Chapter 8

Bonus Recipes - 8-1 and 8-2

8-1: Plotting time series data using pandas

Visualization is a crucial component of data analysis, especially when working with time series data. In previous chapters and recipes, you have encountered numerous instances where plotting data was essential for highlighting specific points or drawing conclusions about the time series. Visualizing time series data allows us to quickly identify patterns, trends, outliers, and other critical insights at a glance. Effective visualization not only aids in understanding the data but also facilitates communication across different groups, bridging the gap between business professionals and data scientists by providing a common platform for discussion and fostering constructive dialogue.

In time series analysis—data science in general—we rely heavily on visualizations during **exploratory data analysis** (**EDA**) to gain a comprehensive understanding of the data we're working with. Visualization is also integral when evaluating models to compare their performance or identify areas for improvement. It also plays a key role in model explainability, helping stakeholders to understand how models make predictions. Furthermore, after deploying our models, we utilize visualizations for monitoring model performance over time to detect issues such as model drift or performance degradation.

The `pandas` library offers built-in plotting capabilities for visualizing data stored in a DataFrame or Series objects. These visualizations are powered by **Matplotlib**, which serves as pandas' default plotting library. By simply calling `DataFrame.plot()` or `Series.plot()`, you can generate a line plot by default.

You can change the type of the plot in two ways:

- **Using the `kind` argument**: Specify the plot type with the `.plot()` method by setting the `kind` parameter. For example, `.plot(kind="hist")` creates a histogram, while `.plot(kind="bar")` generates a bar plot.
- **Using method chaining**: Call specific plot functions by chaining methods. For example, `.plot.hist()` creates a histogram, and or `.plot.line()` generates a line plot.

This recipe will demonstrate how to use the standard pandas `.plot()` method with a Matplotlib backend.

Getting ready

This recipe extensively uses pandas 2.3.3 which is the latest version as of this writing. You can use a more recent version if available.

There are three additional libraries that we will be using:

- `hvplot` (part of `PyViz`)
- `seaborn`
- `matplotlib`

If you are using pip, then you can install these packages from your terminal with the following:

```
pip install hvplot seaborn matplotlib jupyterlab
```

If you are using conda, then you can install these packages with the following:

```
conda install jupyterlab matplotlib seaborn hvplot
```

The hvPlot library will be used to build interactive visualizations in JupyterLab. If you are on the latest version of JupyterLab (jupyterlab >= 3.0), then all the required extensions are automatically installed as they are bundled via the pyviz_comms package. If you are on an older version of JupyterLab (jupyterlab < 3.0), then you will need to install the `jupyterlab_pyviz` extension manually, as shown:

```
jupyter labextension install @pyviz/jupyterlab_pyviz
```

```
import pandas as pd
import matplotlib.pyplot as plt
from utils import load_ch8_datasets, plot_comparison
airp_df, closing_price, co2_df = load_ch8_datasets()
```

You will be using the Closing Price Stock dataset for Microsoft, Apple, and IBM (`closing_price`)

How to do it...

In this recipe, you will learn how to plot time series data, apply themes, create subplots, normalize the data for better comparison, and save the visualizations:

1. Plotting in pandas can be done by simply appending `.plot()` to a DataFrame or Series object:

   ```
   closing_price.plot();
   ```

   This will produce a line plot, which is the default option for the `kind` parameter, equivalent to specifying `.plot(kind="line")`. The output will look like this:

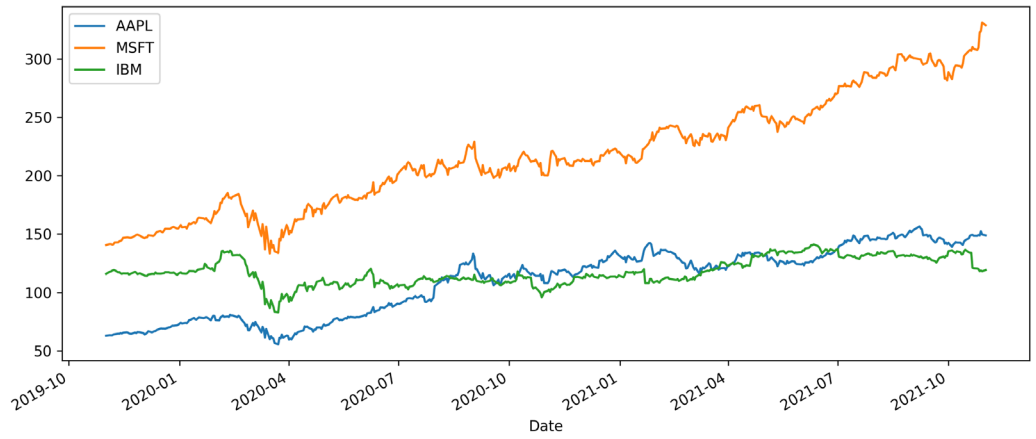

Figure 8.1: Multiline time series plot using pandas

You can further customize the plot by adding a title, updating the axis labels, and adjusting the ticks (`xticks` and `yticks`), to name a few ways. For example, to add a title and update the y-axis label, use the `title` and `ylabel` parameters as shown:

```
start_date = '2019'
end_date = '2021'
closing_price.plot(
    title=f'Closing Prices from {start_date} - {end_date}',
    ylabel= 'Closing Price'
);
```

2. To compare fluctuations in prices across multiple stocks, **normalize** the data so all series start at the same baseline. To accomplish this, just divide each stock's prices by the first-day price (its initial value). This will make all the stocks have the same starting point:

```
closing_price_n = closing_price.div(closing_price.iloc[0])
closing_price_n.plot(
    title=f'Closing Prices from {start_date} - {end_date}',
    ylabel= 'Normalized Closing Price'
);
```

This would produce the following plot:

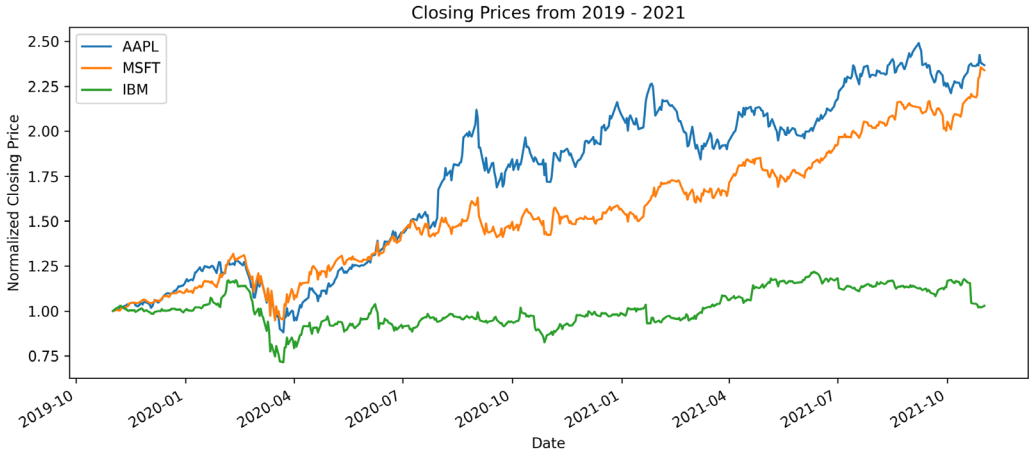

Figure 8.2: Using a simple normalizing technique to make it visually easier to compare price fluctuations

From the output, you can observe that all lines now start at the same origin (`1.0`). This makes it easier to compare how prices deviate from each other over time. Let's examine the actual normalized values to see how our transformation is working. Let's look at the first few rows of our normalized data:

```
closing_price_n.head()
```

Notice that the first row from the output table is set to 1.0 for every stock (displayed in the following figure with higher precision). Each stock's first-day price now serves as the baseline (1.0), and subsequent values represent the proportional change from that starting point:

Date	AAPL	MSFT	IBM
2019-11-01	1.000000	1.000000	1.000000
2019-11-04	1.006567	1.005775	1.015790
2019-11-05	1.005121	1.005149	1.017413
2019-11-06	1.005551	1.002366	1.023980
2019-11-07	1.017156	1.003757	1.027937

Figure 8.3: Output of normalized time series with a common starting point at 1

3. Matplotlib provides several style templates to change the appearance of your plots. Use `plt.style.use()` to specify a style name from an existing template or use a custom style. For example, the following code shows how you can change from the `default` style to the `ggplot` style:

```
plt.style.use('ggplot')
closing_price_n.plot(
    title=f'Closing Prices from {start_date} - {end_date}',
    ylabel= 'Normalized Closing Price'
);
```

The preceding code should produce the same plot in terms of data content but in a different style.

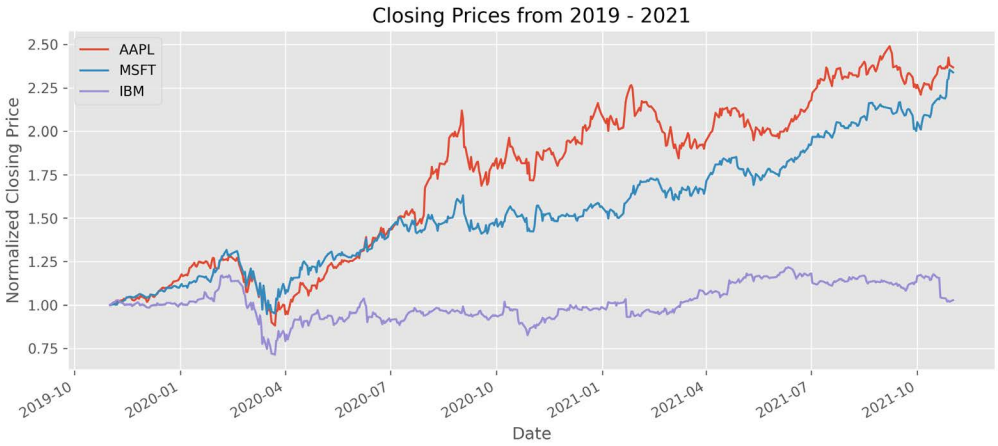

Figure 8.4: Using the ggplot style from Matplotlib

The `ggplot` style was inspired by the `ggplot2` package in **R**.

You can explore other attractive styles such as: `fivethirtyeight` (inspired by **fivethirtyeight.com**), `dark_background`, and `tableau-colorblind10`.

For a list of available style templates in Matplotlib, you can visit their style sheets reference page: `https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html`

To revert to the original (default) style, use this:

```
plt.style.use("default")
```

4. To save your plot as an image file, use the `.savefig()` method, which supports various formats (e.g., JPEG, PNG, or SVG). For example, to save the plot as a high-resolution JPEG file, use the following code:

```
plot = closing_price_n.plot(
    title=f'Closing Prices from {start_date} - {end_date}',
    ylabel= 'Norm. Price'
)
plot.get_figure().savefig('plot_1.jpg', dpi=300)
```

This saves the plot as a high-resolution (300 dpi) image named `plot_1.jpg` in your local directory. Using a higher dpi value, such as 300, ensures the plot is suitable for printing or publication-quality output.

How it works...

The pandas and Matplotlib libraries work closely together, with ongoing efforts to integrate and expand plotting capabilities within pandas. This collaboration allows you to create visualizations directly from pandas data structures, such as DataFrames and Series, while leveraging Matplotlib's extensive customization options.

There are many plotting styles available in pandas, which can be specified using the `kind` argument in the `.plot()` method. Here are common plot types and their use case:

- `Line`: Line charts, ideal for visualizing time series data (default option)
- `bar` or `barh`: Vertical or horizontal bar plots to compare quantities or categories
- `hist`: Histogram plots for visualizing frequency distributions
- `box`: Boxplots, useful for summarizing data distribution and identifying outliers
- `kde` or `density`: Kernel density estimation plots for visualizing probability density
- `area`: Area plots, often used for cumulative data visualization (variations of line plots with filled areas beneath the curves)
- `pie`: Pie charts for categorical data proportions
- `scatter`: Scatter plots to display relationships between two variables
- `hexbin`: Hexagonal bin plots for visualizing density in large datasets

These built-in plot types allow for quick and efficient visualization of time series and other data. Each of these styles provides unique insights and is suited to different types of data or analysis.

There's more...

As observed in the previous section, we plotted all three columns in the time series in one plot, with three line charts displayed on the same axes. However, in some cases, you may want to visualize each column separately to better analyze individual trends or patterns without the distraction of overlapping lines.

To create separate subplots, set the `subplots` parameter to `True` in the `.plot()` method:

```
closing_price.plot(
    subplots=True,
    title=f'Closing Prices from {start_date} - {end_date}'
);
```

This code will generate three separate subplots, one for each column (stock symbol) in the `closing_price` DataFrame, with each subplot displaying its respective time series as a line plot:

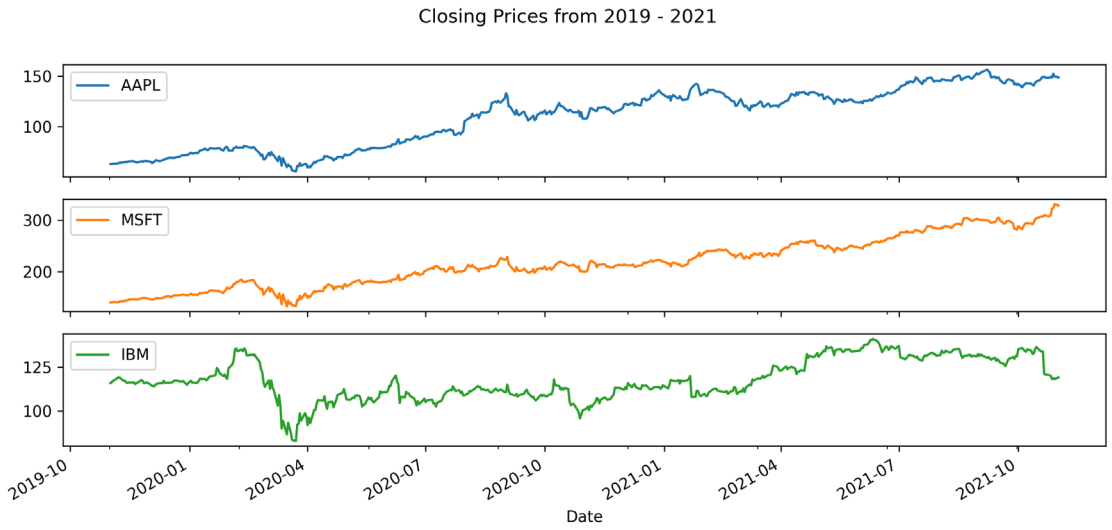

Figure 8.5: Using the pandas subplot feature

This is particularly useful in the following situations:

- **Columns have different scales:** Separate plots prevent overlapping or distorted trends caused by significant differences in value ranges
- **Detailed analysis:** Viewing each time series individually allows you to focus on specific characteristics or patterns within a single variable
- **Presentation clarity:** Subplots can reduce visual complexity, making it easier for stakeholders to interpret the results

See also

To learn more about pandas charting and plotting capabilities, visit the official documentation here:

`https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html`

8.2: Plotting time series data with interactive visualizations using hvPlot

Interactive visualizations allow us to analyze data more efficiently compared to static visuals. Interactions such as zooming, panning, and slicing through visualizations can reveal hidden patterns or anomalies, offering deeper insights for further investigation.

In this recipe, we will explore the **hvPlot** library to create interactive visualizations. Built on top of HoloViews, hvPlot provides a high-level API for data visualization and integrates seamlessly with various data sources, including **pandas**, **Xarray**, **Dask**, **Polars**, **NetworkX**, **Streamz**, **DuckDB**, and **GeoPandas**. For rendering plots, hvPlot supports multiple backends, including **Matplotlib**, **Bokeh** (default), and **Plotly**, giving you flexibility in choosing the output style.

The library allows you to generate dynamic, interactive plots with minimal modifications to your existing pandas plotting code. In this recipe, we will use the `closing_price.csv` dataset to explore the capabilities of hvPlot.

Getting ready

You can download the Jupyter notebooks and datasets needed from the GitHub repository. Please refer to the *Technical requirements* section of this chapter.

How to do it...

1. Start by importing the libraries needed. hvPlot provides a pandas extension, making it very convenient to use with familiar syntax. This will allow you to use the same syntax as in the previous recipe, enabling an easy transition to interactive visualizations:

```
import pandas as pd
import hvplot.pandas

closing_price_n = closing_price.div(closing_price.iloc[0])
```

2. When plotting using pandas, you would typically use the `.plot()` method – for example, `closing_price_n.plot()`. Similarly, hvPlot allows you to render an interactive plot simply by substituting `.plot()` with `.hvplot()`. This is especially useful if you have a dense chart in terms of content. Interactive plots allow you to zoom into a specific portion of the chart and use the panning feature to move to different portions of the chart:

```
start_date = '2019'
end_date = '2021'

closing_price_n.hvplot(
    title=f'Closing Prices from {start_date} - {end_date}')
```

By replacing `.plot` with `.hvplot`, you get an interactive visualization with features such as the hover effect and zooming:

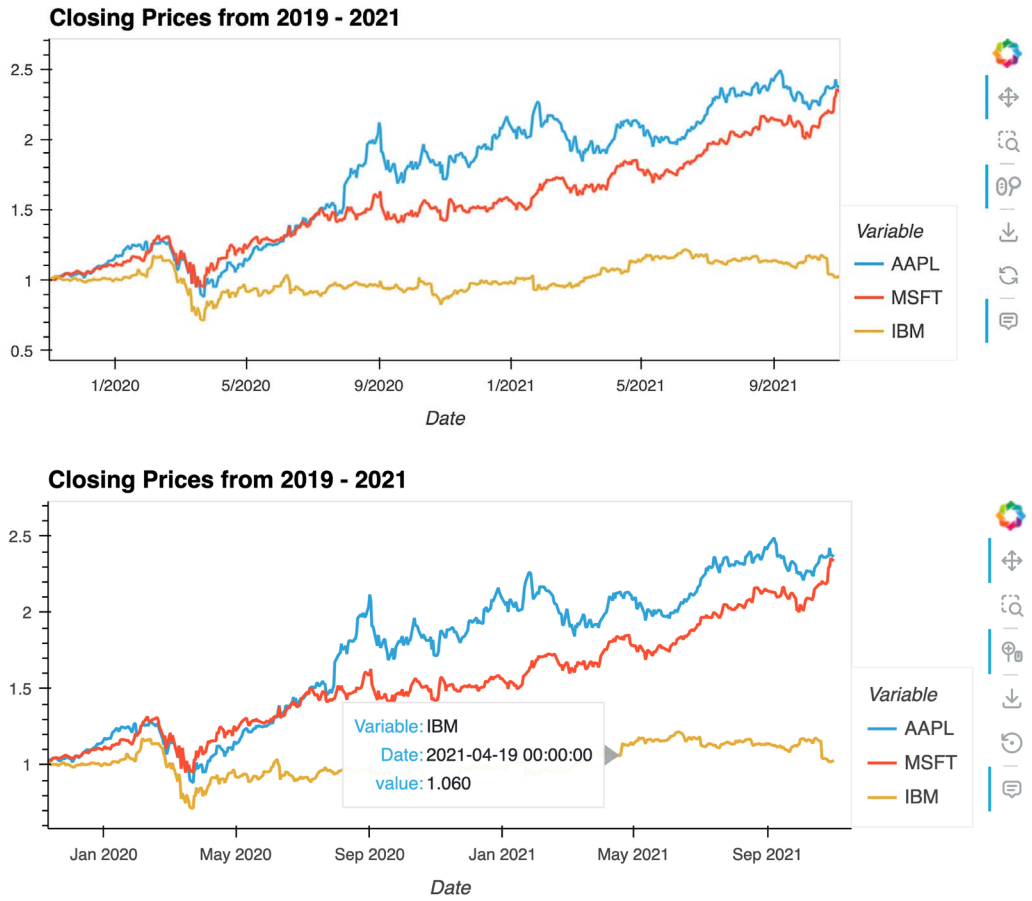

Figure 8.6: hvPlot interactive visualization

You can achieve the same interactive result by switching the pandas' plotting backend from the default `matplotlib` to `hvlpot`. Update the `backend` parameter to `'hvplot'` as shown:

```
closing_price_n.plot(
    backend='hvplot',
    title=f'Closing Prices from {start_date} - {end_date}'
)
```

This should produce the same plot as in *Figure 8.6*. Notice the **widget bar** to the right, which includes six interaction modes: **Pan, Box Zoom, Wheel Zoom, Save, Reset, and Hover:**

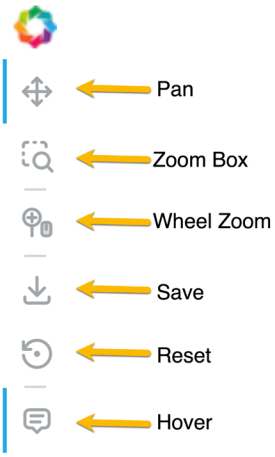

Figure 8.7: Widget bar with six modes of interaction

3. You can split each time series into separate plots per symbol (column). For example, you can split the time series into three separate plots (subplots), one for each symbol: MSFT, AAPL, and IBM. **Subplots** can be created by setting the `subplots=True` parameter:

```
closing_price.hvplot(width=300, subplots=True)
```

This should generate a subplot for each column:

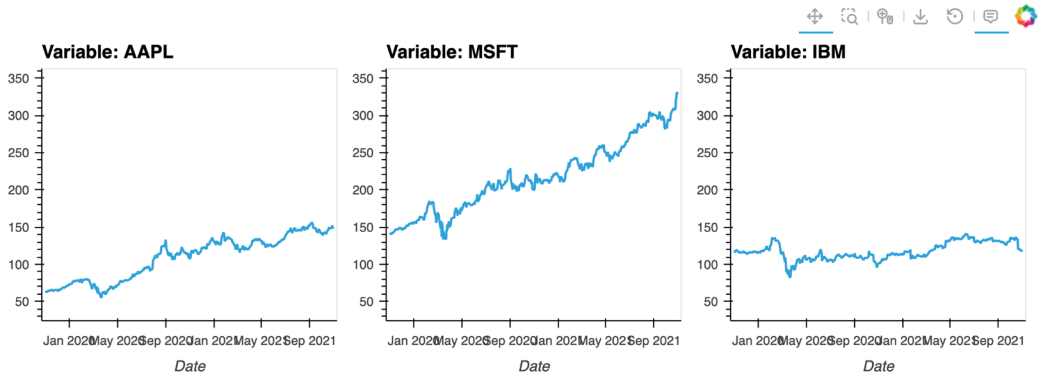

Figure 8.8: hvPlot subplot example

Subplotting is helpful when you want to analyze each time series separately without overlapping data from other columns.

4. For greater control over the subplot layout, use the `.cols()` method. This method allows you to specify the number of plots per row. Here's an example:

   - `cols(1)` creates **one** plot per row
   - `cols(2)` creates **two** plots per row

```
closing_price.hvplot(width=300, subplots=True).cols(2)
```

This should produce a figure with two subplots in the first row and the third subplot on the second row:

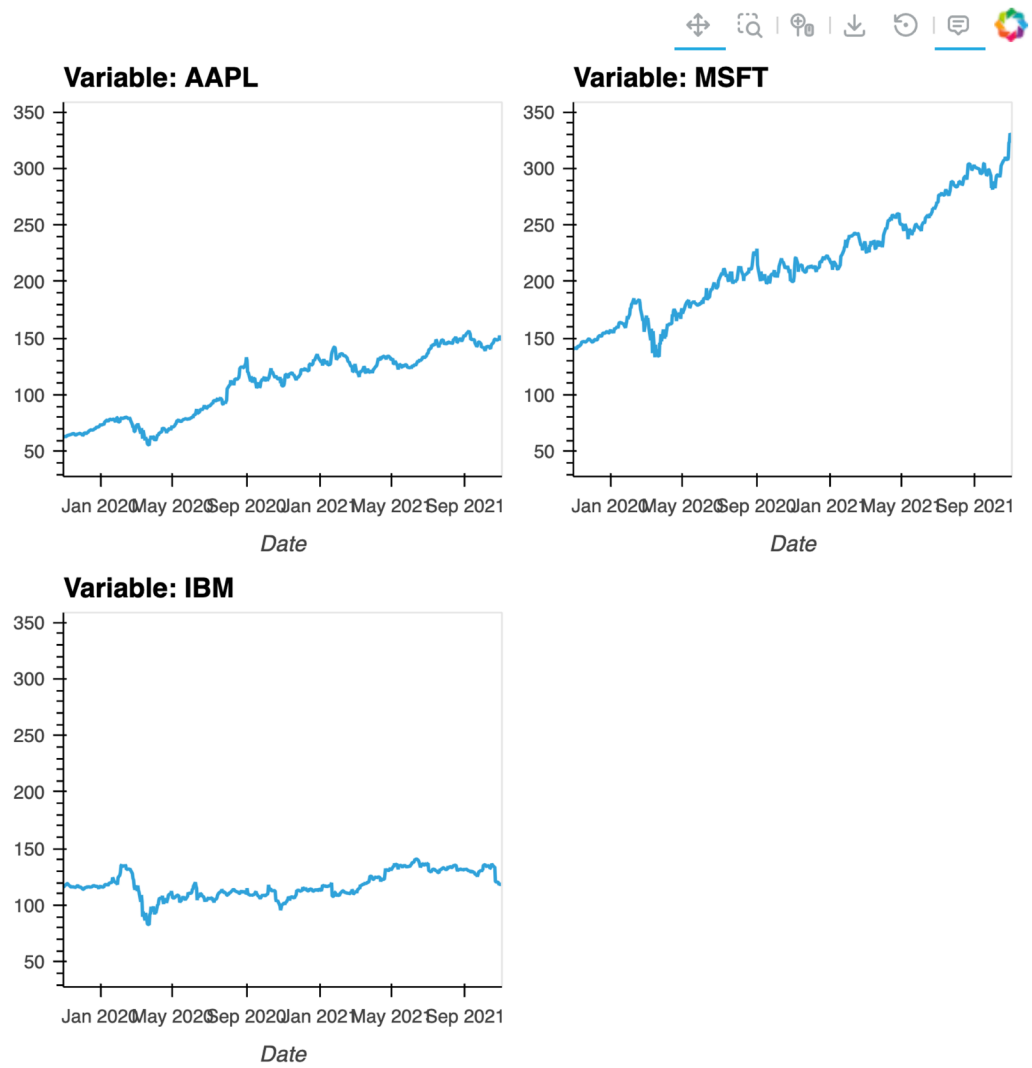

Figure 8.9: Example hvPlot with two columns per row using .col(2)

The `.cols()` method only works if the `subplots` parameter is set to `True`. Otherwise, you will get an error.

How it works...

Given the widespread use of pandas, many libraries now seamlessly integrate with pandas DataFrames and Series for input data. hvPlot builds on this integration by extending pandas functionality with its own `.hvplot()` method, simplifying the process of creating interactive visualizations. Additionally, hvPlot integrates seamlessly with Matplotlib and other backends, making it easy to change the plotting engine used with pandas.

hvPlot provides several convenient options for plotting your DataFrame. You can do the following:

- Switch the backend when using `DataFrame.plot()`
- Use `DataFrame.hvplot()`
- Leverage hvPlot's native API for more advanced visualizations.

Switching backends

By default, hvPlot uses **Bokeh** as its backend, but you can easily switch to other backends, such as **Plotly**. Before switching, ensure the necessary library is installed:

Here's how to install using **conda**:

```
conda install plotly
```

Here's how to install using **pip**:

```
pip install plotly
```

Switching the global backend for pandas

To change the global plotting backend for all pandas plots, set the `pd.options.plotting.backend` option. For example, here's how to switch to **Plotly**:

```
pd.options.plotting.backend = "plotly"
```

To restore the default, **Matplotlib**, just use the following:

```
pd.options.plotting.backend = "matplotlib"
```

Switching the hvPlot backend

If you only want to change the default backend for hvPlot visualizations, use the `hvplot.extension()` method:

```
hvplot.extension('plotly')
```

Here's how to restore the hvPlot default backend (Bokeh):

```
hvplot.extension('bokeh')
```

Changing the backend for a specific visualization

To use a specific backend for an individual visualization, pass the `backend` parameter to the `.plot()` method:

```
closing_price_n.plot(
    backend='plotly',
    title=f'Closing Prices from {start_date} - {end_date}'
)
```

This will generate an interactive plot using **Plotly** as the backend:

Figure 8.10: Example using plotly as the backend

Let's say you get the following error:

```
ValueError: Mime type rendering requires nbformat>=4.2.0 but it
is not installed
```

You will need to install **nbformat**:

To install with **pip**, use the following:

```
pip install nbformat
```

To install with **conda**, use the following:

```
conda install nbformat
```

Once installed, you may need to restart your JupyterLab (or Notebook) kernel.

Switching backends allows you to tailor the visualization experience to your needs.

There's more...

hvPlot offers powerful layout and grouping capabilities using two arithmetic operators, + and *, as well as the by parameter.

The plus operator (+) allows you to align two or more charts side by side on the same row, while the multiplication operator (*) combines two or more charts into a single plot (merge) by overlaying one on top of the other. In the following example, we will add two plots, so they are aligned side by side on the same row:

```
(closing_price_n['AAPL'].hvplot(width=400) +
 closing_price_n['MSFT'].hvplot(width=400))
```

This should produce what is shown in the following figure:

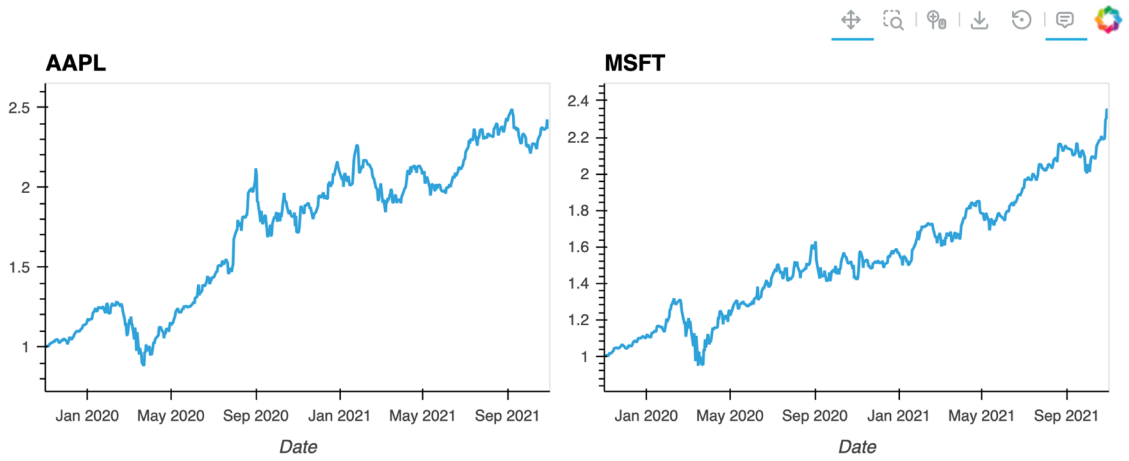

Figure 8.11: Two plots side by side using the addition operator

The two plots will share the same widget bar. For example, if you filter or zoom into one of the charts, the other chart will have the same action applied.

Now, let's see how the multiplication operator combines the two plots into a single visualization:

```
(closing_price_n['AAPL'].hvplot(width=500, height=300) *
 closing_price_n['MSFT'].hvplot()).opts(legend_position='top_left')
```

The code should produce a single plot that combines both AAPL and MSFT:

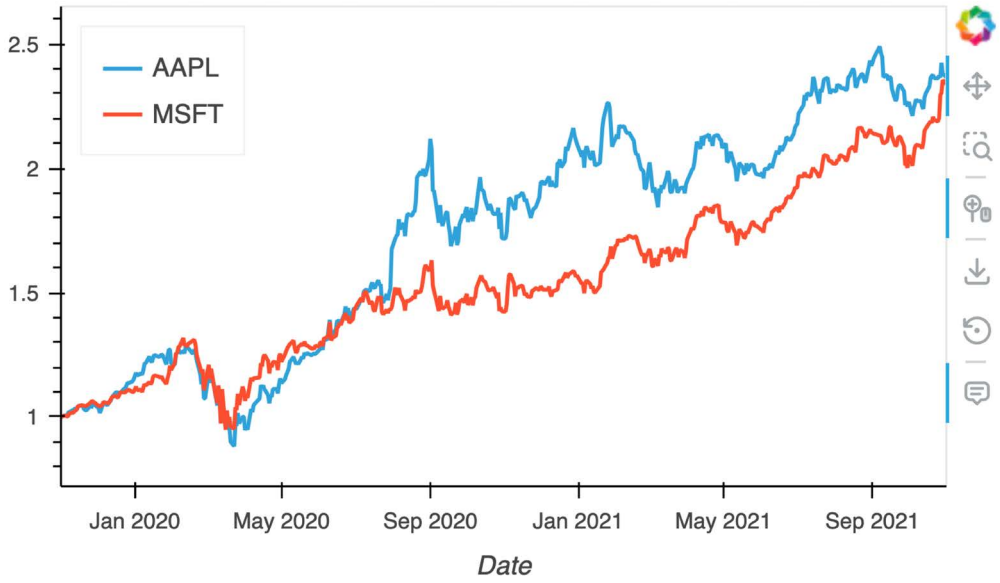

Figure 8.12: Two plots combined into one using the multiplication operator

To create **subgroups** (akin to a `group by` operation) where each group is represented by a different color, you can use the `by` parameter. For example, you can group time series data by year as follows:

```
closing_price['AAPL'].hvplot.line(by=['index.year'])
```

This generates a line chart segmented by year (grouped by), as shown in *Figure 8.13*:

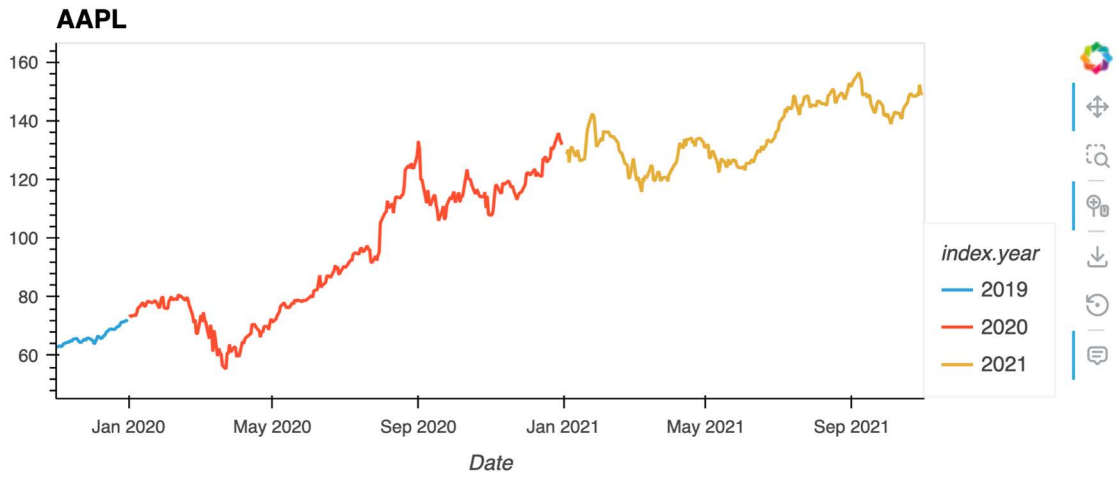

Figure 8.13: Line chart with subgroups (by year)

Given that we have data spanning three years, you will observe three distinct colors on the chart, each corresponding to a different year, as indicated by the legend.

See also

For more information on hvPlot, please visit their official page here: `https://hvplot.holoviz.org/`