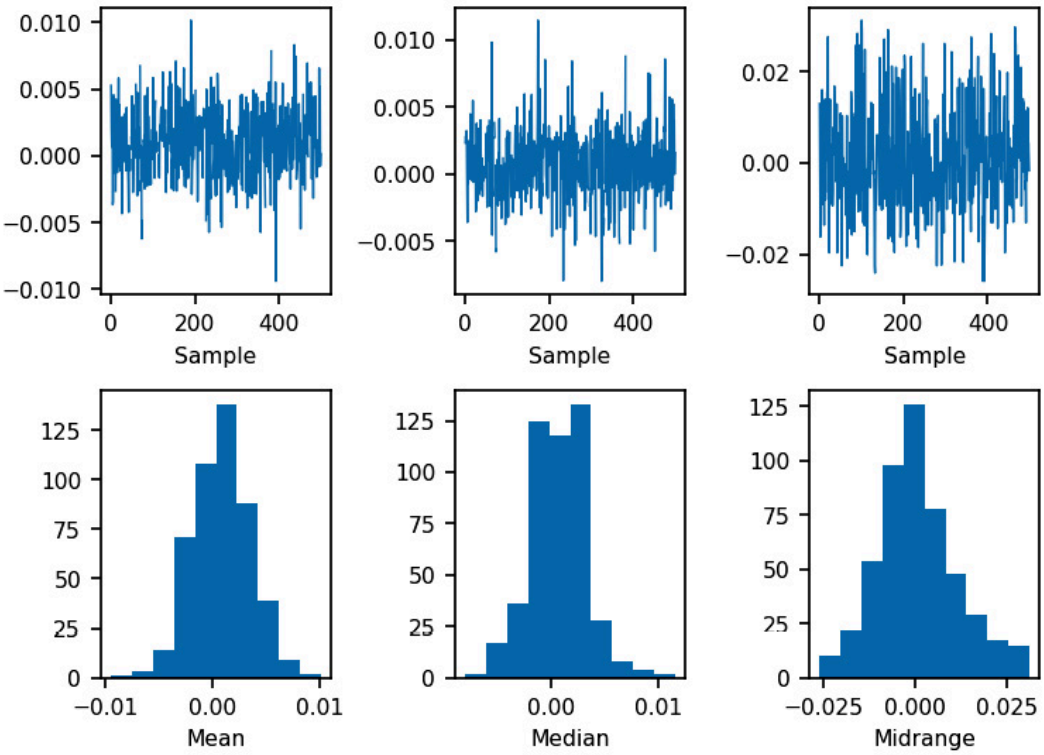

Figure 3.7: A bootstrap plot that depicts random subsets of data.

See also

pandas has detailed documentation on its plots. You can read more here:

- pandas plot documentation: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html**.
- pandas **`scatter_matrix`** documentation: **https://pandas.pydata.org/docs/reference/api/pandas.plotting.scatter_matrix.html**.
- pandas **`bootstrap_plot`** documentation: **https://pandas.pydata.org/docs/reference/api/pandas.plotting.bootstrap_plot.html**.

To see more practical examples of using pandas plotting, you can check out past issues of the PyQuant Newsletter here: **https://www.pyquantnews.com/past-pyquant-newsletter-issues**.

Animating the evolution of the yield curve with Matplotlib

Even though pandas use Matplotlib as its backend, there are times you need to go deeper. One such case is when you want to visualize the change in data through time—like when analyzing the evolution of the yield curve. The yield curve, which charts the yields of bonds of the same quality across different maturities, typically slopes upward. This means that longer-term bonds have higher yields than shorter-term bonds, which makes sense given the additional risks associated with holding a bond for a longer time (e.g., inflation, higher interest rate volatility). However, there are times when the yield curve inverts, meaning that shorter-term bonds yield more than longer-term ones. Many traders and economists view an inverted yield curve as a precursor to a recession.

An inverted yield curve has historically preceded U.S. recessions, suggesting traders' anticipation of lower future interest rates and a coming economic slowdown. The inversion can constrict bank profitability, leading to reduced lending and a potential economic deceleration. Additionally, the expectation of a recession can become a self-fulfilling prophecy as businesses and consumers curtail spending. Investors may also shift towards safer assets, limiting funding for riskier ventures.

How to do it...

Creating an animated plot requires a specialized function called **`animation`** which is imported from Matplotlib.

1. Import the libraries:

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation
```

```
from mpl_toolkits.mplot3d import Axes3D
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Create a list of maturities and download the data:

```
maturities = ["3m", "6m", "1y", "2y", "3y", "5y", "7y", "10y",
    "30y"]
data = obb.fixedincome.government.treasury_rates(
    start_date="1985-01-01",
    provider="federal_reserve",
).dropna(how="all").drop(columns=["month_1", "year_20"])
data.columns = maturities
```

3. Use boolean indexing to mark where the yield curve is inverted:

```
data["inverted"] = data["30y"] < data["3m"]
```

4. Initialize the figure:

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
line, = ax.plot([], [])
```

5. Set the range of ticks

```
ax.set_xlim(0, 8)
ax.set_ylim(0, 20)
```

6. Set the tick locations:

```
ax.set_xticks(range(9))
ax.set_yticks([2, 4, 6, 8, 10, 12, 14, 16, 18])
```

7. Set the axis labels:

```
ax.set_xticklabels(maturities)
ax.set_yticklabels([2, 4, 6, 8, 10, 12, 14, 16, 18])
```

8. Force the y-axis labels to the left:

```
ax.yaxis.set_label_position("left")
ax.yaxis.tick_left()
```

9. Add the axis labels:

```
plt.ylabel("Yield (%)")
plt.xlabel("Time to maturity")
plt.title("U.S. Treasury Bond Yield Curve")
```

10. Create the function that is run when the animation is initialized:

```
def init_func():
    line.set_data([], [])
    return line,
```

11. Create the function that runs at each iteration through the data:

```
def animate(i):
    x = range(0, len(maturities))
    y = data[maturities].iloc[i]
    dt_ = data.index[i].strftime("%Y-%m-%d")
    if data.inverted.iloc[i]:
        line.set_color("r")
    else:
        line.set_color("y")
    line.set_data(x, y)
    plt.title(f"U.S. Treasury Bond Yield Curve ({dt_})")
    return line,
```

12. Generate the animation that brings it all together:

```
ani = animation.FuncAnimation(
    fig,
    animate,
    init_func=init_func,
    frames=len(data.index),
    interval=250,
    blit=True
)
```

13. Persist the plot to give the script time to update and display the chart:

```
plt.show()
```

    The result is an animated, interactive chart that displays the shape of the yield curve through time.

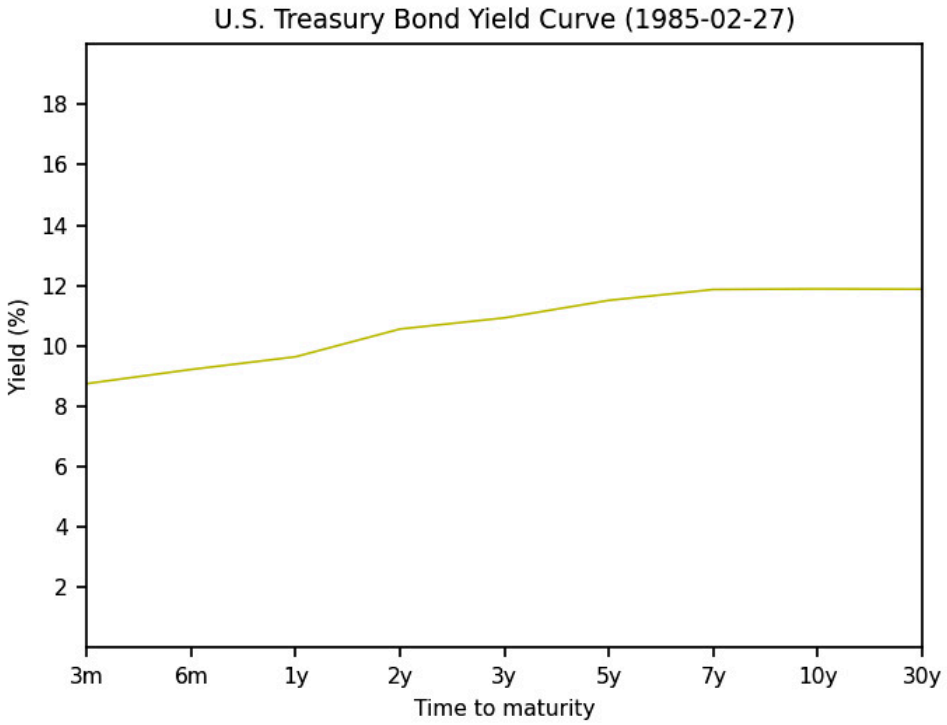

Figure 3.8: One frame from the animated plot of the US yield curve. This frame depicts the shape of the yield curve on 27 February 1985.

How it works...

A plot is prepared with maturity times on the horizontal axis and yield values on the vertical axis. A line plot is then set up to display the yield data.

The initialization function (**`init_func`**) establishes the starting frame of the animation. The main animation function (**`animate`**) is where the data gets updated. It redraws the line with the yields for each bond maturity at different time points. If the data indicates an inverted yield curve on a specific date, the line color changes to red, otherwise, it remains yellow.

The actual animation is put together using the **`FuncAnimation`** class. The class calls the animate function at regular intervals of 250 milliseconds. Using blitting helps make the animation more efficient by only redrawing the parts of the plot which change. The show command at the end displays the animation.

There's more...

`FuncAnimation` is a versatile class. Apart from the arguments used above, it also accepts the following:

1. `repeat`: A boolean value indicating if the animation should repeat once all frames have been displayed.
2. `repeat_delay`: The delay, in milliseconds, between consecutive repetitions of the animation.
3. `fargs`: Any additional arguments to pass to the `func` (i.e., the animation function).
4. `save_count`: The number of values from frames to cache, to improve performance when saving the animation. If `None`, then the entirety of `frames` is cached.
5. `cache_frame_data`: If `True`, the return values of the `animate` function are cached, which can be useful for speeding up the saving of long animations. If you set it to `False`, the frames will be recreated via the animation function during the save process.
6. `event_source`: An instance of `EventSource` or `None`. If `None`, a new instance of `TimerBase` will be created. This is useful when you want to synchronize multiple animations.

These are some of the main arguments you can use with `FuncAnimation`. There are also some internal and base class arguments available, but they're less commonly used in most applications.

See also

The animation API is very rich offering many animation options. You can read more here:

- https://matplotlib.org/stable/api/animation_api.html — Documentation for Matplotlib's animation API.

Plotting options implied volatility surfaces with Matplotlib

Traders use Matplotlib to visualize complex data, such as **options implied volatility surfaces**. These visuals help understand how implied volatility of options changes with different expiration dates and strike prices. Implied volatility surfaces are important for traders for information on the market's expectations of future volatility.

These surfaces show two main features: skew and term structure. Skew refers to how implied volatility varies at different strike prices for the same expiration date. It can indicate the market's expectation of significant price shifts. Term structure shows how implied volatility changes for options with the same strike price but different expiration dates. Term structure shows how volatility is expected to evolve over time.

Although a detailed explanation of skew and term structure is beyond the scope of this book, it's important to note these aspects of the volatility surface are important for making informed trading decisions. They also play a significant role in selecting the right options for hedging and in fine-tuning the pricing and risk assessment models for more complex financial instruments.

Getting ready...

We'll start with a fresh Jupyter Notebook to avoid plotting the surface plot on the same axis as the animated plot.

How to do it...

As with the animated charts, to create three dimensional surfaces, we need to import the `Axes3D` class from the `mpl_toolkits` module.

1. Import the libraries:

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download options data using the OpenBB SDK:

```
chains = obb.derivatives.options.chains(
    "AAPL",
    provider="cboe",
)
```

3. Filter the calls with days to expiration under three months and with a strike price greater than 100. Then drop any duplicates:

```
calls = calls[
    (calls.dte < 100)
    & (calls.strike >= 100)
]
calls.drop_duplicates(subset=["strike", "dte"], keep=False,
    inplace=True)
```

4. Pivot the DataFrame to put strikes in the index, days to expiration along the columns, and the implied volatility within the cells. Then drop rows where all values in a column is nan:

```
vol_surface = (
    calls
    .pivot(
        index="strike",
        columns="dte",
        values="implied_volatility"
    )
    .dropna(how="all", axis=1)
)
```

5. Use NumPy's meshgrid method to create a two-dimensional grid using the strike price and days to expiration for use in the plot:

```
strike, dte = np.meshgrid(
    vol_surface.columns,
    vol_surface.index
)
```

6. Finally, plot the surface:

```
fig = plt.figure(figsize=(15, 15))
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel("Days to Expiration")
ax.set_ylabel("Strike Price")
ax.set_zlabel("Implied Volatility")
ax.plot_surface(
    strike,
    dte,
    vol_surface.values,
    cmap="viridis"
)
```

The result is a three-dimensional surface plot with the strike price on the x-axis, the days to expiration on the y-axis, and the implied volatility on the z-axis:

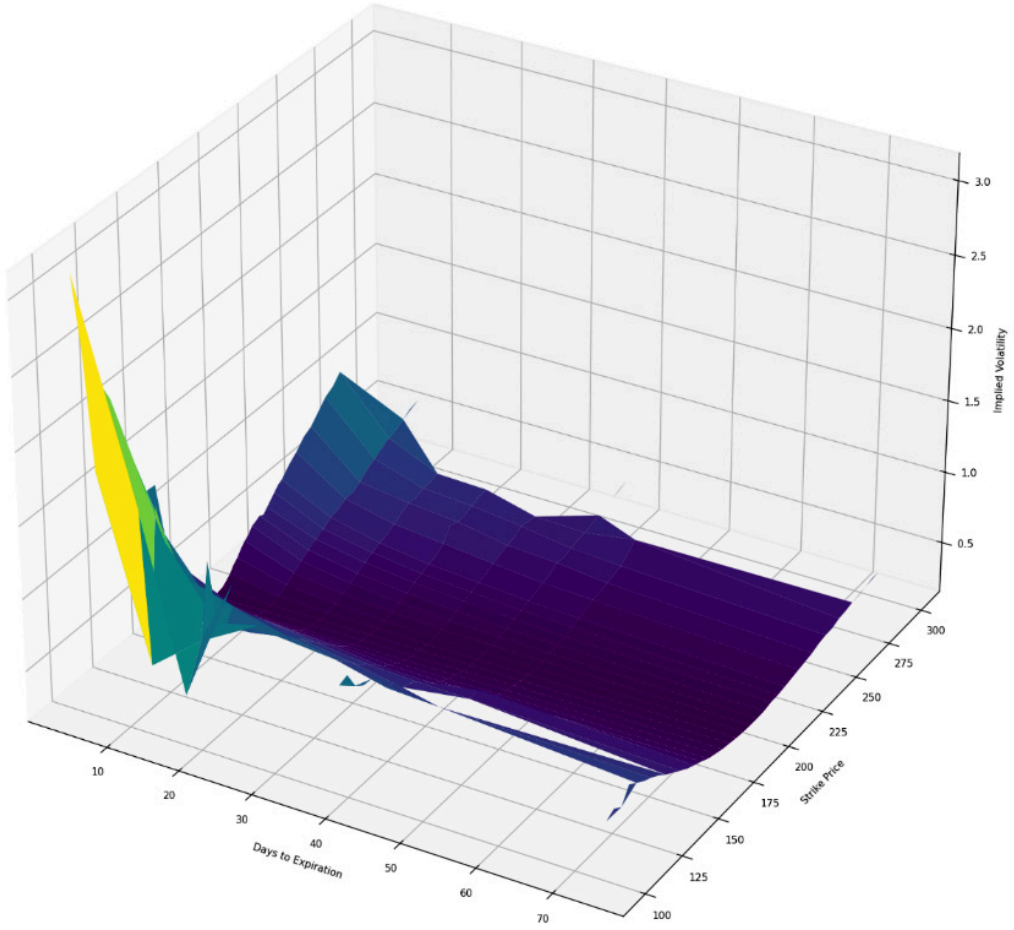

Figure 3.9: Options implied volatility surface showing for AAPL call options.

How it works...

We download options data and narrow down the dataset to options expiring in less than 90 days with a strike price of $100 or more. It then cleans the data by removing any entries with identical strike prices and expiration periods to ensure uniqueness. The core of the process involves reorganizing the data into a DataFrame where each row and column represent different strike prices and expiration times. We then use the NumPy **`meshgrid`** method to build two-dimensional grids corresponding to the strike prices and expiration times. The grid is the same structure as the **`vol_surface`** DataFrame. The culmination of this process is the generation of a 3D plot, where the axes denote the strike price, days to expiration, and implied volatility, respectively.

There's more...

The `plot_surface` method of Matplotlib's 3D axes provides an easy way to create three-dimensional surface plots. In addition to the arguments we used in the preceding section, there are several other arguments commonly used in algorithmic trading:

1. `rstride` and `cstride`: These arguments control the stride (step size) used to create the surface plot. The `rstride` and `cstride` parameters set the stride size for row and column data, respectively.
2. `color`: A single color that can be used to color the entire surface.
3. `facecolors`: A matrix of the same size as Z that provides the colors for each face of the surface plot.
4. `linewidth`: The linewidth for the wireframe is drawn on the surface plot. The default is 0, meaning no wireframe.
5. `antialiased`: If set to `True`, the surface will be antialiased (smoothed).
6. `shade`: If `True`, the surface plot will be shaded, giving it a gradient effect based on light source and orientation.
7. `vmin` and `vmax`: The `colorbar` range. If either is `None`, it will be computed using `min(Z)` or `max(Z)` respectively.
8. `facecolors`: The face colors of the individual patches of the surface plot.

See also

To learn more about plotting , yield curves, and implied volatility surfaces, see the below resources:

- Documentation for Matplotlib's `plot_surface` method: **https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.mplot3d.axes3d.Axes3D.plot_surface.html**.
- More about the history of inverted yield curves: **https://www.investopedia.com/terms/i/invertedyieldcurve.asp**.

- More about how the implied volatility surface is used in practice: https://www.investopedia.com/articles/stock-analysis/081916/volatility-surface-explained.asp.

Visualizing statistical relationships with Seaborn

A major part of algorithmic trading is engineering factors. Factor engineering involves creating predictors for algorithmic trading models, often called **alpha factors**. These factors represent patterns or anomalies in the market. They aim to predict future price movements based on historical and real-time data. Factor engineering is increasingly reliant on machine learning and statistical methods. These tools help in automatically extracting patterns from vast datasets. Machine learning models, such as neural networks, can identify non-linear relationships missed by traditional methods. Feature selection techniques aid in determining the most relevant predictors. Regularization techniques prevent overfitting, ensuring model robustness. Clustering and dimensionality reduction help manage complex datasets.

The `seaborn` library is tailor-made for visualizing statistical relationships, making it an important tool for factor engineering. Seaborn is built on Matplotlib and integrates with pandas making it familiar to users of those tools (which we now are).

How to do it...

We'll import Seaborn for the statistical plots.

1. Import the libraries:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

```
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download stock price history using the OpenBB SDK:

```
data = obb.equity.price.historical(
    ["AAPL", "SPY"],
    start_date="2020-01-01",
    provider="yfinance"
).pivot(columns="symbol", values="close")
```

3. Compute the daily returns:

```
returns = (
    data
    .pct_change(fill_method=None)
    .dropna()
)
```

4. Reshape the data to a long format using the pandas melt method:

```
returns = returns.reset_index()
melted = pd.melt(
    returns,
    id_vars=["date"],
    value_vars=["AAPL"],
    var_name="stock",
    value_name="returns",
)
```

5. Add a new column for the month:

```
melted["month"] = melted["date"].dt.to_period("M")
```

6. Generate the box plot:

```
g = sns.boxplot(
    x="month",
    y="returns",
    hue="stock",
    data=melted
)
g.set_xticklabels(
    melted["month"].unique(),
```

```
    rotation=45
)
```

The result is a box plot summarizing the monthly returns:

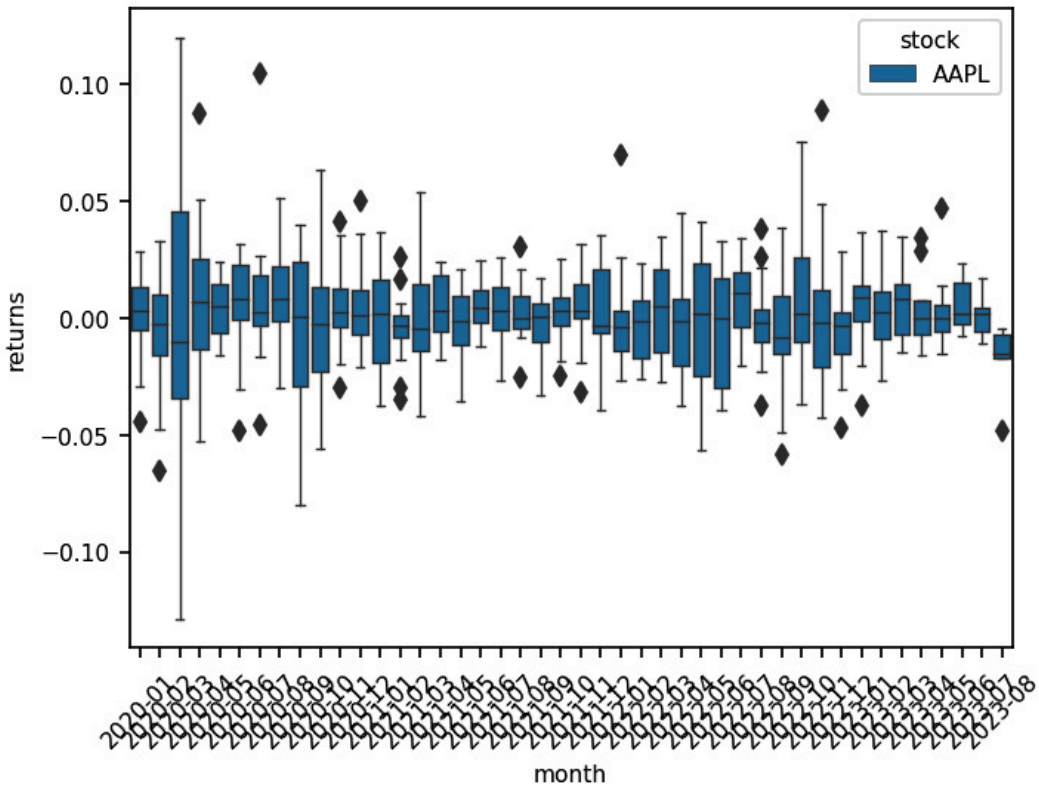

Figure 3.10: Box plot showing monthly summary statistics for AAPL returns.

How it works...

The code computes daily percentage changes in AAPL's closing prices and removes any **`nan`** values. This Series of returns is then transformed into a DataFrame with two columns: the date and the respective return for that date. Using **`melt`**, the DataFrame is reshaped to be compatible with Seaborn's boxplot function. It also extracts the month from each date and adds it as a new column. Lastly, the **`boxplot`** method creates a box plot with months on the x-axis and returns on the y-axis. The x-axis labels (months) are set with a 45-degree rotation to help with readability. The final visualization shows the distribution of AAPL's daily returns for each month in the data.

There's more...

Another popular chart type is the **`jointplot`** which is a combination of a scatter plot and histograms for each variable along the margins. Traders use joint plots to visually assess the correlation between assets, aiding in diversification, pairs trading, and risk management. The plots help in identifying linear relationships, distribution patterns, and potential outliers.

```
g = sns.jointplot(
    x="SPY",
    y="AAPL",
    data=returns,
    kind="reg",
    truncate=False,
)
```

The result is a joint plot.

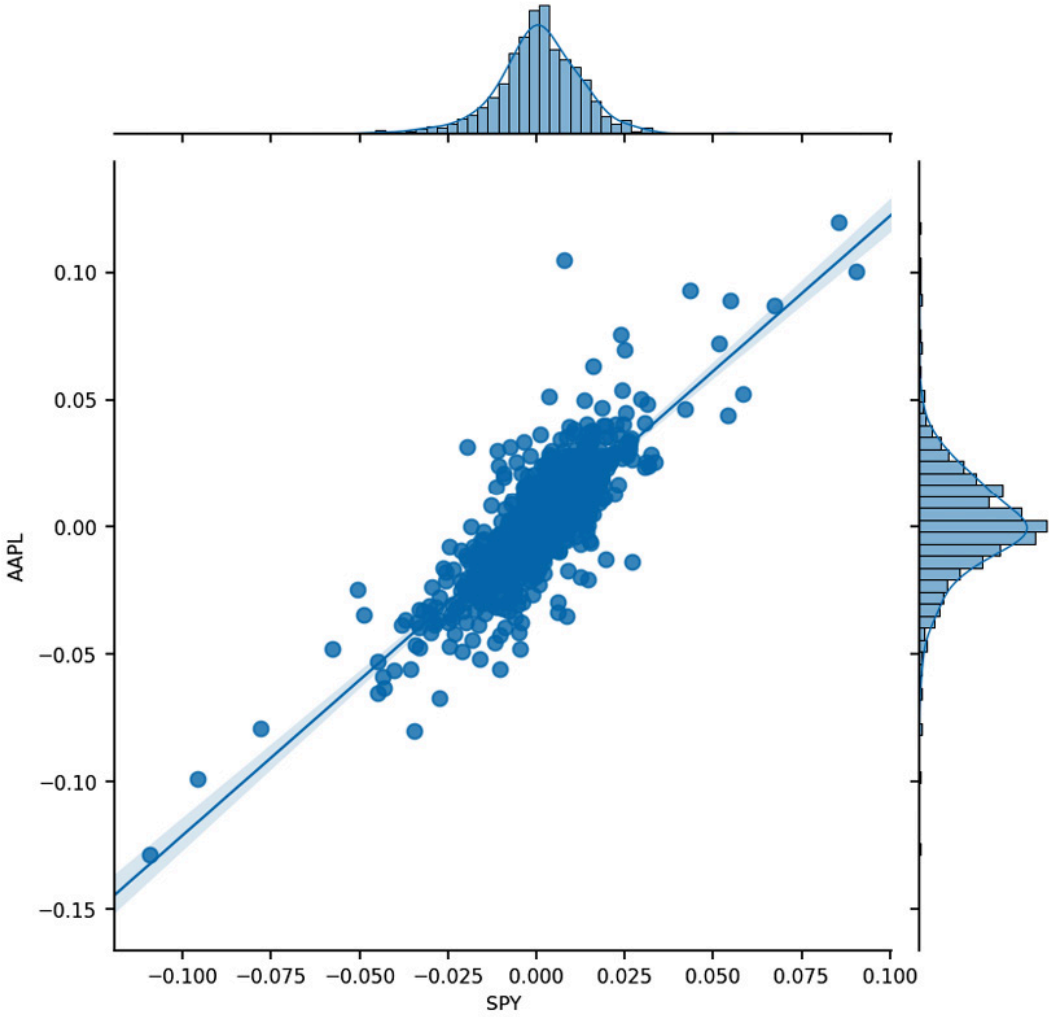

Figure 3.11: Joint plot demonstrating a positive linear relationship between AAPL and SPY returns.

> *TIP*
>
> *Note the* **`reg`** *argument passed to the* **`kind`** *parameter. This indicates the type of plot to draw. The value "reg" stands for regression, meaning Seaborn will not only plot the scatter points of SPY vs. AAPL returns but also compute and display a linear regression fit to the data.*

Traders use correlation matrices to understand the linear relationships between multiple assets simultaneously. Let's build a correlation matrix for the stocks in the Dow Jones Industrial Average.

1. The first step is to grab the list of companies and their ticker symbols from the DJIA Wikipedia page.

```
dji = pd.read_html(
    "https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average"
)[1]
```

The **`read_html`** method is a convenient function in pandas that extracts tables from a given webpage. We're interested in the second table (indexed as **`[1]`**) which has details about the companies in the DJIA.

2. Now that we have the ticker symbols, let's use them to get the historical stock price data and calculate the daily returns.

```
dji_data = (
    obb.equity.price.historical(
        dji.Symbol, start_date="2020-01-01", provider="yfinance"
    )
).pivot(columns="symbol", values="close")
dji_returns = dji_data.pct_change(fill_method=None).dropna()
```

3. To understand the relationship between different stocks, we compute the pairwise correlation between all of them using the pandas **`corr`** method.

```
corr = dji_returns.corr()
```

4. Before visualizing the correlation, we'll make some tweaks to make our heatmap more intuitive. First, we'll create a mask to hide the upper triangle of the correlation matrix, as it mirrors the lower triangle.

```
mask = np.triu(
    np.ones_like(corr, dtype=bool)
)
```

5. Next, we'll generate a color palette that will help visually distinguish positive from negative correlations in our heatmap.

```
cmap = sns.diverging_palette(230, 20, as_cmap=True)
```

6. Set the font size to 4 points to include more of the labels and plot the heatmap.

```
plt.rcParams["font.size"] = 4
sns.heatmap(
    corr,
    mask=mask,
    cmap=cmap,
    vmin=-1.0,
    vmax=1.0,
    center=0,
    square=True,
    linewidths=0.5,
)
```

The result is a correlation heatmap.

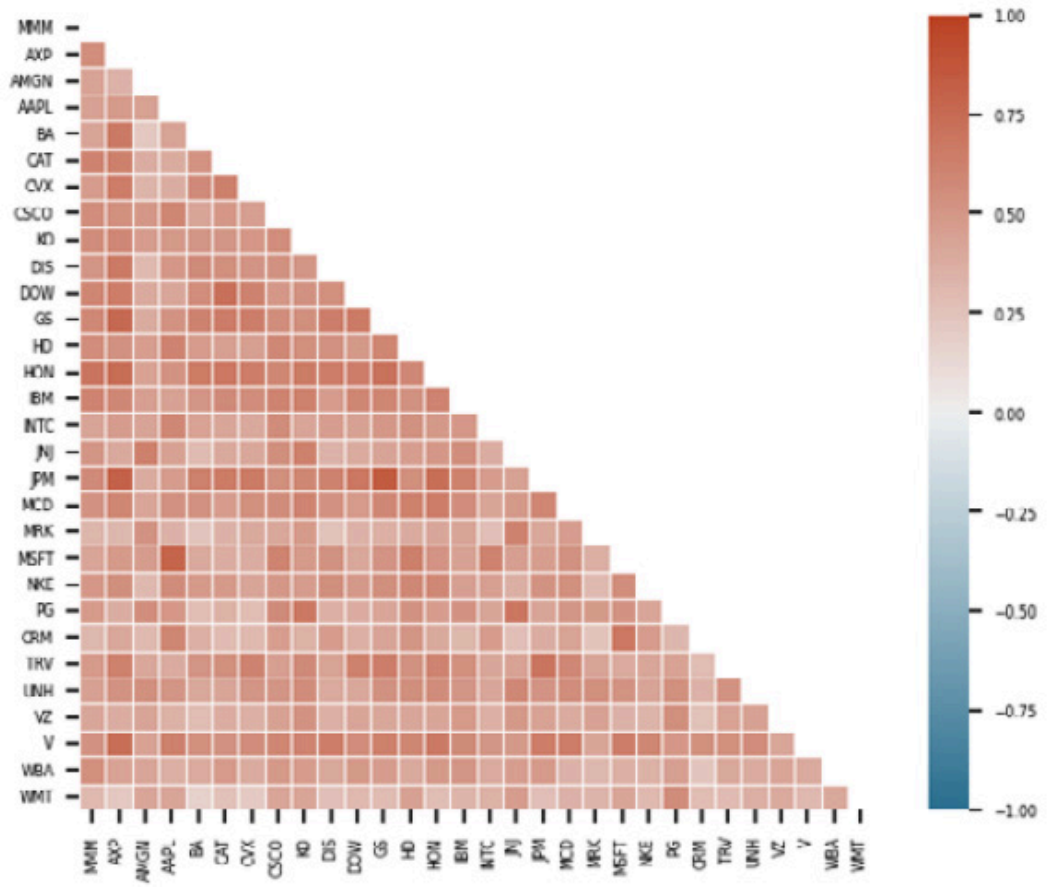

Figure 3.12: Heatmap shaded with the correlations between the constituents of the Dow Jones Industrial Average.

See also

Seaborn is a popular charting library with many unique visualization types. Read more here:

- https://seaborn.pydata.org/generated/seaborn.boxplot.html — Documentation for the Seaborn boxplot.
- https://seaborn.pydata.org/generated/seaborn.jointplot.html — documentation for the Seaborn joint plot.
- https://seaborn.pydata.org/generated/seaborn.heatmap.html — documentation for the Seaborn heatmap.

Creating an interactive PCA analytics dashboard with Plotly Dash

Principal component analysis is used widely in data science. It's a way to reduce the number of dimensions in a data set. In a stock portfolio, a dimension might be a column of returns for one of the stocks. In a portfolio of 100 stocks, there are 100 dimensions. PCA converts those 100 dimensions into the few that explain the most variance in the data.

PCA isolates the statistical return drivers of a portfolio. These drivers are called "alpha factors" (or just factors) because they create returns that are not explained by a benchmark. Quants use factors in trading strategies. First, they isolate the components. Then they buy the stocks with the largest exposure to a factor and sell the stocks with the smallest exposure to a factor. We'll look at PCA in a later recipe. For now, we'll use it to create a Plotly Dash app.

Instead of changing the dates, number of components, and ticker symbols in code, we can do it in an interactive web app. In this recipe, we'll create a Plotly Dash app that accepts a list of ticker symbols, iden-

tifies the principal components of their returns, and generates plots to visualize the top factors.

Getting ready…

So far, we've been writing code in Jupyter Notebook. For this recipe, you'll create a Python script called **`app.py`** and run it from the command line. Our aim is to highlight some of the intriguing features of Plotly Dash, even if they aren't essential for a basic app for conducting principal component analysis on stock returns.

You'll need to install a few new libraries. Make sure your virtual environment is active and run the following command:

```
pip install dash plotly dash-bootstrap-components scikit-learn
```

Plotly and **`scikit-learn`** may already be installed through the OpenBB Platform installation.

How to do it…

All the code below should be written in the **`app.py`** script file.

1. Begin by importing all necessary libraries to create the web application:

```
import datetime
import numpy as np
import pandas as pd
import dash
from dash import dcc, html
import dash_bootstrap_components as dbc
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import plotly.io as pio
from sklearn.decomposition import PCA
```

```
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Set the default styling, chart templates, and initialize the app:

```
pio.templates.default = "plotly"
app = dash.Dash(__name__, external_stylesheets=[
    dbc.themes.BOOTSTRAP])
```

3. Construct the components of the user interface starting with the text field to enter the list of ticker symbols, the dropdown to select the number of components, the date picker to select the range of data, and the submit button to run the app:

```
ticker_field = [
    html.Label("Enter Ticker Symbols:"),
    dcc.Input(
        id="ticker-input",
        type="text",
        placeholder="Enter Tickers separated by commas (
            e.g. AAPL,MSFT)",
        style={"width": "50%"}
    ),
]
components_field = [
    html.Label("Select Number of Components:"),
    dcc.Dropdown(
        id="component-dropdown",
        options=[{"label": i,"value": i} for i in range(1,6)],
        value=3,
        style={"width": "50%"}
    ),
]
date_picker_field = [
    html.Label("Select Date Range:"),  # Label for date picker
    dcc.DatePickerRange(
        id="date-picker",
        start_date=datetime.datetime.now() - datetime.timedelta(
            365 * 3),
        end_date=datetime.datetime.now(),
        # Default to today's date
        display_format="YYYY-MM-DD",
```

```
    ),
]
submit = [
    html.Button("Submit", id="submit-button"),
]
```

4. Combine the form elements and placeholders for visualizations to form the app layout:

```
app.layout = dbc.Container(
    [
        html.H1("PCA on Stock Returns"),
        dbc.Row([dbc.Col(ticker_field)]),
        dbc.Row([dbc.Col(components_field)]),
        dbc.Row([dbc.Col(date_picker_field)]),
        dbc.Row([dbc.Col(submit)]),
        dbc.Row(
            [
                dbc.Col([dcc.Graph(id="bar-chart")], width=4),
                dbc.Col([dcc.Graph(id="line-chart")],
                    width=4),
                dbc.Col([dcc.Graph(id="scatter-plot")],
                    width=4),
            ]
        ),
    ]
)
```

5. Implement the function that updates the charts upon user input:

> *IMPORTANT*
>
> *All the code between steps 5 and 13 belongs to the same function. Make sure you properly indent the code after the definition.*

```
@app.callback(
    [
        Output("bar-chart", "figure"),
        Output("line-chart", "figure"),
        Output("scatter-plot", "figure"),
```

```
    ],
    [Input("submit-button", "n_clicks")],
    [
        dash.dependencies.State("ticker-input", "value"),
        dash.dependencies.State("component-dropdown", "value"),
        dash.dependencies.State("date-picker", "start_date"),
        dash.dependencies.State("date-picker", "end_date"),
    ],
)
def update_graphs(n_clicks, tickers, n_components, start_date, end_date):
    if not tickers:
        return {}, {}, {}
```

6. Parse inputs from the user:

```
    tickers = tickers.split(",")
    start_date = datetime.datetime.strptime(start_date,
        "%Y-%m-%dT%H:%M:%S.%f").date()
    end_date = datetime.datetime.strptime(end_date,
        "%Y-%m-%dT%H:%M:%S.%f").date()
```

7. Download stock data:

```
    data = obb.equity.price.historical(
        tickers,
        start_date=start_date,
        end_date=end_date,
        provider="yfinance"
    ).pivot(columns="symbol", values="close")
    daily_returns = data.pct_change().dropna()
```

8. Fit the principal component model:

```
    pca = PCA(n_components=n_components)
    pca.fit(daily_returns)
    explained_var_ratio = pca.explained_variance_ratio_
```

9. Generate the bar chart for individual explained variance:

```
    bar_chart = go.Figure(
        data=[
            go.Bar(
                x=["PC" + str(i + 1) for i in range(
```

```
                    n_components)],
                y=explained_var_ratio,
            )
        ],
        layout=go.Layout(
            title="Explained Variance by Component",
            xaxis=dict(title="Principal Component"),
            yaxis=dict(title="Explained Variance"),
        ),
    )
```

10. Generate the line chart for cumulative explained variance:

```
    cumulative_var_ratio = np.cumsum(explained_var_ratio)
    line_chart = go.Figure(
        data=[
            go.Scatter(
                x=["PC" + str(i + 1) for i in range(
                    n_components)],
                y=cumulative_var_ratio,
                mode="lines+markers",
            )
        ],
        layout=go.Layout(
            title="Cumulative Explained Variance",
            xaxis=dict(title="Principal Component"),
            yaxis=dict(title="Cumulative Explained Variance"),
        ),
    )
```

11. Compute factor exposures:

```
    X = np.asarray(daily_returns)
    factor_returns = pd.DataFrame(
        columns=["f" + str(i + 1) for i in range(
            n_components)],
        index=daily_returns.index,
        data=X.dot(pca.components_.T),
    )
    factor_exposures = pd.DataFrame(
        index=["f" + str(i + 1) for i in range(n_components)],
        columns=daily_returns.columns,
```

```
            data=pca.components_,
        ).T
        labels = factor_exposures.index
        data = factor_exposures.values
```

12. Generate the chart for factor exposures:

```
        scatter_plot = go.Figure(
            data=[
                go.Scatter(
                    x=factor_exposures["f1"],
                    y=factor_exposures["f2"],
                    mode="markers+text",
                    text=labels,
                    textposition="top center",
                )
            ],
            layout=go.Layout(
                title="Scatter Plot of First Two Factors",
                xaxis=dict(title="Factor 1"),
                yaxis=dict(title="Factor 2"),
            ),
        )
```

13. Return the charts to the app:

```
        return bar_chart, line_chart, scatter_plot
```

To run the app, open the terminal, navigate to the directory where the **`app.py`** script is located, and run the following command:

```
python app.py
```

Running the code starts the app and prints the URL where it is running. Navigate to the URL, enter a list of tickers and press submit. The result will look something like this:

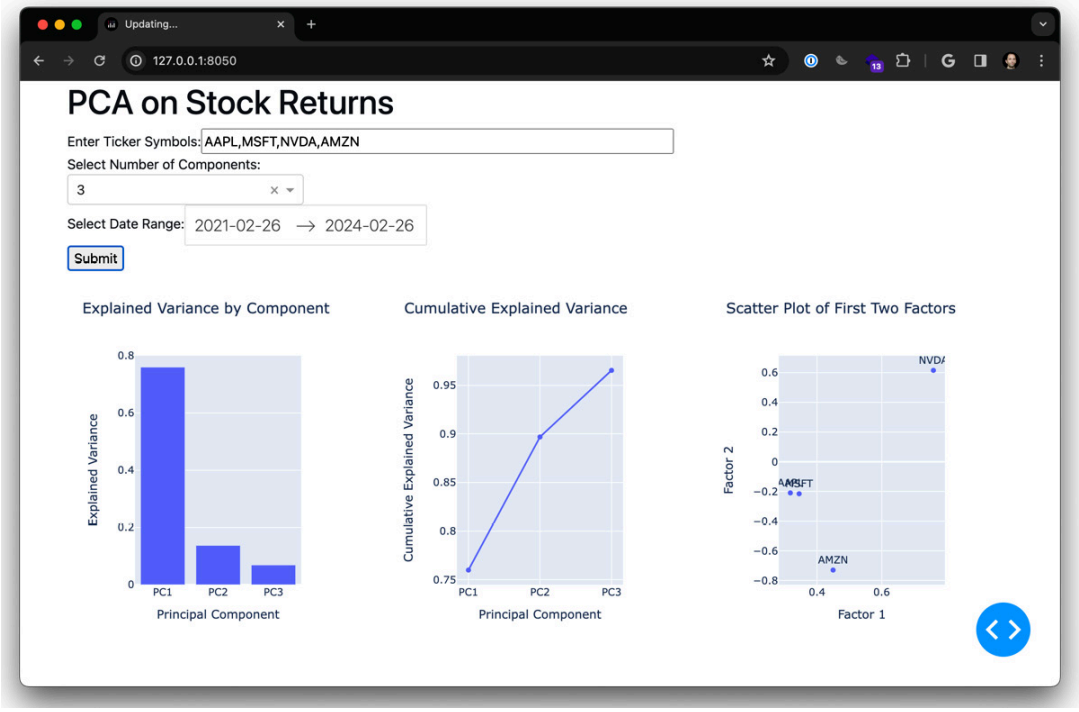

Figure 3.13: Screenshot of our new PCA analytics app

How it works...

Callbacks in Plotly Dash define the logic that connects the components in the application. These callbacks are what enable user interactions. A callback refers to a Python function that gets automatically executed by Dash whenever an input component's property changes. In our app, the callback is defined as **`update_graphs`**.

First we import the necessary libraries we need for the analysis and web app. We import **`datetime`** for handling dates, NumPy for numerical operations, and pandas for data manipulation. Dash, its extensions, and other related packages provide the backbone for our web application. Specifically, **`dash_bootstrap_components`** assists in styling, and OpenBB fetches the stock data. Lastly, PCA from sklearn will help us perform the principal component analysis.

We then choose a default template for styling the charts and initialize our Dash app with a Bootstrap theme. Next we construct the components the user will interact with like a text field for entering ticker symbols, a dropdown to choose the number of principal components,

a date picker for specifying the data range, and a submit button to start the analysis. Next we organize our UI components into a layout.

Next, we implement the callback function, **`update_graphs`**, which ties our user inputs to the visualization outputs. Every time the user presses the submit button, this function fetches the data, performs the PCA, and updates the visualizations. We define a callback by using the **`callback`** decorator provided by Dash. This decorator specifies which component properties to watch (**`Input`**) and which to update (**`Output`**). In our app, clicking the submit button triggers the callback function. The decorator takes three inputs:

- **`Output`**: Specifies which components will be updated once the callback function is executed.
- **`Input`**: Specifies which components the callback should listen to.
- **`State`**: Specifies which components the callback should read the current value from without triggering the function.

When the submit button is clicked, **`update_graphs`** gets called, the current values of the specified components are passed as arguments to the function, and the logic inside **`update_graphs`** executes. The function returns the visualizations, and these are rendered in the specified **`Output`** components in the app.

There's more...

Plotting with different Python libraries has distinct advantages depending on the context. pandas, while inherently great for data manipulation, offer basic plotting capabilities that are mainly suited for quick and simple visualizations directly from DataFrames. Matplotlib, one of the most widely used plotting libraries for Python, provides a greater degree of flexibility and customization but generally lacks interactive features. Seaborn, which is built on top of Matplotlib, enhances visual aesthetics and has functions tailored for statistical visualizations, making it more intuitive for certain analyses. Plotly

Express, on the other hand, is a more modern library designed for interactivity from the ground up, making it well-suited for dynamic environments like Plotly Dash apps.

See also

PCA is common in all data sciences and especially useful in algorithmic trading. Read more about the technical details of PCA at Wikipedia.

- Description of PCA: **https://en.wikipedia.org/wiki/Principal_component_analysis** .
- Plotly Dash user guide: **https://dash.plotly.com**.