Applied Causal Inference with Examples from Electricity Markets

Davide Cacciarelli

Jul 07, 2024

CONTENTS

This work-in-progress book is designed to provide comprehensive tutorials on causal inference methodologies. This book explains key concepts in this rapidly growing research area and showcases potential applications within electricity markets.

Prerequisites

To get the most out of this book, it is recommended that readers have a basic understanding of:

- **Statistics and probability**: familiarity with basic statistical concepts and probability theory.
- **Python programming**: basic knowledge of Python and possibly experience with libraries such as NumPy, pandas, statsmodels, and scikit-learn.
- **Machine learning**: an understanding of linear regression models, and an overview of machine learning concepts.

If you feel like you need a refresher, at the beginning of this book you can find a **crash course** to review these essential concepts. These chapters are designed to help you brush up on the foundational knowledge required to fully engage with the material in this book. However, if you are already familiar with these concepts or prefer to go directly to the applied causal inference part, feel free to skip those chapters.

Learning objectives

By the end of this book, readers will be able to:

- Understand and apply causal inference techniques to analyse relationships between variables.
- Use directed acyclic graphs (DAGs) to represent and reason about causal structures.
- Implement and interpret various causal discovery algorithms.
- Estimate causal effects using methods like instrumental variables, double machine learning, and difference-in-differences.
- Interpret results from complex machine learning models.
- Design experiments to gather data that supports causal analysis.

Note

The examples are provided in **Python**:

- Each chapter is designed to be as self-contained as possible, enabling you to focus on specific topics without needing prior chapters.
- Chapters can be run independently, allowing you to generate data and apply statistical methods on their own.
- Each chapter is available for download as a **Jupyter Notebook**, facilitating hands-on learning and enabling you to reproduce the results effortlessly.

References

This book is highly applied, aimed at providing essential intuitions and practical examples of how to apply causal inference methods to real-world problems. For additional theoretical and technical explanations, we suggest:

- Causality: Models, Reasoning and Inference
- Elements of Causal Inference
- Statistical Causal Discovery: LiNGAM Approach
- Introduction to Causal Inference from a Machine Learning Perspective

---

Built with Jupyter Book 2.0 tool set, as part of the ExecutableBookProject.

If you have any inquiries, please contact *d.cacciarelli@imperial.ac.uk*.

Part I

Crash course on Stats and ML

CHAPTER

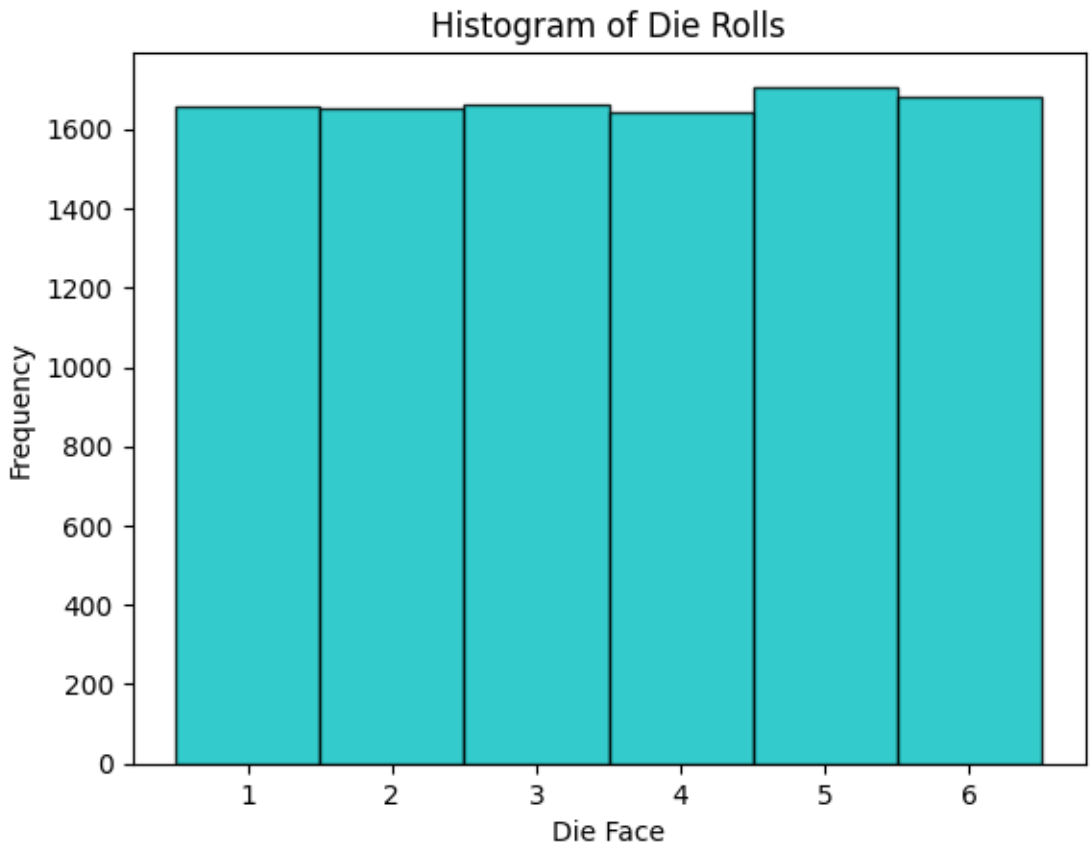
ONE

PROBABILITY THEORY AND STATISTICS

1.1 Basic concepts

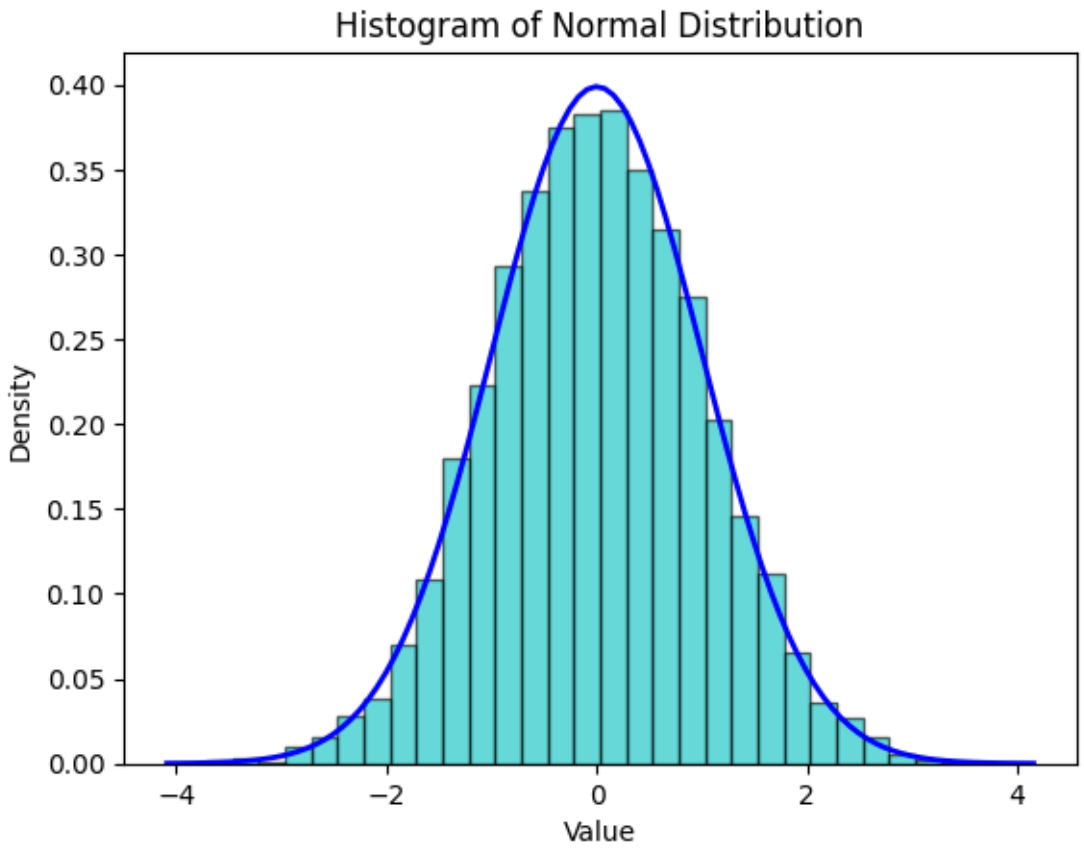
A **random variable** is a fundamental concept in probability and statistics. It represents a variable whose values are determined by the outcomes of a random phenomenon. A **discrete random variable** can take on a finite or countable number of distinct values. For example, the roll of a fair six-sided die is a discrete random variable, as it can result in one of six possible outcomes (1 through 6). The probability distribution of this random variable is uniform, meaning each outcome has an equal probability of occurring.

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

# Simulate rolling a die 10000 times
rolls = np.random.randint(1, 7, size=10000)

# Plot the results
plt.hist(rolls, bins=np.arange(1, 8) - 0.5, edgecolor='k', color='c', alpha=.8)
plt.xticks(range(1, 7))
plt.xlabel('Die Face')
plt.ylabel('Frequency')
plt.title('Histogram of Die Rolls')
plt.show()
```



A **continuous random variable** can take any value within a given range. The normal distribution (or Gaussian distribution) is a common continuous distribution characterized by its bell-shaped curve. It is defined by its mean (expected value) and standard deviation (a measure of variability).

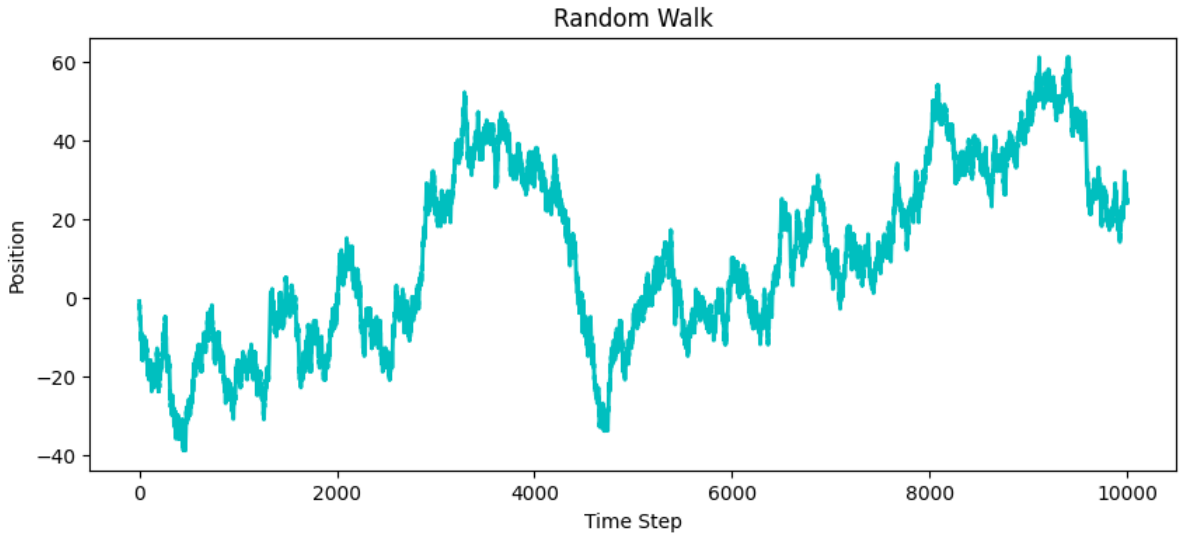
```
import scipy.stats as stats

# Generate data from a normal distribution
mean = 0
std_dev = 1
data = np.random.normal(mean, std_dev, 10000)

# Plot the results
plt.hist(data, bins=30, density=True, alpha=0.6, color='c', edgecolor='k')

# Plot the normal distribution
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
p = stats.norm.pdf(x, mean, std_dev)
plt.plot(x, p, 'b', linewidth=2)
plt.xlabel('Value')
plt.ylabel('Density')
plt.title('Histogram of Normal Distribution')
plt.show()
```



A **random process** (or stochastic process) is a collection of random variables indexed by time or another variable, used to model systems that evolve randomly over time or space. A random process is a function that assigns a random variable to each point in a time or space domain (examples include stock market prices, weather patterns, and noise signals in electrical engineering). We can distinguish between discrete-time processes (whose indices are countable), and continuous-time processes (whose indices are an interval). We can also distinguish between stationary processes (whose statistical properties are constant over time), and nonstationary processes.

A random walk is a simple example of a discrete-time random process, where each step is determined randomly, leading to a path that evolves over time.

```
# Generate a random walk
n_steps = 10000
steps = np.random.choice([-1, 1], size=n_steps)
random_walk = np.cumsum(steps)

# Plot the random walk
plt.figure(figsize=(10, 4))
plt.plot(random_walk, c='c', lw=2)
plt.xlabel('Time Step')
plt.ylabel('Position')
plt.title('Random Walk')
plt.show()
```



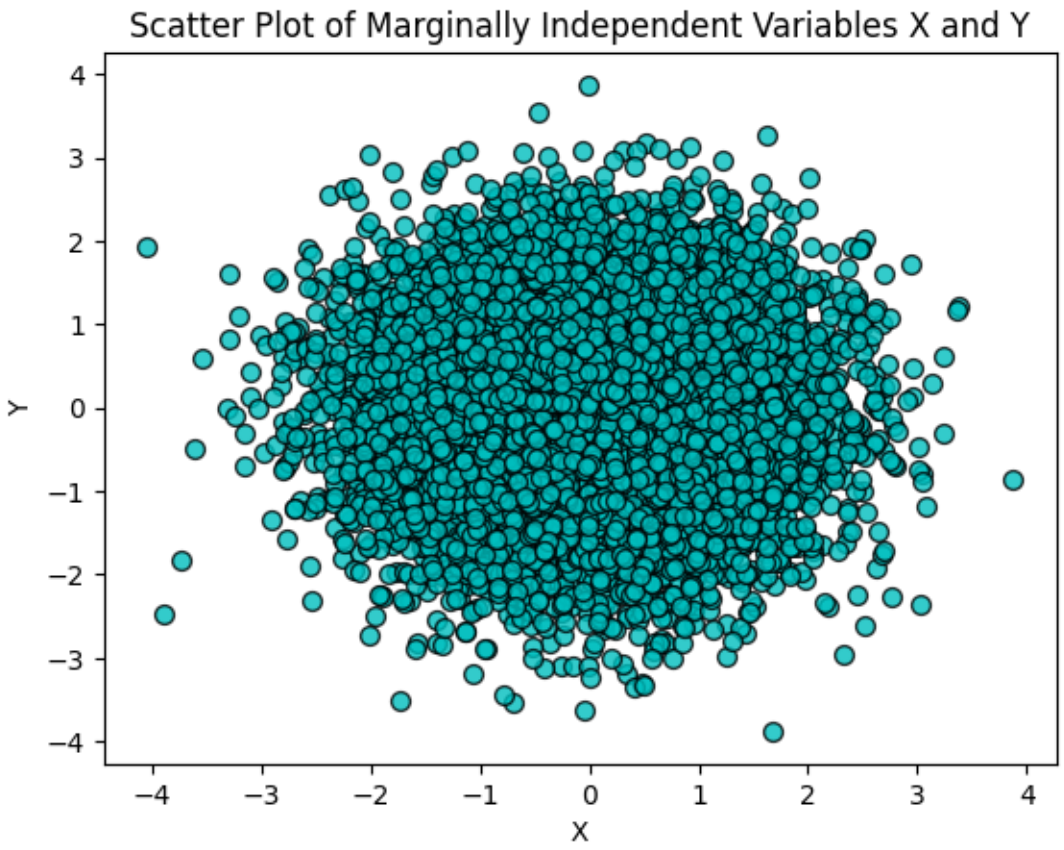
Statistical indepencence refers to the lack of a relationship between two or more random variables. More formally, we can distinguish between two types of statistical independence:

1. **Marginal independence** refers to the lack of a relationship between two random variables, without considering the effect of any other variables. Mathematically, two random variables X and Y are marginally independent ($X \perp Y$) if their joint probability distribution can be expressed as the product of their marginal probability distributions, as in

$$P(X, Y) = P(X)P(Y) \tag{1.1}$$

To illustrate this concept, let's generate two independent random variables X and Y from a normal distribution with mean 0 and standard deviation 1. We then plot their joint distribution using a scatter plot. If X and Y are truly independent, the scatter plot will show no discernible pattern.

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

# Generate two independent random variables
X = np.random.normal(0, 1, 10000)
Y = np.random.normal(0, 1, 10000)

# Plot their joint distribution
plt.scatter(X, Y, alpha=0.8, c='c', edgecolor='k', s=50)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Scatter Plot of Marginally Independent Variables X and Y')
plt.show()
```



2. **Conditional independence** refers to the lack of a relationship between two random variables, given the value of one or more other variables. Mathematically, two random variables X and Y are conditionally independent given a variable ($X \perp Y|Z$) if their conditional probability distribution satisfies

$$P(X, Y|Z) = P(X|Z)P(Y|Z) \tag{1.2}$$

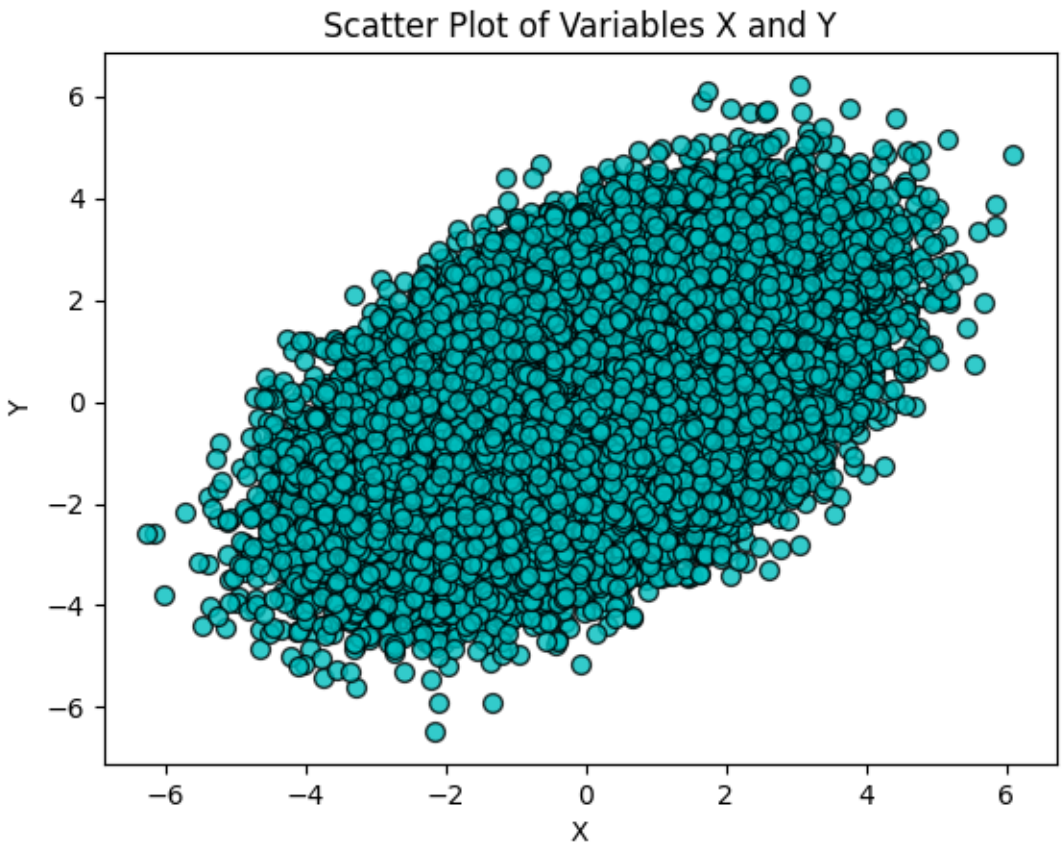
To illustrate this, let's generate three random variables X, Y, and Z such that:

- X and Y are dependent on Z
- X and Y are conditionally independent given Z

This is done by adding noise to Z to generate X and Y.

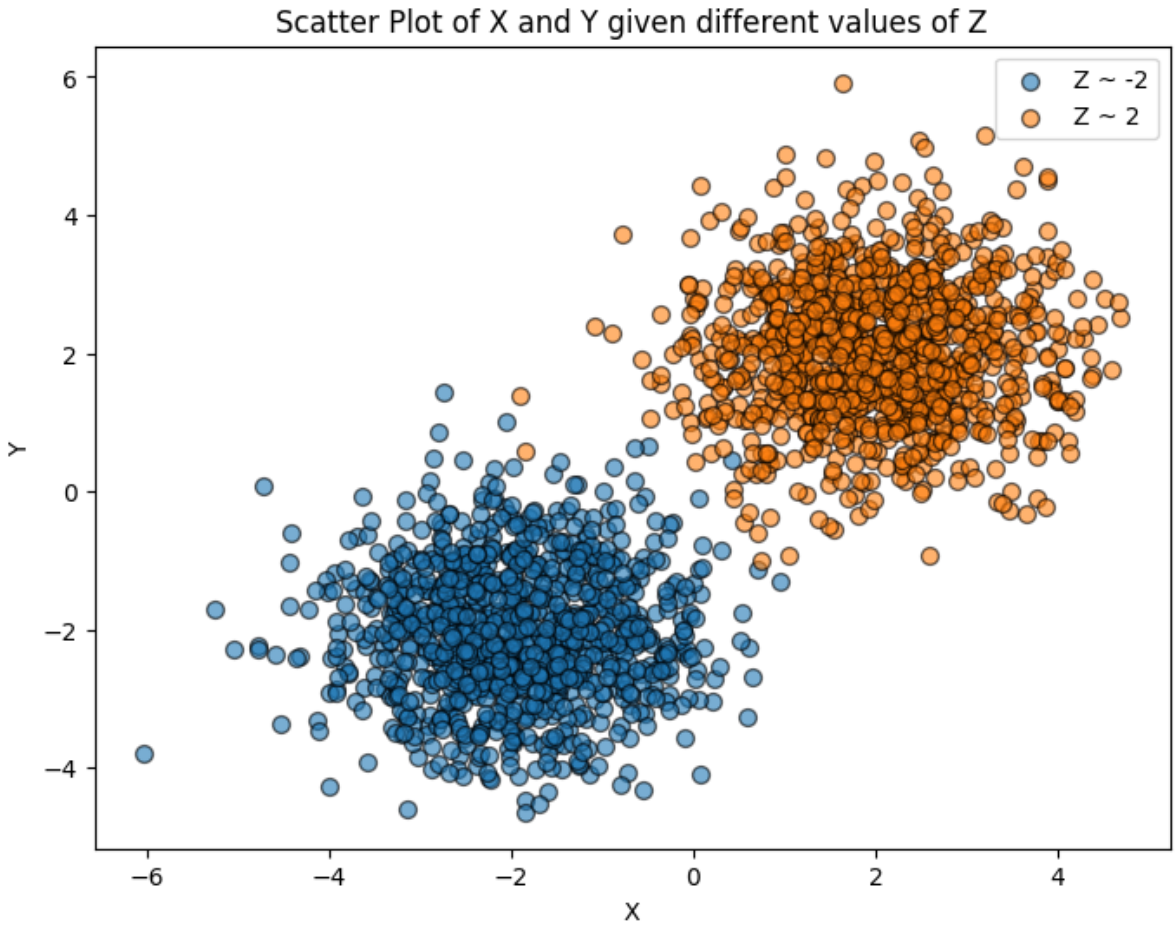
We first plot the scatter plot of X and Y without considering Z, which may show some correlation.

```
# Generate three random variables such that X and Y are independent given Z
Z = np.random.normal(0, 1, 100000)
X = Z + np.random.normal(0, 1, 100000)
Y = Z + np.random.normal(0, 1, 100000)

# Plot X and Y
plt.scatter(X, Y, alpha=0.8, c='c', edgecolor='k', s=50)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Scatter Plot of Variables X and Y')
plt.show()
```



Then, we plot X and Y for different ranges of Z. If X and Y are conditionally independent given Z, the scatter plots for different values of Z should show no pattern, demonstrating that knowing Z removes the dependence between X and Y.

```
# Now, plot X and Y given Z
plt.figure(figsize=(8, 6))
for z_value in [-2, 2]:
    mask = (Z >= z_value - 0.1) & (Z <= z_value + 0.1)
    plt.scatter(X[mask], Y[mask], alpha=0.6, label=f'Z ~ {z_value}', edgecolor='k',␣
 ↪s=50)

plt.xlabel('X')
plt.ylabel('Y')
plt.title('Scatter Plot of X and Y given different values of Z')
plt.legend()
plt.show()
```



1.2 Expectation, variance, and covariance

The expectation (or expected value) of a continuous random variable X with probability density function $p(x)$ is

$$\mathbb{E}[X] = \int xp(x)dx \tag{1.3}$$

while the expectation of a discrete random variable with probability mass function $p(x)$ is

$$\mathbb{E}[X] = \sum_x xp(x) \tag{1.4}$$

The expectation of any function of a random variable, $f(X)$, is given by

$$\mathbb{E}[X] = \int f(x)p(x)dx \tag{1.5}$$

The deviation or fluctuation of X from its expected value is $X - \mathbb{E}[X]$. The variance of a random variable X measures the dispersion around its mean, and it is given by

$$\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] \tag{1.6}$$

The covariance of two random variables X and Y measures the degree to which two random variables change together. If the variables tend to show similar behavior (they tend to be above or below their expected values together), the covariance

is positive. If one variable tends to increase when the other decreases, the covariance is negative. It is given by

$$\text{Cov}[X, Y] = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] \tag{1.7}$$

Some **algebraic properties** of expectation, variance, and covariance will be extremely useful in manipulating and deriving statistical quantities of interest:

- **Linearity of expectations**:
$$\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y] \tag{1.8}$$
Expectation is a linear operator. The expectation of a sum of random variables is the sum of their expectations, and the expectation of a scaled random variable is the scale factor times the expectation of the variable.

- **Variance identity**:
$$\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{1.9}$$
Expanding $\mathbb{E}[(X-\mathbb{E}[X])^2]$ we get $\mathbb{E}[X^2-2X\mathbb{E}[X]+(\mathbb{E}[X])^2]$, which is equal to $\mathbb{E}[X^2]-2\mathbb{E}[X\mathbb{E}[X]]+\mathbb{E}[(\mathbb{E}[X])^2]$. However, $\mathbb{E}[X]$ is a constant (because it is the expected value of a random variable, it is not random anymore), so $\mathbb{E}[\mathbb{E}[X]]$ is just $\mathbb{E}[X]$. So that becomes $\mathbb{E}[X^2] - 2\mathbb{E}[X]\mathbb{E}[X] + (\mathbb{E}[X])^2 = \mathbb{E}[X^2] - 2(\mathbb{E}[X])^2 + (\mathbb{E}[X])^2$.

- **Covariance identity**:
$$\text{Cov}[X, Y] = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \tag{1.10}$$
Expanding the product $(X-\mathbb{E}[X])(Y-\mathbb{E}[Y])$ we get $X-X\mathbb{E}[Y]-Y\mathbb{E}[X]+\mathbb{E}[X]\mathbb{E}[Y]$. Taking the expectation we get $\mathbb{E}[XY]-\mathbb{E}[X\mathbb{E}[Y]]-\mathbb{E}[Y\mathbb{E}[X]]+\mathbb{E}[\mathbb{E}[X]\mathbb{E}[Y]]$. Because the expectation is constant, we get to $\mathbb{E}[XY]-\mathbb{E}[X]\mathbb{E}[Y]-\mathbb{E}[Y]\mathbb{E}[X]+\mathbb{E}[X]\mathbb{E}[Y]$.

- **Covariance is symmetric**:
$$\text{Cov}[X, Y] = \text{Cov}[Y, X] \tag{1.11}$$
The direction of comparison does not matter for covariance, whether you measure how X varies with Y or Y with X, the result is the same.

- **Variance is covariance with itself**:
$$\text{Cov}[X, X] = \text{Var}[X] \tag{1.12}$$
Covariance measures how two variables vary together, and variance is a special case where these two variables are the same.

- **Variance is not linear**:
$$\text{Var}[aX + b] = a^2\,\text{Var}[X] \tag{1.13}$$
The square in the variance formula leads to a squared scale factor when a random variable is scaled. The addition of a constant b does not affect variance, as variance measures dispersion around the mean, which is unaffected by constant shifts. To show why a becomes a^2, we have $\text{Var}[aX] = \mathbb{E}[(aX - \mathbb{E}(aX)^2]$, which is equal to $\mathbb{E}[a^2(X - \mathbb{E}(X)^2] = a^2\mathbb{E}[(X - \mathbb{E}(X)^2]$.

- **Covariance is not linear**:
$$\text{Cov}[aX + b, Y] = a\,\text{Cov}[X, Y] \tag{1.14}$$
Scaling one variable in a covariance relationship scales the covariance itself but does not affect the relationship's direction or absence (signified by zero covariance). The addition of a constant does not affect covariance, as it does not change how one variable varies with another.

- **Variance of a sum**:
$$\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y] + 2\,\text{Cov}[X, Y] \tag{1.15}$$
The variance of a sum includes the individual variances and an additional term to account for how the variables co-vary. This comes from expanding $\mathbb{E}[(X + Y - \mathbb{E}[X + Y])^2]$ and using the linearity of expectations.

- **Variance of a large sum**:

$$\text{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} \text{Cov}[X_i, X_j] = \sum_{i=1}^{n} \text{Var}[X_i] + 2\sum_{i=1}^{n-1}\sum_{j>i} \text{Cov}[X_i, X_j] \tag{1.16}$$

  The variance of a sum of multiple random variables includes both their individual variances and the covariance terms for every pair.

- **Law of total expectations**:

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X|Y]] \tag{1.17}$$

  Suppose we have two random variables X and Y. We want to express the expectation of X (a marginal expectation) in terms of its conditional expectation given Y. his law states that the overall expectation of X can be found by taking the expectation of the conditional expectation of X given Y. In practice, what we are doing is splitting the entire probability space into parts based on the values of Y, calculating the expected value of X (this is $\mathbb{E}[X|Y]$), and then taking the expectation of these conditional expectations over the distribution of Y. Imagine Y as categorizing or segmenting the probability space into different scenarios or groups. Within each group, you calculate the average value of X (this gives you $\mathbb{E}[X|Y = y]$ for each y). Then, you average these averages over all possible groups (weighted by the probability of each group $Y = y$, leading back to the overall average of X. This law is particularly useful in scenarios where direct calculation of $\mathbb{E}[X]$ is complex but where conditional expectations $\mathbb{E}[X|Y]$ are simpler to compute. This is the expected value (or mean) of X given a particular value of Y. In many statistical models, especially in predictive modeling, this conditional mean can be thought of as a ``prediction" of X based on the knowledge of Y. For example, if Y represents a set of features or conditions, then $\mathbb{E}[X|Y]$ is our best guess or prediction of X under those conditions.

- **Law of total variance**:

$$\text{Var}[X] = \text{Var}[\mathbb{E}[X|Y]] + \mathbb{E}[\text{Var}[X|Y]] \tag{1.18}$$

  This law decomposes the total variance into two parts. The first part can be thought of as between-group variability and measures how much the conditional means vary as Y changes. The second term is the within-group variability and represents the average of the variances within each group defined by Y.

- **Independence implies zero covariance**:

$$X \perp Y \rightarrow \text{Cov}[X, Y] = 0 \tag{1.19}$$

  Independence between two variables means the occurrence of one does not affect the probability distribution of the other. This lack of influence translates mathematically to zero covariance. However, the converse is not necessarily true as zero covariance does not capture nonlinear dependencies.

1.3 Convergence

The **law of large numbers (LLN)** states that, for a sequence of independent and identically distributed (i.i.d.) random variables $X_1, X_2 \ldots, X_n$ each with expected value $\mathbb{E}[X]$, the sample mean converges to the expected value as n approaches infinity

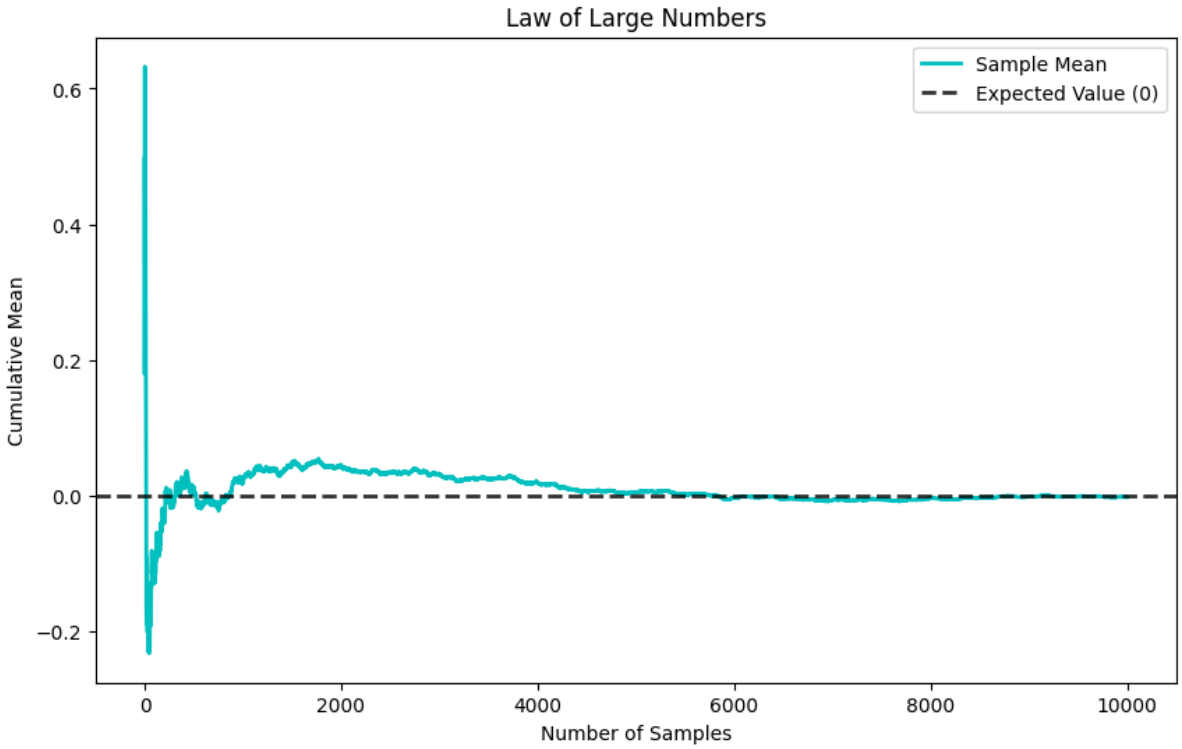
$$\frac{1}{n}\sum_{i=1}^{n} X_i \rightarrow \mathbb{E}[X] \quad \text{as } n \rightarrow \infty \tag{1.20}$$

The **i.i.d. assumption** is a fundamental concept in probability theory and statistics, with significant implications. Independence implies that the occurrence of one event or the value of one variable does not influence the occurrence of another. In the context of random variables, $X_1, X_2 \ldots, X_n$ being independent means the outcome of X_i provides no information about the outcome of X_j, $i \neq j$. Identically distributed means that each of the random variables has the same probability distribution. They do not need to take on the same value, but the rules governing their behavior (i.e., the likelihood of each outcome) are identical. In the context of supervised learning, the i.i.d. assumption assumes that the training and test data are independently drawn from the same underlying probability distribution. This enables the use of

statistical tools and techniques, such as maximum likelihood estimation and hypothesis testing, which are based on the assumption of independent and identically distributed data. In real-world data, this assumption is often violated as data may be dependent and non-identically distributed due to distribution shifts across geography or time, sampling practices, or the presence of confounding variables and selection bias.

To illustrate the LLN, we can generate a sequence of independent and identically distributed (i.i.d.) random variables and observe how their sample mean converges to the expected value as the number of samples increases.

```
# Generate a sequence of i.i.d. random variables
np.random.seed(42)
n = 10000
X = np.random.normal(loc=0, scale=1, size=n)

# Compute the cumulative mean
cumulative_mean = np.cumsum(X) / np.arange(1, n + 1)

# Plot the cumulative mean
plt.figure(figsize=(10, 6))
plt.plot(cumulative_mean, label='Sample Mean', lw=2, c='c')
plt.axhline(y=0, color='k', linestyle='--', label='Expected Value (0)', lw=2, alpha=.
 ↪8)
plt.xlabel('Number of Samples')
plt.ylabel('Cumulative Mean')
plt.title('Law of Large Numbers')
plt.legend()
plt.show()
```



The **central limit theorem (CLT)** states that if $X_1, X_2, \ldots, X_n$ are i.i.d. random variables with an expected value $\mathbb{E}[X]$ and a finite variance $\text{Var}[X]$, the distribution of the sample mean

$$\bar{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{1.21}$$

approaches a normal distribution as $n \to \infty$. Specifically, the standardized form

$$\frac{\bar{X}_n - \mathbb{E}[X]}{\sqrt{\text{Var}[X]/n}} \tag{1.22}$$

approaches the standard normal distribution $N(0, 1)$.

The motivation behind the CLT lies in its ability to provide a predictable and well-understood behavior (the normal distribution) for averages of random variables, regardless of the original distribution of these variables. This is particularly useful in practical scenarios such as statistical sampling and hypothesis testing, where it is often necessary to make inferences about population parameters. The intuition of the CLT is that as we increase the number of random variables in our sample, the peculiarities and individual randomness of each variable tend to cancel out. This leads to the emergence of the normal distribution, which is symmetric and centered around the mean. The CLT is powerful because it applies to a wide range of distributions, whether they are symmetric, skewed, or even arbitrary, as long as the variables are i.i.d. with a finite variance.

1.4 Estimation: bias and variance

When observing values $X_1, X_2, \ldots, X_n$ from a distribution, the true nature of this distribution is often unknown. In many cases, we are interested in estimating a parameter θ of this distribution, such as the mean or variance. This process involves several key concepts:

- **Statistic**: a function of the observed data, or the data alone.
- **Estimator**: a rule or a function that tells you how to infer or guess the value of a parameter θ, or some function of it, denoted as $h(\theta)$. Suppose you want to estimate the population mean. The sample mean (denoted usually as $\bar{X}$) is an estimator. It is a function that calculates the mean of your sample data.
- **Estimand**: the quantity that we want to estimate.
- **Estimate**: the actual numerical value obtained by applying the estimator to your data. It is an approximation of some estimand, which we get using data.

We typically denote an estimator of θ as $\hat{\theta}_n$, where the hat symbol signifies that it approximates the true parameter, and the subscript n indicates its dependence on the sample size. An estimator is itself a random variable because it is a function of random data. Its distribution, known as the sampling distribution, depends on the distribution of the data X_i. A desirable property of an estimator is consistency, which means that $\hat{\theta}_n$ converges to θ as $n \to \infty$. An estimator that fails to be consistent is generally not desirable. Two important properties of estimators are:

1. **Bias**: the difference between the expected value of the estimator and the true parameter value. An estimator is unbiased if $\mathbb{E}[\hat{\theta}_n] = \theta$ for all θ.
2. **Variance**: a measure of the spread of the estimator's sampling distribution. The variance of an estimator indicates how much the estimator varies from sample to sample. The square root of this variance is called the standard error, which provides a measure of how precise our estimate is.

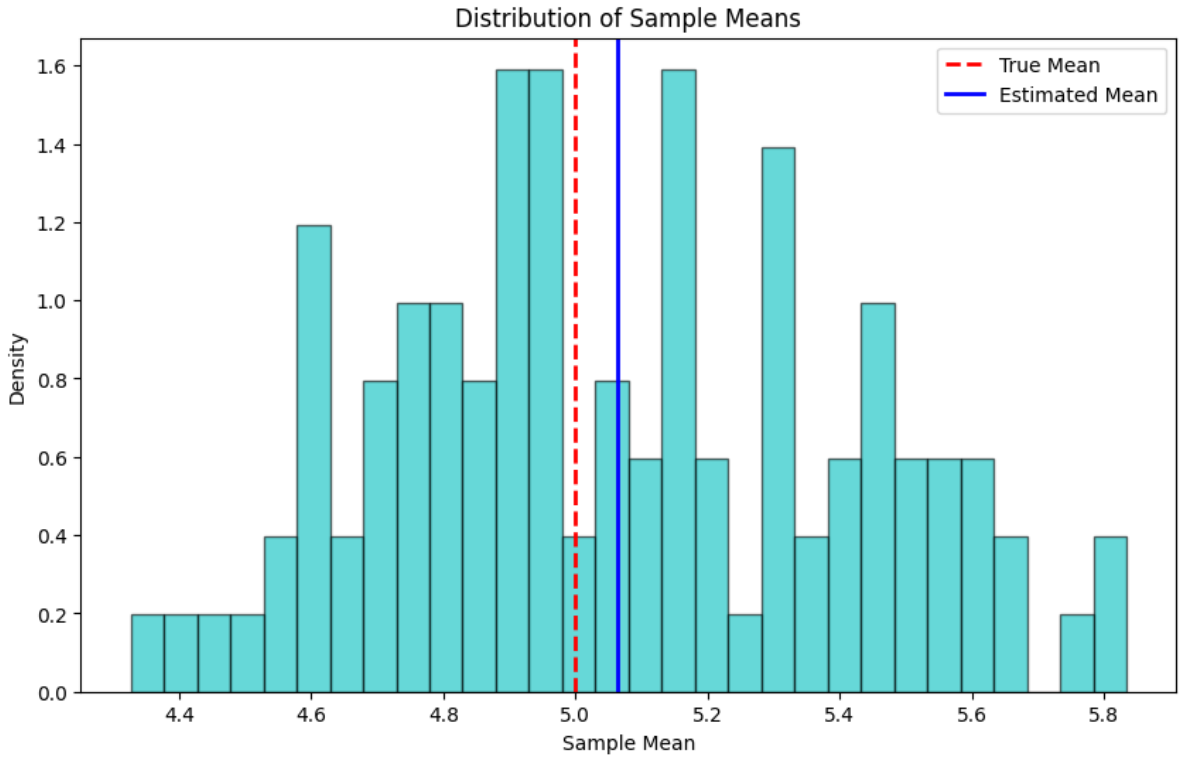
Let's now provide an example to illustrate the concepts of bias and variance in the context of estimating the mean of a normal distribution. We assume a true mean (μ) of 5 and generate 100 samples, each consisting of 30 observations drawn from a normal distribution with this true mean and a standard deviation of 2. For each sample, we compute the **sample mean**, which serves as our estimator for the population mean.

The sample means are then used to evaluate the bias and variance of the estimator:

- The bias is calculated as the difference between the average of the sample means (the expected value of the estimator) and the true mean. A low bias indicates that the estimator is accurate on average
- The variance is a measure of the spread of the sample means around their average. A low variance indicates that the estimator is reliable and produces similar estimates across different samples.

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

# True parameter
true_mean = 5

# Generate multiple samples
np.random.seed(42)
n_samples = 100
sample_size = 30
samples = [np.random.normal(loc=true_mean, scale=2, size=sample_size) for _ in␣
↪range(n_samples)]

# Calculate sample means
sample_means = np.array([np.mean(sample) for sample in samples])

# Calculate bias and variance
estimated_mean = np.mean(sample_means)
bias = estimated_mean - true_mean
variance = np.var(sample_means)

# Print results
print(f"True Mean: {true_mean}")
print(f"Estimated Mean: {estimated_mean}")
print(f"Bias: {bias}")
print(f"Variance: {variance}")

# Plot the distribution of sample means
plt.figure(figsize=(10, 6))
plt.hist(sample_means, bins=30, density=True, alpha=0.6, color='c', edgecolor='k')
plt.axvline(true_mean, color='r', linestyle='--', linewidth=2, label='True Mean')
plt.axvline(estimated_mean, color='b', linestyle='-', linewidth=2, label='Estimated␣
↪Mean')
plt.xlabel('Sample Mean')
plt.ylabel('Density')
plt.title('Distribution of Sample Means')
plt.legend()
plt.show()
```

```
True Mean: 5
Estimated Mean: 5.06400167175167
Bias: 0.0640016717516696
Variance: 0.12043348413823464
```



The histogram of the sample means provides a visual representation of the estimator's sampling distribution. The true mean is marked by a red dashed line, and the average of the sample means (the estimated mean) is shown with a solid blue line. This visualization helps to see how the sample means are distributed around the true mean, illustrating the concepts of bias (how far the average estimate is from the true value) and variance (how spread out the estimates are).

1.4.1 Main Probability Distributions

Normal Distribution

The normal distribution, also known as the Gaussian distribution, is central in statistics due to its symmetric, bell-shaped curve. It is characterized by its mean μ and standard deviation σ, with the probability density function (PDF) given by

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1.23}$$

The significance of the normal distribution arises from the Central Limit Theorem (CLT), which states that sums of independent random variables converge to a normal distribution, regardless of the original distribution of these variables, making it applicable in a wide array of scenarios. This property underscores the normal distribution's role in approximating the behavior of various real-world random processes.

Chi-Squared Distribution

The chi-squared distribution is another key distribution in statistics, especially in hypothesis testing and confidence interval estimation. It arises as the sum of the squares of independent standard normal variables. Specifically, if $Z_1, Z_2, ..., Z_k$ are independent standard normal random variables, then $\sum_{i=1}^{k} Z_i^2$ follows a chi-squared distribution with k degrees of freedom. Its PDF for $x > 0$ and k degrees of freedom is

$$f(x; k) = \frac{x^{k/2-1} e^{-x/2}}{2^{k/2}\Gamma(k/2)} \tag{1.24}$$

This distribution is asymmetric and skewed to the right, with its shape and spread depending on the degrees of freedom k. It is primarily used in the chi-squared test for independence and goodness of fit, and in estimating variances of normal distributions.

F-Distribution

The F-distribution is crucial in the context of variance analysis and hypothesis testing. It is the ratio of two scaled chi-squared distributions: if U follows a chi-squared distribution with d_1 degrees of freedom and V follows an independent chi-squared distribution with d_2 degrees of freedom, then the ratio $\frac{U/d_1}{V/d_2}$ follows an F-distribution. Its PDF is described by

$$f(x; d_1, d_2) = \frac{\sqrt{\frac{(d_1 x)^{d_1} d_2^{d_2}}{(d_1 x + d_2)^{d_1+d_2}}}}{x\mathrm{B}\left(\frac{d_1}{2}, \frac{d_2}{2}\right)} \tag{1.25}$$

where d_1 and d_2 are the degrees of freedom. The distribution is non-symmetric, bounded at the left by 0, and its shape varies with the degrees of freedom. It is particularly useful in comparing variances between two samples, as in ANOVA and regression analysis.

Student's t-Distribution

The Student's t-distribution arises when estimating the mean of a normally distributed population in situations where the sample size is small and the population standard deviation is unknown. It is defined by the PDF

$$f(t) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{t^2}{\nu}\right)^{-\frac{\nu+1}{2}} \tag{1.26}$$

where ν denotes degrees of freedom. The t-distribution resembles the normal distribution but has heavier tails, meaning it is more prone to producing values that fall far from its mean. This property makes it particularly useful in hypothesis testing and constructing confidence intervals when the sample size is small. As the sample size increases, the t-distribution approaches the normal distribution, illustrating the connection between them.

The Student's t-distribution is particularly useful when dealing with small sample sizes or when the population variance is unknown. It is defined as the distribution of the ratio of a standard normal random variable Z (with mean 0 and variance 1) and the square root of a chi-square random variable X divided by its degrees of freedom v, i.e.,

$$T = \frac{Z}{\sqrt{X/v}} \tag{1.27}$$

where Z follows a standard normal distribution and X follows a chi-square distribution with v degrees of freedom. The resulting T follows a Student's t-distribution with v degrees of freedom.

This distribution is symmetric and bell-shaped like the normal distribution but has heavier tails, meaning it is more prone to producing values that fall far from its mean. This property makes the t-distribution particularly suitable for small sample sizes, as it accounts for the increased uncertainty that comes with fewer observations.

The t-distribution is central to many statistical tests, including the t-test for assessing the statistical significance of the difference between two sample means, the construction of confidence intervals for the mean of a normally distributed population when the standard deviation is unknown, and in linear regression analysis.

The relationship between the normal distribution, the chi-square distribution, and the t-distribution highlights the importance of understanding how distributions can be related and transformed into each other, providing a powerful framework for statistical inference.

Exponential Distribution

The exponential distribution models the time between events in processes with a constant rate of occurrence and is pivotal in reliability analysis and queuing theory. Its memoryless property implies that the probability of an event occurring in the next instant is independent of how much time has already elapsed. The PDF is

$$f(x; \lambda) = \lambda e^{-\lambda x} \tag{1.28}$$

for $x \geq 0$. This distribution describes the time until an event like failure or arrival occurs and is widely used in survival analysis and reliability engineering.

Binomial Distribution

The binomial distribution is fundamental in modeling binary outcomes and represents the number of successes in a fixed number of independent Bernoulli trials. The PDF is

$$f(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{1.29}$$

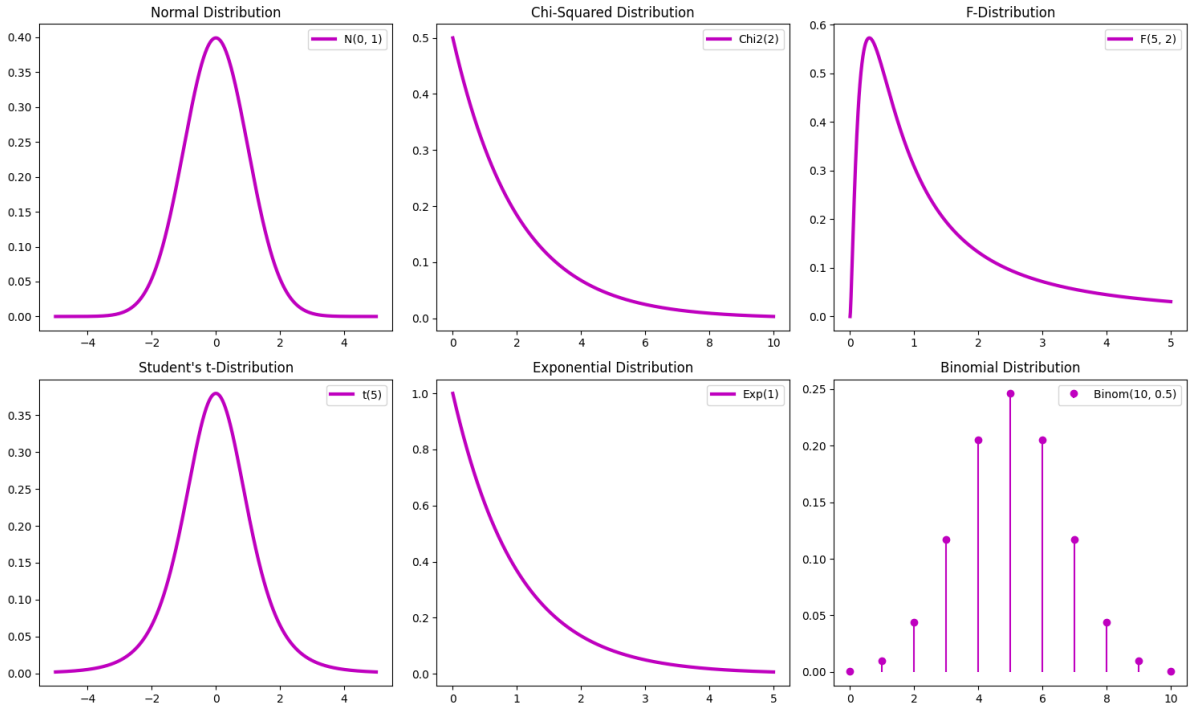
where k is the number of successes, n the number of trials, and p the probability of success. The shape of the binomial distribution can be symmetric or skewed depending on the values of n and p. It is extensively used in scenarios like quality control, survey analysis, and clinical trials, providing a model for situations where outcomes are binary and probabilistically independent.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

# Set up the figure
fig, axes = plt.subplots(2, 3, figsize=(15, 9))

# Normal Distribution
mu, sigma = 0, 1
x = np.linspace(-5, 5, 1000)
y = stats.norm.pdf(x, mu, sigma)
axes[0, 0].plot(x, y, label=f'N({mu}, {sigma**2})', lw=3, c='m')
axes[0, 0].set_title('Normal Distribution')
axes[0, 0].legend()

# Chi-Squared Distribution
df = 2
x = np.linspace(0, 10, 1000)
y = stats.chi2.pdf(x, df)
axes[0, 1].plot(x, y, label=f'Chi2({df})', lw=3, c='m')
axes[0, 1].set_title('Chi-Squared Distribution')
axes[0, 1].legend()

# F-Distribution
d1, d2 = 5, 2
x = np.linspace(0, 5, 1000)
y = stats.f.pdf(x, d1, d2)
axes[0, 2].plot(x, y, label=f'F({d1}, {d2})', lw=3, c='m')
axes[0, 2].set_title('F-Distribution')
axes[0, 2].legend()

# Student's t-Distribution
df = 5
x = np.linspace(-5, 5, 1000)
y = stats.t.pdf(x, df)
axes[1, 0].plot(x, y, label=f't({df})', lw=3, c='m')
axes[1, 0].set_title('Student\'s t-Distribution')
axes[1, 0].legend()

# Exponential Distribution
lambda_ = 1
x = np.linspace(0, 5, 1000)
```

(continues on next page)

(continued from previous page)

```
y = stats.expon.pdf(x, scale=1/lambda_)
axes[1, 1].plot(x, y, label=f'Exp({lambda_})', lw=3, c='m')
axes[1, 1].set_title('Exponential Distribution')
axes[1, 1].legend()

# Binomial Distribution
n, p = 10, 0.5
x = np.arange(0, n + 1)
y = stats.binom.pmf(x, n, p)
axes[1, 2].stem(x, y, basefmt=" ", label=f'Binom({n}, {p})', linefmt='m')
axes[1, 2].set_title('Binomial Distribution')
axes[1, 2].legend()

plt.tight_layout()
plt.show()
```



Explanation of plots:

1. **Normal distribution:** the plot shows the symmetric bell-shaped curve of the normal distribution with mean 0 and standard deviation 1.
2. **Chi-Squared distribution:** the plot illustrates the chi-squared distribution with 2 degrees of freedom, showing its right-skewed nature.
3. **F-distribution:** the plot shows the F-distribution with degrees of freedom 5 and 2, highlighting its right-skewed shape.
4. **Student's t-distribution:** the plot depicts the t-distribution with 5 degrees of freedom, showing its similarity to the normal distribution but with heavier tails.
5. **Exponential distribution:** the plot displays the exponential distribution with a rate parameter of 1, showing its memoryless property and right-skewed shape.

6. **Binomial distribution:** the plot illustrates the binomial distribution with 10 trials and a success probability of 0.5, showing the discrete probability mass function.

These visualizations help in understanding the different probability distributions and their properties, which are fundamental in various statistical analyses and applications.

1.5 Hypothesis testing framework

Hypothesis testing is a fundamental framework in statistics used to determine whether there is enough evidence in a sample of data to infer that a certain condition holds for the entire population. In hypothesis testing, two contradictory hypotheses about a population parameter are considered: the null hypothesis (H_0) and the alternative hypothesis (H_1). The null hypothesis represents a default position or a statement of no effect or no difference. The alternative hypothesis represents what we want to prove or establish.

A test statistic is calculated from the sample data and is used to assess the truth of the null hypothesis. The choice of test statistic depends on the nature of the data and the hypothesis being tested. The P-value is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the assumption that the null hypothesis is true. A smaller P-value indicates that the observed data is less likely under the null hypothesis. Based on the P-value and a predetermined significance level (usually denoted as α, commonly set at 0.05), a decision is made: if the P-value is less than α, the null hypothesis is rejected in favor of the alternative hypothesis; if the P-value is greater than α, there is not enough evidence to reject the null hypothesis.

In hypothesis testing, two types of errors can occur. Type I error corresponds to rejecting the null hypothesis when it is actually true (false positive). The probability of making a Type I error is α. Type II error is failing to reject the null hypothesis when the alternative hypothesis is true (false negative).

Hypothesis testing is a critical tool in statistics for making inferences about populations based on sample data. It allows researchers to test assumptions and make decisions based on statistical evidence. The goal of hypothesis testing is not to prove the null hypothesis but to assess the strength of evidence against it. Hypothesis testing is a foundational concept in statistics used to infer the properties of a population based on sample data. The choice of distribution for conducting a hypothesis test depends on the nature of the data, the size of the sample, and the assumptions that can be made about the population. Below, we explore various hypothesis tests, the distributions used, and the rationale for their use.

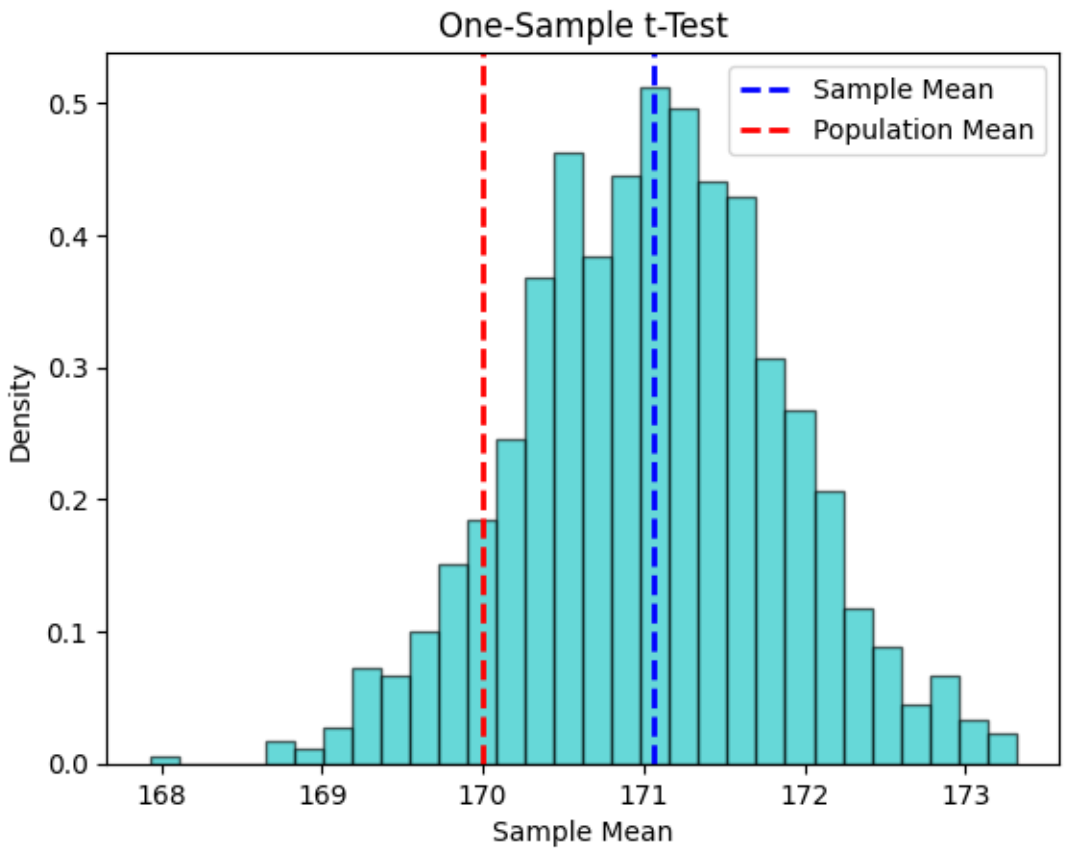
1.5.1 Comparing Means

- **Z-test:** Used when comparing the mean of a sample to a known population mean, or comparing the means of two large independent samples. The Z-test is applicable when the population variance is known and the sample size is large (typically $n > 30$). The normal distribution is used due to the Central Limit Theorem (CLT), which states that the sampling distribution of the sample mean will approximate a normal distribution as the sample size becomes large, regardless of the population's distribution.
- **T-test:** Used when the population variance is unknown and the sample size is small, the t-distribution is used. The t-test is more accommodating of the uncertainty in the sample estimate of the variance, providing more accurate confidence intervals and P-values. The t-distribution converges to the normal distribution as the sample size increases.
  - **One-sample t-test:** Compares the mean of a single sample to a known mean.
  - **Two-sample t-test:** Compares the means of two independent samples.
  - **Paired t-test:** Compares means from the same group at different times or under different conditions.

Let'see an example of a one-sample t-test to check if the mean height of a sample of people is different from a known population mean of 170 cm.

```
# Sample data
np.random.seed(42)
sample_heights = np.random.normal(loc=172, scale=5, size=30)

# Perform one-sample t-test
population_mean = 170
t_statistic, p_value = stats.ttest_1samp(sample_heights, population_mean)

# Significance level
alpha = 0.05

# Output the results
print(f"T-statistic: {t_statistic:.3f}, P-value: {p_value:.3f}")
print(f"Significance level (alpha): {alpha}")

# Automated conclusion based on P-value
if p_value < alpha:
    conclusion = (f"Reject the null hypothesis. There is sufficient evidence to␣
 ↪conclude that the mean height of the "
                  f"sample is significantly different from the population mean of 170␣
 ↪cm at the {alpha} significance level.")
else:
    conclusion = (f"Fail to reject the null hypothesis. There is not sufficient␣
 ↪evidence to conclude that the mean height "
                  f"of the sample is significantly different from the population mean␣
 ↪of 170 cm at the {alpha} significance level.")

print(conclusion)

# Plotting the distribution of sample means
sample_means = [np.mean(np.random.choice(sample_heights, size=30, replace=True)) for _
 ↪ in range(1000)]
plt.hist(sample_means, bins=30, density=True, alpha=0.6, color='c', edgecolor='black')
plt.axvline(np.mean(sample_heights), color='b', linestyle='dashed', linewidth=2,␣
 ↪label='Sample Mean')
plt.axvline(population_mean, color='r', linestyle='--', linewidth=2, label=
 ↪'Population Mean')
plt.xlabel('Sample Mean')
plt.ylabel('Density')
plt.title('One-Sample t-Test')
plt.legend()
plt.show()
```

```
T-statistic: 1.289, P-value: 0.207
Significance level (alpha): 0.05
Fail to reject the null hypothesis. There is not sufficient evidence to conclude␣
 ↪that the mean height of the sample is significantly different from the␣
 ↪population mean of 170 cm at the 0.05 significance level.
```



Let's also see an example of the two-sample t-test to compare the mean heights of two independent samples and determine if there is a statistically significant difference between the two population means. Using a significance level of 0.05, the two-sample t-test calculates the t-statistic and P-value to assess the null hypothesis that the means of the two samples are equal. If the P-value is less than 0.05, the null hypothesis is rejected, indicating a significant difference between the sample means.

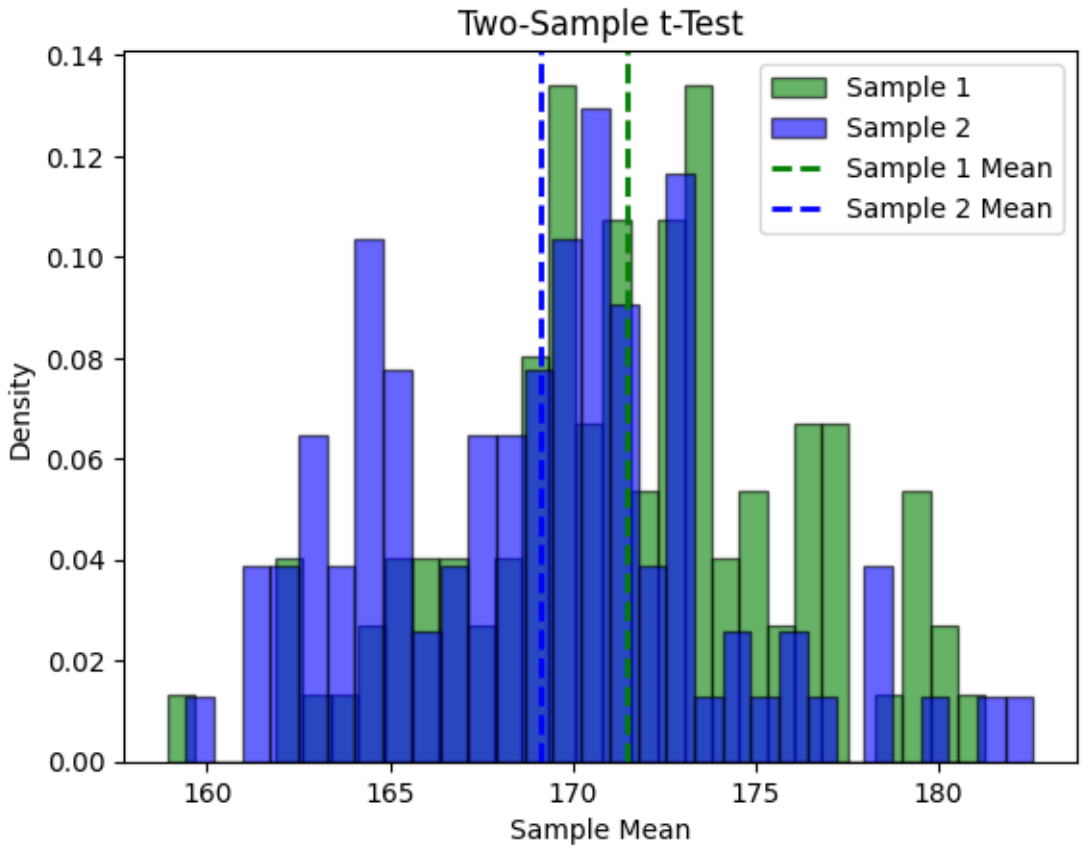
```
# Sample data
np.random.seed(42)
sample1 = np.random.normal(loc=172, scale=5, size=100)
sample2 = np.random.normal(loc=169, scale=5, size=100)

# Perform two-sample t-test
t_statistic, p_value = stats.ttest_ind(sample1, sample2)

# Significance level
alpha = 0.05

# Output the results
print(f"T-statistic: {t_statistic:.3f}, P-value: {p_value:.3f}")
print(f"Significance level (alpha): {alpha}")

# Automated conclusion based on P-value
if p_value < alpha:
    conclusion = (f"Reject the null hypothesis. There is sufficient evidence to␣
↪conclude that the mean heights of the "
                  f"two samples are significantly different at the {alpha}␣
↪significance level.")
```

(continues on next page)

(continued from previous page)

```
else:
    conclusion = (f"Fail to reject the null hypothesis. There is not sufficient␣
 ↪evidence to conclude that the mean heights "
                  f"of the two samples are significantly different at the {alpha}␣
 ↪significance level.")

print(conclusion)

# Plotting the distribution of sample means
plt.hist(sample1, bins=30, density=True, alpha=0.6, color='g', edgecolor='black',␣
 ↪label='Sample 1')
plt.hist(sample2, bins=30, density=True, alpha=0.6, color='b', edgecolor='black',␣
 ↪label='Sample 2')
plt.axvline(np.mean(sample1), color='g', linestyle='dashed', linewidth=2, label=
 ↪'Sample 1 Mean')
plt.axvline(np.mean(sample2), color='b', linestyle='dashed', linewidth=2, label=
 ↪'Sample 2 Mean')
plt.xlabel('Sample Mean')
plt.ylabel('Density')
plt.title('Two-Sample t-Test')
plt.legend()
plt.show()
```

```
T-statistic: 3.598, P-value: 0.000
Significance level (alpha): 0.05
Reject the null hypothesis. There is sufficient evidence to conclude that the mean␣
 ↪heights of the two samples are significantly different at the 0.05 significance␣
 ↪level.
```



1.5.2 Comparing Variances

- **F-test:** Used in the analysis of variance (ANOVA) and for comparing the variances of two samples. The F-distribution arises naturally when comparing the ratio of two variances, each of which follows a chi-squared distribution when the underlying population is normally distributed. The F-test assesses whether the groups have the same variance, an assumption often required in ANOVA and regression analysis. ANOVA is used to compare the means of three or more samples. The F-distribution is used in ANOVA to compare the ratio of the variance explained by the model to the variance within the groups. This test helps to determine if there are significant differences between the means of the groups.

1.5.3 Regression Analysis

- **T-tests for regression coefficients:** To determine if individual predictors are significantly related to the dependent variable, t-tests are used, leveraging the t-distribution. This is because the estimates of the coefficients have distributions that are best modeled by the t-distribution, especially with small sample sizes.
- **F-test for overall model significance:** The F-test is used to assess whether at least one predictor variable has a non-zero coefficient, indicating that the model provides a better fit to the data than a model with no predictors. This test uses the F-distribution, comparing the model's explained variance to the unexplained variance.

1.5.4 Goodness of Fit and Independence Tests

- **Chi-squared test:** Used for categorical data to assess how likely it is that an observed distribution is due to chance. It is used in goodness-of-fit tests to compare the observed distribution to an expected distribution, and in tests of independence to evaluate the relationship between two categorical variables in a contingency table. The chi-squared distribution is used because the test statistic follows this distribution under the null hypothesis.
- **Non-parametric tests:** Used when the assumptions about the population distribution are not met. These tests do not rely on the normality assumption and often use ranking methods or resampling techniques. Examples include the Mann-Whitney U test, Wilcoxon signed-rank test, and Kruskal-Wallis H test.

1.6 Bayesian Learning

Statistical methods can be broadly divided into two macro-categories: frequentist and Bayesian. The frequentist approach views parameters as fixed but unknown quantities, uses data to estimate these parameters, and makes point estimates (i.e., a single best guess) for these parameters. In contrast, the Bayesian approach views parameters as random variables, uses data and prior beliefs (prior distributions) to update our beliefs about these parameters, and results in a probability distribution over the parameters, capturing the uncertainty.

One key difference is that Bayesian statistics treats probability as a measure of belief or certainty rather than frequency. This means probabilities are subjective and can be updated as new information becomes available. In frequentist statistics, probability is interpreted as the long-run frequency of events, relying on the concept of an infinite sequence of repeated trials. Bayesian methods incorporate prior knowledge or beliefs through the use of prior probability distributions, while frequentist methods make inferences solely from the data at hand.

Bayesian learning is formalized using **Bayes' theorem**:

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}$$

where:

- $P(\theta|X)$ is the posterior distribution of the parameters given the data.
- $P(X|\theta)$ is the likelihood of the data given the parameters. It represents the probability of observing the data X given a particular set of parameters θ.
- $P(\theta)$ is the prior distribution of the parameters (our beliefs before seeing the data).
- $P(X)$ is the evidence or marginal likelihood. It is the probability of the data over all possible parameter values, acting as a normalizing constant to ensure the posterior distribution sums (or integrates) to 1.

Bayes' Theorem allows us to update our initial beliefs or probabilities $P(\theta)$ in light of new evidence (X). In other words, it provides a way to revise existing predictions or hypotheses given new or additional information.

Estimating the posterior distribution

Computing the posterior distribution means determining the probability distribution of the parameters of a model given the observed data. This involves updating our beliefs about possible parameter values based on the evidence provided by the data. The main challenge in Bayesian learning is computing the posterior distribution, especially for complex models. This is where methods like Markov chain Monte Carlo (MCMC) come into play. Because of these challenges, we often resort to methods like sampling (e.g., MCMC) or approximations to estimate the posterior distribution, rather than computing it exactly. The goal is to get a representation of the distribution that lets us make informed decisions about the likely values of the parameters given the data.

Estimating the posterior distribution is the core of Bayesian inference. After observing data, we combine our prior beliefs with the likelihood of the observed data to compute the posterior distribution. The shape of this distribution reflects our updated beliefs about the parameters given the data. Once we have the posterior distribution, we can draw samples from

it. Each sample represents a plausible value of the parameter(s) given our prior beliefs and the observed data. By looking at the spread and distribution of these samples, we can understand the uncertainty associated with our estimates.

In many situations, especially with complex models, the posterior distribution might not have a simple analytical form. In these cases, we cannot just "look" at the posterior directly. Instead, we use sampling techniques (like MCMC methods) to draw samples from the posterior, even if we cannot describe the posterior in a simple equation. These samples then serve multiple purposes:

- **Uncertainty estimation:** the spread and distribution of the samples give a sense of how uncertain we are about our parameter estimates.
- **Predictive modeling:** we can use the samples to make predictions for new data and to get a sense of uncertainty in those predictions.
- **Model checking:** we can compare the predictions of our model (using the posterior samples) to the actual observed data to see if our model is a good fit.
- **Decision making:** in practical scenarios, decisions might be based on the posterior samples, especially when we need to consider the uncertainty in our estimates.

In summary, while the posterior distribution encapsulates our updated beliefs after seeing data, sampling from the posterior allows us to quantify, explore, and make decisions based on the uncertainty in those beliefs.

1.6.1 Markov chain Monte Carlo (MCMC)

MCMC algorithms are used to approximate complex probability distributions. They are especially useful in Bayesian statistics when direct computation of the posterior distribution is challenging. The basic idea is to generate samples from a complex distribution, which is too intricate to tackle directly. Instead of computing it exactly, we generate samples that come from that distribution. Over time, the distribution of these samples will closely match the target distribution.

The key concepts of MCMC are two-fold:

- **Markov chain:** a sequence of random samples where each sample depends only on the one before it. It is like a random walk where each step is influenced only by the current position.
- **Monte Carlo:** a technique where random sampling is used to get numerical results for problems that might be deterministic in principle. The name originates from the Monte Carlo Casino, as it relies on randomness.

The main steps of MCMC algorithms are:

- **Initialization:** start at a random position (a random parameter value).
- **Proposal:** at each step, propose a new position based on the current one. This can be a random jump, but it is typically a small move.
- **Acceptance:** decide whether to move to the proposed position. If the new position is a better fit to the data (higher posterior probability), we will likely accept it. If it is worse, we might still accept it but with a lower probability. This decision process ensures we explore the whole space but spend more time in high-probability areas.
- **Iteration:** repeat the proposal and acceptance steps many times. The more steps, the better the approximation will be.
- **Burn-in:** the initial samples might not be representative because the chain might start far from a high-probability area. So, we discard an initial set of samples, a process called "burn-in".

How do we determine if the posterior distribution is higher if we do not have an analytical form? This is the key idea behind MCMC methods (like the Metropolis-Hastings algorithm). We do not need to know the exact value of the posterior distribution; we only need to know it up to a constant of proportionality. In many cases, while the full posterior is hard to compute (due to the difficulty in calculating the normalization constant), its unnormalized version is computable.

Remember the basic Bayes' formula:

$$\text{posterior} \propto \text{likelihood} \times \text{prior}$$

In many applications, we can compute the product of the likelihood and the prior for any given set of parameters, but we might not be able to easily normalize it to get a true probability distribution. So, when deciding whether to accept a new proposed position in MCMC, we first compute the unnormalized posterior at the current position (which is the product of the likelihood and the prior). Then we compute the unnormalized posterior at the proposed new position. Finally, we compare these values. If the unnormalized posterior is higher at the new position, then it means the true posterior is also higher there. Even if we cannot say exactly what the posterior value is at that position, we can still determine if it is higher or lower than at the current position. This relative comparison, rather than an absolute value, is what drives the decision to accept or reject the new proposed position. For the case where the proposed position has a lower unnormalized posterior value, the Metropolis-Hastings algorithm provides a rule to accept it with a probability proportional to the ratio of the unnormalized posteriors (proposed to current). This ensures exploration of the entire parameter space, preventing the algorithm from getting stuck in local modes.

In essence, MCMC is a systematic way to "wander around" in a parameter space to understand a probability distribution, especially when direct computation is difficult or impossible.

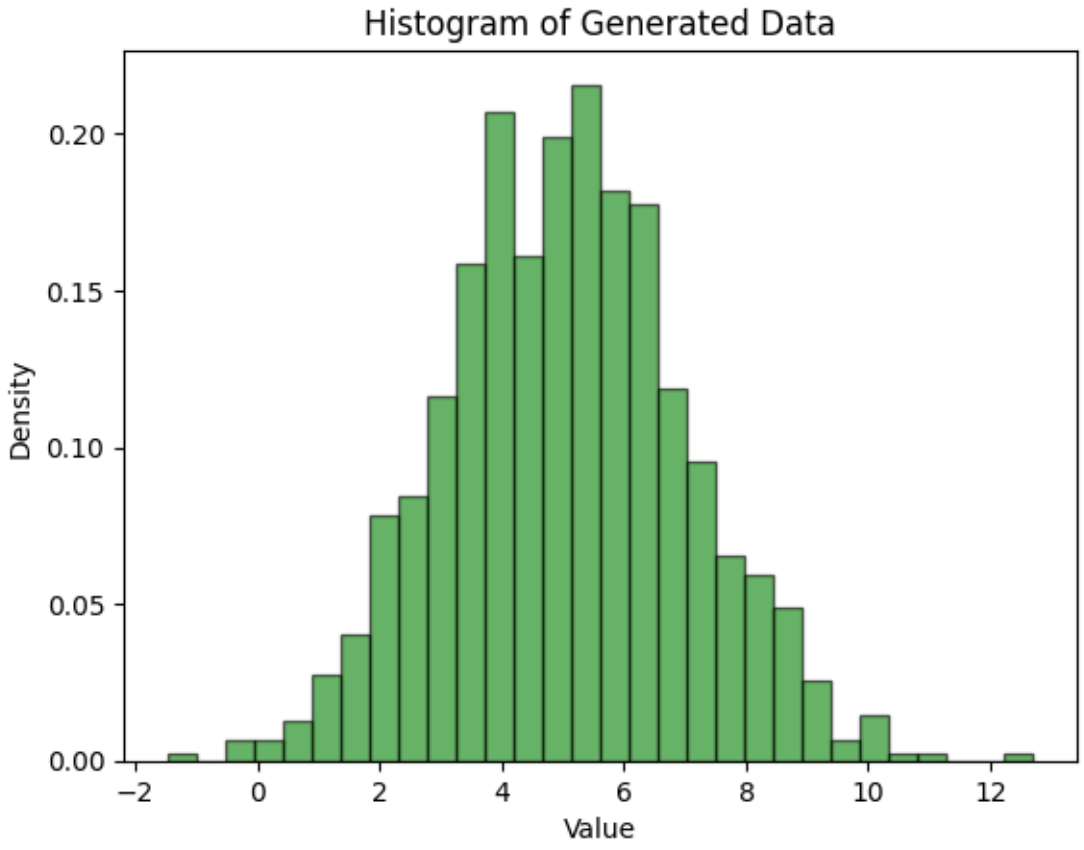
Let's use a simple example to illustrate Bayesian inference. We will use MCMC to estimate the posterior distribution of the mean of a normally distributed data set.

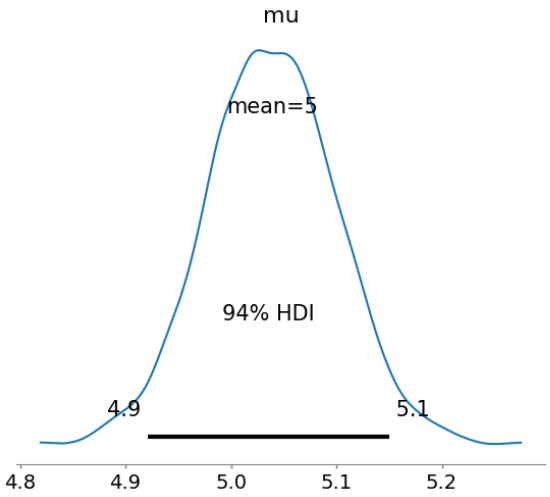
```
import pymc as pm
import logging

# Suppress pymc logging
logger = logging.getLogger('pymc')
logger.setLevel(logging.ERROR)

# Generate some data
np.random.seed(42)
data = np.random.normal(loc=5, scale=2, size=1000)

# Plot the data
plt.hist(data, bins=30, density=True, alpha=0.6, color='g', edgecolor='black')
plt.xlabel('Value')
plt.ylabel('Density')
plt.title('Histogram of Generated Data')
plt.show()

# Define the Bayesian model
with pm.Model() as model:
    # Prior for the mean (μ)
    mu = pm.Normal('mu', mu=0, sigma=10)
    # Prior for the standard deviation (σ)
    sigma = pm.HalfNormal('sigma', sigma=10)
    # Likelihood (sampling distribution) of the data
    likelihood = pm.Normal('likelihood', mu=mu, sigma=sigma, observed=data)

    # Perform MCMC sampling
    with model:
        trace = pm.sample(1000, return_inferencedata=True, progressbar=False)

# Plot the posterior distributions
pm.plot_posterior(trace)
plt.show()
```

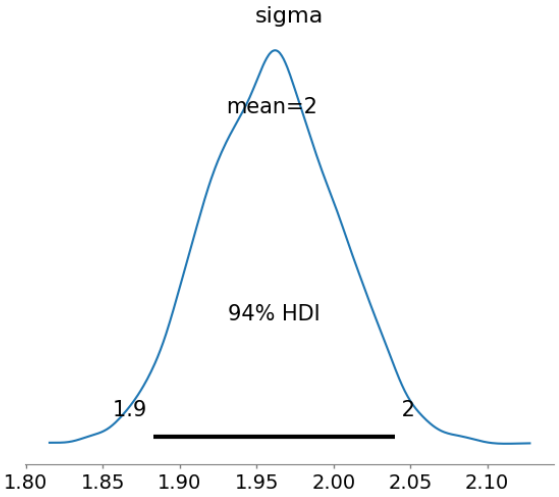
(continues on next page)

(continued from previous page)

```
# Summary of the posterior distributions
summary = pm.summary(trace)
print(summary)
```








```
        mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  \
mu     5.039  0.061   4.921     5.15      0.001    0.001    4330.0    2970.0
sigma  1.960  0.043   1.883     2.04      0.001    0.000    4506.0    3119.0
```

(continues on next page)

(continued from previous page)

```
       r_hat
mu       1.0
sigma    1.0
```

Explanation:

1. **Generate data:** we generate 100 data points from a normal distribution with a mean of 5 and a standard deviation of 2.
2. **Plot data:** a histogram is plotted to visualize the generated data.
3. **Define the Bayesian model:** ee define a Bayesian model using the PyMC library.
   - The prior distribution for the mean (μ) is set to a normal distribution with mean 0 and standard deviation 10.
   - The prior distribution for the standard deviation (σ) is set to a half-normal distribution with a standard deviation of 10.
   - The likelihood of the data is modeled as a normal distribution with mean μ and standard deviation σ.
4. **Perform MCMC sampling:** e perform MCMC sampling to estimate the posterior distributions of μ and σ.
5. **Plot posterior distributions:** the posterior distributions of the parameters are plotted.
6. **Summary of posterior distributions:** a summary of the posterior distributions is printed, showing the estimated parameters and their uncertainties.

Bayesian learning provides a powerful framework for updating beliefs about parameters in light of new data. By combining prior knowledge with observed data through Bayes' theorem, we can obtain a posterior distribution that captures our updated beliefs and uncertainties. Methods like MCMC enable us to estimate the posterior distribution even for complex models, allowing for informed decision-making based on the uncertainty in our estimates.

CHAPTER

TWO

LINEAR REGRESSION

2.1 Motivating linear approximations

Linear models are extremely useful becase they can be used to approximate complex relationships with easy and interpretable models. When employing linear models to learn a function f relating Y to X, we do not need to necessarily assume that the relationship between X and Y is linear. We are just looking for the best linear approximation to the true relationship, whatever that might be. Taylor's theorem gives reasons to believe that a linear model is a sensible approximation even for more complex functions, at least locally. Indeed, if the true regression function $f(x)$ is a smooth function, given a specific value x_0, we can expand the function as

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots \tag{2.1}$$

This expansion breaks down the function into an infinite sum of terms based on the function's derivatives at x_0. The approximation starts with the function's value at x_0, then adds adjustments based on how the function changes (its derivatives) as you move away from x_0. For x close enough to x_0, we can get away with truncating the series at first order, as in

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) \tag{2.2}$$

When you truncate the Taylor series after the first derivative term, we are essentially creating a linear model. This model approximates the function $f(x)$ using a straight line that tangentially matches the function's slope at x_0. This approach is valid as long as the higher-order terms (like the quadratic term and beyond) are negligible, which usually means x is close enough to x_0. Thus, while a linear approximation may work well locally (near x_0), extending this approximation globally (over a wide range of x values) may not always be accurate unless the function is nearly linear over that range. The key to a successful linear approximation lies in determining how "close" x must be to x_0 for the higher-order terms to be negligible. For a linear approximation to be valid, we want the influence of this term (and all higher-order terms) to be small compared to the first derivative term. We formalize this by imposing that the linear term dominates over the quadratic

$$|x - x_0| f'(x_0) \gg \frac{f''(x_0)}{2} \tag{2.3}$$

which is true if

$$2\frac{f'(x_0)}{f''(x_0)} \gg |x - x_0| \tag{2.4}$$

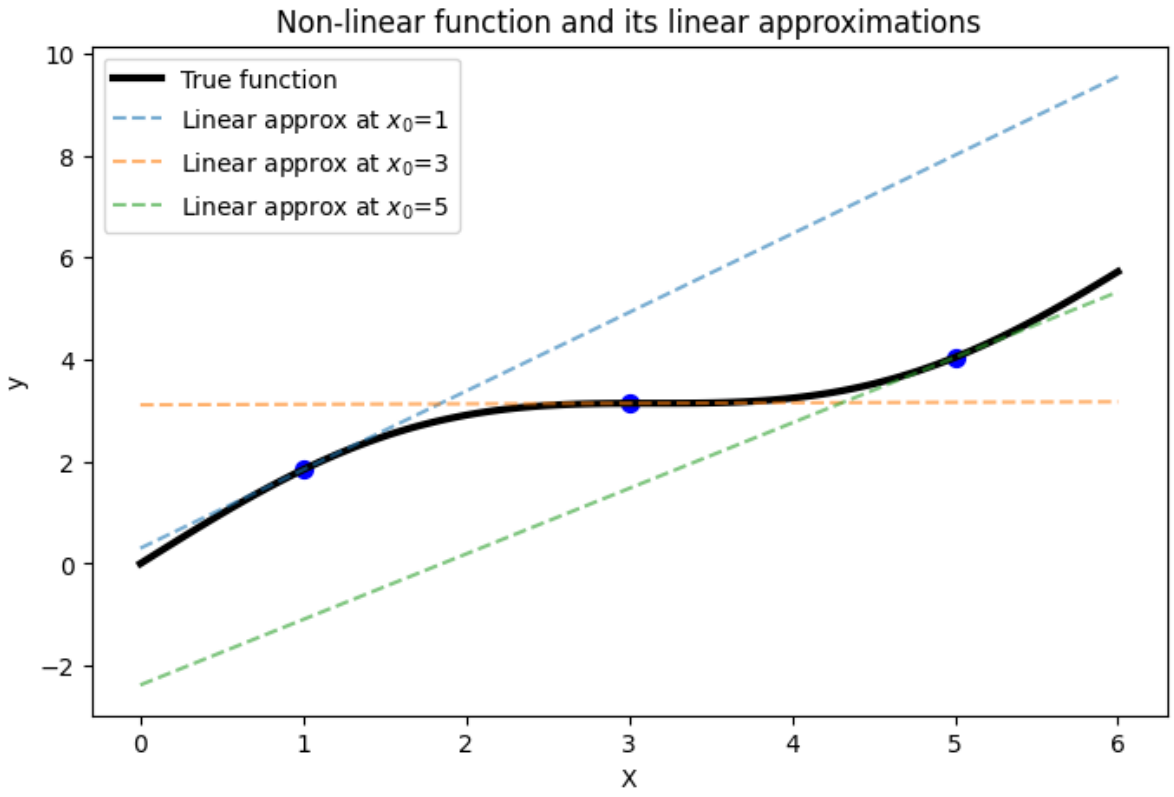
This tells us that the distance between x and x_0 must be smaller than twice the ratio of the magnitude of the first derivative to the magnitude of the second derivative for the linear approximation to hold effectively. The exact bounds of "close enough" depend on the relative sizes of the first and second derivatives of the function at x_0, providing a rule of thumb for when a linear model is likely to be a good approximation.

To show this concept, let's try to create a plot where we use several linear approximations at different points along a non-linear function. This will demonstrate how linear functions can locally approximate the non-linear function at different regions.

- First, we will define a **non-linear function** $f(x) = \sin(x) + x$, and compute its first derivative $f'(x) = \cos(x) + x$.

- Then, we select three points ($x_0 = 1, 3, 5$) at which to compute the **linear approximations**. For each point x_0:
  - We compute $y_0 = f(x_0)$ and the slope $f'(x_0)$.
  - We use the linear approximation formula $f(x) \approx y_0 + f'(x_0)(x - x_0)$.
  - We plot the linear approximation along with the original non-linear function.

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

# Define the non-linear function
def true_function(x):
    return np.sin(x) + x

# Compute the first derivative of the true function
def true_function_derivative(x):
    return np.cos(x) + 1

# Points at which we will create linear approximations
x0_points = [1, 3, 5]

# Generate data for the non-linear function
X_nonlinear = np.linspace(0, 6, 100).reshape(-1, 1)
y_nonlinear = true_function(X_nonlinear)

plt.figure(figsize=(8, 5), dpi=100)

# Plot the true non-linear function
plt.plot(X_nonlinear, y_nonlinear, color='k', label='True function', lw=3)

# Plot linear approximations at different points
for x0 in x0_points:
    y0 = true_function(x0)
    slope = true_function_derivative(x0)
    y_approx = y0 + slope * (X_nonlinear - x0)

    plt.plot(X_nonlinear, y_approx, linestyle='--', label=f'Linear approx at $x_0$=
 ↪{x0}', alpha=0.6)
    plt.scatter([x0], [y0], color='blue', s=50)

plt.xlabel('X')
plt.ylabel('y')
plt.title('Non-linear function and its linear approximations')
plt.legend()
plt.show()
```



The resulting plot shows how different linear functions can locally approximate the non-linear function reasonably well at different regions. However, we can see how the approximation is only valid in the vicinity of x_0.

2.2 Linear regression framework

A linear regression model is a model used to analyze the relationship between a dependent variable (response) and one or more independent variables (predictors or covariates). In the case of one predictor, the linear regression model is referred to as a simple linear regression model, and it is given by

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i \tag{2.5}$$

where y_i is the ith observation of the response variable, x is the ith observation of the predictor, β_0 is the intercept (the value of y when x is zero), β_1 is the regression coefficient (the expected change in y per unit change in x), and ϵ represents the error term, accounting for the variability in y that cannot be explained by the linear relationship with x. In most of the cases, we will assume ϵ to be normally distributed around zero with finite variance, $\epsilon \sim \mathcal{N}(0, \sigma^2)$. In the case of p predictors and n observations, the model is referred to as a multiple linear regression model, and we can express it in matrix notation, as in

$$\mathbf{y} = \mathbf{X}\beta + \epsilon \tag{2.6}$$

where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & & \vdots & \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \quad \text{and} \quad \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

$\mathbf{y}$ is a $n \times 1$ vector of response variables, $\mathbf{X}$ is a $n \times (p+1)$ model matrix, β is a $(p+1) \times 1$ vector of regression coefficients, and ϵ is a $n \times 1$ vector representing the noise, with covariance matrix $\sigma^2\mathbf{I}$. If the predictors and the response

are centered (for example by subtracting the mean), the intercept term can be removed from the model. In that case, the size of the model matrix becomes $n \times p$, and β a $p \times 1$ vector. We will assume this is the case in the following sections.

Linear regression models are based on several **key assumptions**:

- **Linearity**: the relationship between the predictors and the response is linear. This implies that a change in a predictor leads to a proportional change in the response. While real-life processes are rarely purely linear, we can often assume local linearity. This means that the linearity assumption holds within a reasonably limited range of the design space, recognizing that the relationship may not be linear over a wider range.
- **Independence**: observations are independent of each other. In other words, the observations do not influence each other. This assumption can be violated in many situations, particularly in time series data or spatial data where there might be autocorrelation (i.e., the value of a variable at one point in time or space is correlated with its values at other points).
- **Homoscedasticity**: the variance of the error terms (residuals) is constant across all levels of the independent variables. This condition, known as homoscedasticity, implies that the spread of the residuals should be roughly the same for all values of the predictors. If the variance of the residuals changes with the level of the predictors, the condition is known as heteroscedasticity, which can lead to inefficiencies and bias in the estimation of parameters.
- **Normality of the error terms**: this assumption is particularly important for hypothesis testing and creating confidence intervals. It's crucial to note that this assumption pertains to the errors, not necessarily to the distributions of the predictors or the response variable. While linear regression can be robust to mild violations of this assumption, severe departures can affect the reliability of inference procedures

2.3 Ordinary least squares

In regression analysis, the most common method to estimate the unknown model parameters ⍰ is ordinary least squares (OLS). The OLS method seeks to find the coefficients that minimize the sum of squares of the errors, ϵ_i. Recall that a key assumption in linear regression models is that $\{\epsilon_i\}$ are uncorrelated random variables. We aim to find the vector of least squares estimators $beta$ that minimizes

$$\mathcal{L} = \sum_{i=1}^{n} \epsilon_i^2 = \epsilon^\top \epsilon \tag{2.7}$$

Because $\mathbf{y} = \mathbf{X}\beta + \epsilon$, we can express $\epsilon = \mathbf{y} - \mathbf{X}\beta$. So, we have

$$\begin{aligned}
\mathcal{L} &= (\mathbf{y} - \mathbf{X}\beta)^\top (\mathbf{y} - \mathbf{X}\beta) && (2.8)\\
&= (\mathbf{y}^\top - \beta^\top \mathbf{X}^\top)(\mathbf{y} - \mathbf{X}\beta) && (2.9)\\
&= \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{X}\beta - \beta^\top \mathbf{X}^\top \mathbf{y} + \beta^\top \mathbf{X}^\top \mathbf{X}\beta && (2.10)\\
&= \mathbf{y}^\top \mathbf{y} - 2\beta^\top \mathbf{X}^\top \mathbf{y} + \beta^\top \mathbf{X}^\top \mathbf{X}\beta && (2.11)
\end{aligned}$$

Note that $\mathbf{y}^\top \mathbf{X}\beta$ is a scalar, because we have $(1 \times n) \times (n \times p) \times (p \times 1)$, resulting in a 1×1 matrix, which is a scalar. Similarly, $\beta^\top \mathbf{X}^\top \mathbf{y}$, having dimensions $(1 \times p) \times (p \times n) \times (n \times 1)$, also results in a 1×1 matrix, or scalar. Moreover, due to the properties of transposition and the commutative property of scalar multiplication, these two expressions are not only scalars but also represent the same scalar value. Transposing a scalar does not affect its value, thus we have $(\mathbf{y}^\top \mathbf{X}\beta)^\top = \beta^\top \mathbf{X}^\top \mathbf{y}$. Remind that, in matrix multiplication, if we transpose the product of two matrices, we reverse the order of multiplication and transpose each matrix: $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$.

Now, we need to get the derivative of $\mathcal{L}$ with respect to the parameter vector β and set it to zero. This way we will find the estimated coefficients **b**.

$$\frac{\partial \mathcal{L}}{\partial \beta}\Big|_{\hat{\beta}} = -2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X}\hat{\beta} = \mathbf{0} \tag{2.12}$$

which simplifies to

$$\mathbf{X}^\top \mathbf{X}\hat{\beta} = \mathbf{X}^\top \mathbf{y} \tag{2.13}$$

Multiplying both sides by the inverse of $\mathbf{X}^\top\mathbf{X}$ we get the OLS estimate

$$\hat{\beta} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} \tag{2.14}$$

The matrix $\mathbf{X}^\top\mathbf{X}$ is sometimes referred to as the Gram matrix or the moment matrix, because it contains the ``second moments" (i.e., variances and covariances) of the independent variables. The diagonal elements are the sums of squares of each predictor, and the off-diagonal elements represent the sums of cross-products (or covariances) between different predictors. The fitted regression model is then given by

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta} = \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} = \mathbf{H}\mathbf{y} \tag{2.15}$$

where the matrix $\mathbf{H}$ is referred to as the hat matrix or influence matrix. We can see how the fitted values at the data points used to estimate the model are linear combinations of the observed responses, with weights given by the hat matrix. Geometrically, this means that we find the fitted values by taking the vector of observed responses $\mathbf{y}$ and projecting it onto a certain plane, which is entirely defined by the values in $\mathbf{X}$. If we repeat our experiment (e.g., survey, observation) many times at the same locations $\mathbf{X}$, we get different responses $\mathbf{y}$ every time. But $\mathbf{H}$ does not change. The properties of the fitted values are thus largely determined by the properties of $\mathbf{H}$.

2.4 Expected value and variance of the least square estimators

Because we know that $\epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I}_n)$, we can say that $\mathbf{y} \sim \mathcal{N}(\mathbf{X}\beta, \sigma^2\mathbf{I}_n)$. Since ϵ follows a multivariate normal distribution, any linear combination of ϵ is also multivariate normally distributed, including $\mathbf{y}$. The expectation of $\mathbf{y}$ is $\mathbf{X}\beta$ because ϵ has expectation zero. Similarly, since $\hat{\beta}$ is a linear transformation of $\mathbf{y}$, it is also normally distributed.

Assuming the model is correct, we can first evaluate the bias of the OLS estimator by looking at the expected value of $\hat{\beta}$, which is given by

$$\begin{aligned}
\mathbb{E}[\hat{\beta}] &= \mathbb{E}[(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}] && (2.16)\\
&= \mathbb{E}[(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top(\mathbf{X}\beta + \epsilon)] && (2.17)\\
&= \mathbb{E}[(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X}\beta + (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\epsilon)] && (2.18)\\
&= \mathbb{E}[\beta + (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\epsilon] && (2.19)\\
&= \beta && (2.20)
\end{aligned}$$

Thus, $\hat{\beta}$ is an unbiased estimator of β if the model is correct. In the derivation of the expected value we used that $\mathbb{E}[\epsilon] = \mathbf{0}$ and that $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X} = \mathbf{I}$, and of course the expected value of the constant β is β itself.

The variance property of $\hat{\beta}$ is expressed by the covariance matrix

$$\text{Cov}[\hat{\beta}] = \mathbb{E}[(\hat{\beta} - \mathbb{E}[\hat{\beta}])(\hat{\beta} - \mathbb{E}[\hat{\beta}])^\top] \tag{2.21}$$

which is a $p \times p$ symmetric matrix whose jth diagonal element is the variance of $\hat{\beta}_j$ and whose (ij)th off-diagonal element is the covariance between $\hat{\beta}_i$ and $\hat{\beta}_j$. The covariance matrix of $\hat{\beta}$ is found by applying a variance operator to $\hat{\beta}$

$$\text{Cov}[\hat{\beta}] = \text{Var}[\hat{\beta}] = \text{Var}[(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}] \tag{2.22}$$

Now let us remind that, in general, if a vector $\mathbf{v}$ has covariance matrix $\mathbf{C}$ we have that

$$\text{Cov}[\mathbf{A}\mathbf{v}] = \mathbf{A}\mathbf{C}\mathbf{A}^\top \tag{2.23}$$

where $\mathbf{A}$ is a linear transformation. Thus, knowing that $\mathbf{y}$ has a covariance equal to $\sigma^2\mathbf{I}_n$, we get that

$$\begin{aligned}
\text{Var}[\hat{\beta}] &= \text{Var}[(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}] && (2.24)\\
&= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top(\sigma^2\mathbf{I}_n)((\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top)^\top && (2.25)\\
&= \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{I}_n((\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top)^\top && (2.26)\\
&= \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1} && (2.27)\\
&= \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1} && (2.28)
\end{aligned}$$

Thus, we have that our estimator is normally distributed around the true parameter vector as

$$\hat{\beta}|\mathbf{X} \sim \mathcal{N}(\beta, \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}) \tag{2.29}$$

2.4.1 First-order model example

Let's create an example to show how we can manually implement OLS to fit a multiple linear regression model. First, we generate 1000 data points with two predictors $\mathbf{x}_1$ and $\mathbf{x}_2$, and y is calculated as

$$\mathbf{y} = 3 + 4\mathbf{x}_1 + 5\mathbf{x}_2 + \text{noise}$$

```
# Generate synthetic data with two predictors
np.random.seed(0)
X1 = 2 * np.random.rand(1000, 1)
X2 = 3 * np.random.rand(1000, 1)
y = 3 + 4 * X1 + 5 * X2 + np.random.randn(1000, 1)
```

To estimate the regression coefficients, we first create the data matrix $\mathbf{X}$, including a column of ones to account for the intercept term, and the two features $\mathbf{x}_1$ and $\mathbf{x}_2$.

```
# Combine the predictors into one matrix and adding a column of ones for the intercept
X = np.c_[np.ones((1000, 1)), X1, X2]
```

Then, we can manually implement OLS and find the best-fitting parameters $\hat{\beta} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$.

```
# Manually implement the OLS method
beta = np.linalg.inv(X.T @ X) @ X.T @ y

# Print the estimated coefficients
print(f"Intercept: {beta[0][0]:.3f}")
print(f"Coefficient for X1: {beta[1][0]:.3f}")
print(f"Coefficient for X2: {beta[2][0]:.3f}")
```

```
Intercept: 3.079
Coefficient for X1: 3.963
Coefficient for X2: 4.956
```

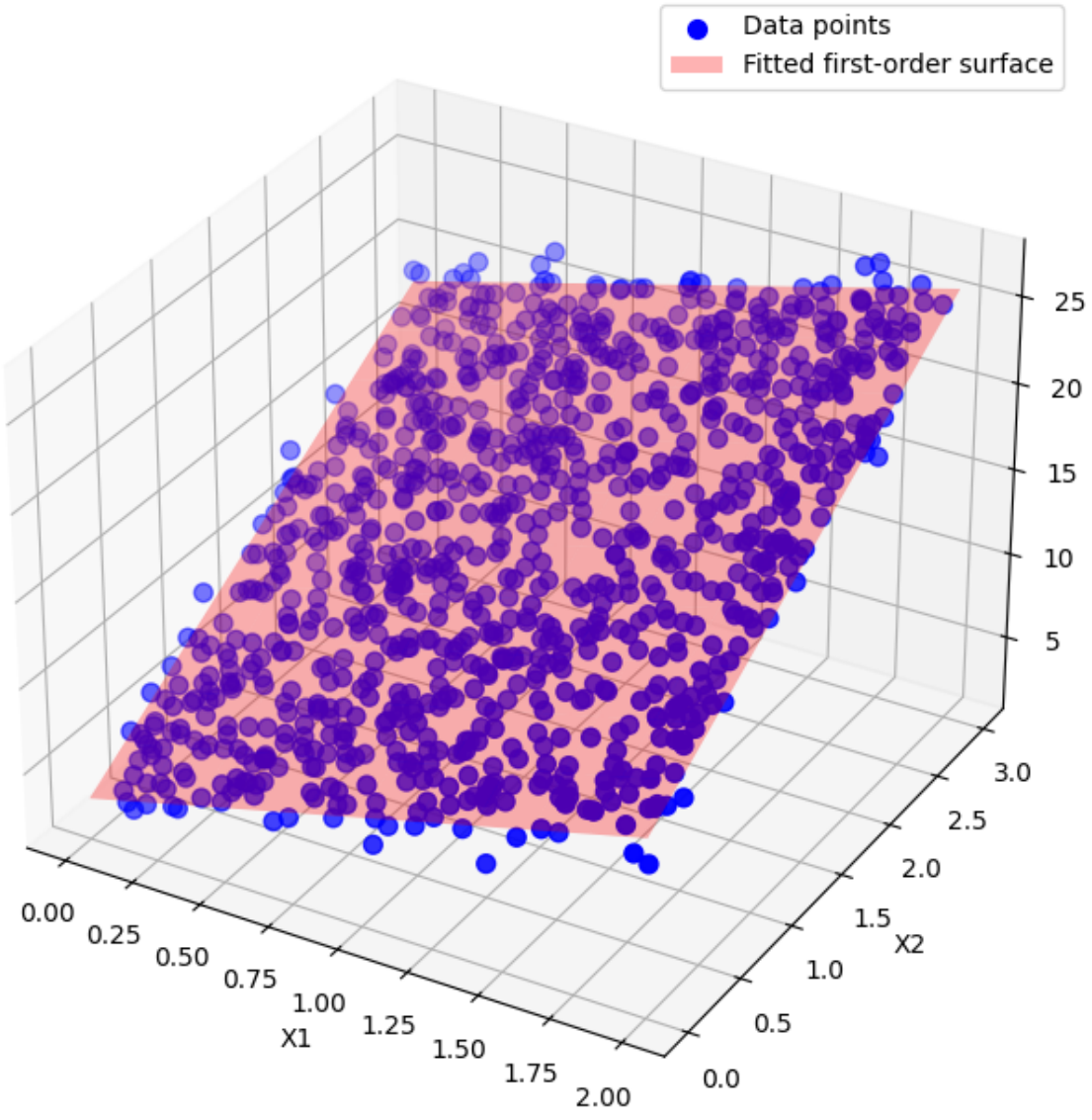
We can see how the estimated coefficients are very close to the real ones. Now, we can visualise the fitted surface by creating a grid of values for $\mathbf{x}_1$ and $\mathbf{x}_2$ and computing the predicted $\mathbf{y}$ values over this grid to generate the regression surface.

```
from mpl_toolkits.mplot3d import Axes3D

# Predictions for the surface plot
x1_surf, x2_surf = np.meshgrid(np.linspace(0, 2, 1000), np.linspace(0, 3, 1000))
y_surf = beta[0] + beta[1] * x1_surf + beta[2] * x2_surf

# Plotting the data and the regression surface
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X1, X2, y, color='blue', s=50, label='Data points')
ax.plot_surface(x1_surf, x2_surf, y_surf, color='red', alpha=0.3, rstride=100,␣
↪cstride=100, label='Fitted first-order surface')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('y')
plt.legend()
plt.show()
```



2.5 Second-order model example

A second-order model is a model that extends the linear regression model to capture non-linear relationships by including quadratic terms. The model can be written as:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \epsilon_i$$

where:

- y_i is the i-th observation of the response variable,
- x_i is the i-th observation of the predictor,
- β_0 is the intercept,
- β_1 is the linear coefficient,
- β_2 is the quadratic coefficient,
- ϵ_i represents the error term.

In matrix notation, for n observations and one predictor, the model can be expressed as:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

where:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{11}^2 \\ 1 & x_{21} & x_{21}^2 \\ \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n1}^2 \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}, \quad \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

Then, once we have augmented the matrix $\mathbf{X}$ to include quadratic terms, we can find the regression coefficients using the OLS estimate, as we did in the first-order case. It is easy to see how this is also a valid approach when we have more than one predictor and multiple quadratic terms.

Let's now generate some data from the following second-order model:

$$\mathbf{y} = 2 + 3\mathbf{x} + 5\mathbf{x}^2 + \text{noise}$$

```
# Generate synthetic data
np.random.seed(0)
X = 2 * np.random.rand(100, 1) - 1
y = 2 + 3 * X + 5 * X**2 + np.random.randn(100, 1)  # Quadratic function with noise
```

As we did before, we create the data matrix $\mathbf{X}$ by stacking a column of ones, and two more columns: one for the original feature $\mathbf{x}$, and one for its squared version $\mathbf{x}^2$.

```
# Prepare the design matrix for second-order polynomial regression
X_poly = np.c_[np.ones((100, 1)), X, X**2]
```

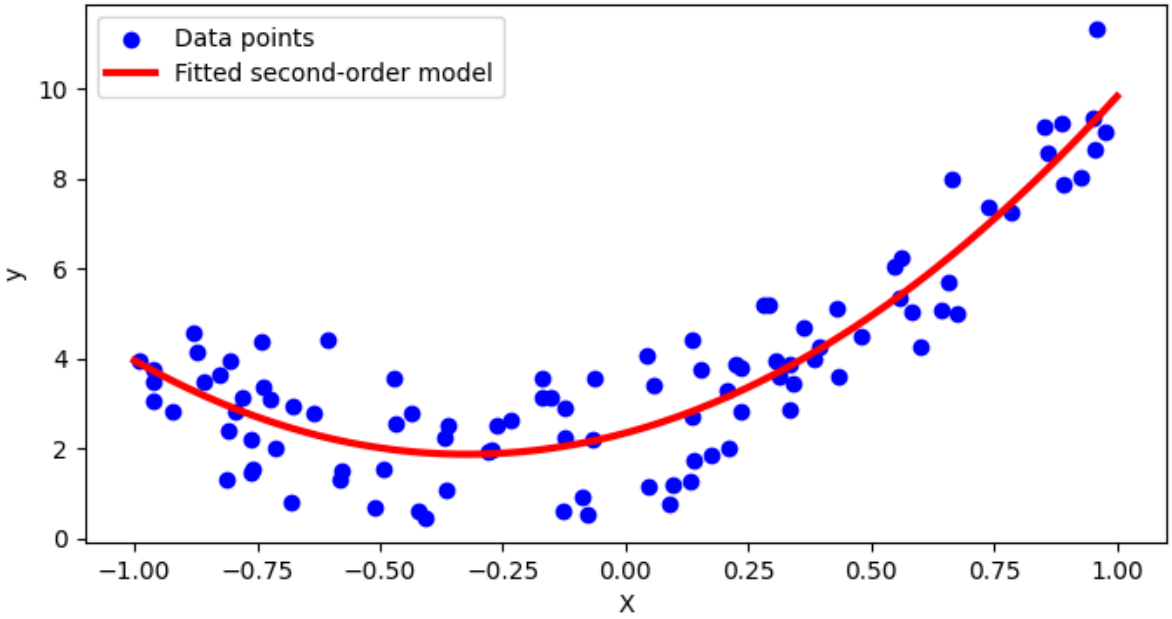
Then, we can find the estimated coefficients using the OLS formula.

```
# Manually implement the OLS method
beta = np.linalg.inv(X_poly.T @ X_poly) @ X_poly.T @ y

# Print the estimated coefficients
print(f"Intercept: {beta[0][0]:.3f}")
print(f"Coefficient for x: {beta[1][0]:.3f}")
print(f"Coefficient for x^2: {beta[2][0]:.3f}")
```

```
Intercept: 2.341
Coefficient for x: 2.937
Coefficient for x^2: 4.548
```

Finally, we cna plot the fitted curve.

```
# Predictions for plotting
X_new = np.linspace(-1, 1, 100).reshape(100, 1)
X_new_poly = np.c_[np.ones((100, 1)), X_new, X_new**2]
y_pred = X_new_poly.dot(beta)

# Plotting the data and the polynomial regression model
plt.figure(figsize=(8, 4))
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X_new, y_pred, color='red', linewidth=3, label='Fitted second-order model')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```



2.6 Hypothesis testing for regression coefficients

2.6.1 Starting with an example

In linear regression, hypothesis testing is used to determine whether the relationship between the independent variables (predictors) and the dependent variable (response) is statistically significant. The null hypothesis (H_0) typically states that there is no relationship between the predictor and the response, i.e., the coefficient for the predictor is zero. The alternative hypothesis (H_1) states that there is a significant relationship, i.e., the coefficient is not zero.

Let's generate a synthetic dataset with multiple predictors, fit a multiple linear regression model, and perform hypothesis testing on the coefficients to determine their significance. We will generate a dataset with multiple predictors and some

extra predictors that are not significant to test the hypothesis testing process.

Step-by-step procedure:

1. Generate synthetic data with multiple predictors. The true coefficients are `[5, 3, -2, 1, 0, 0]`, where some predictors have zero coefficients to simulate non-significant predictors. The response variable is generated as $y = 5 + 3X_1 - 2X_2 + X_3 +$ noise, where the noise is normally distributed with mean 0 and standard deviation 1.
2. Fit a multiple linear regression model using `statsmodels`.
3. Interpret the regression summary table, including the p-values.
4. Explain the formulas for p-value computation.

```
import pandas as pd
import statsmodels.api as sm

# Generate synthetic data
np.random.seed(0)
n_samples = 1000

# True coefficients
beta = np.array([5, 3, -2, 1, 0, 0])  # Including some zero coefficients for non-
↪significant predictors

# Generate random predictors
X1 = 2 * np.random.rand(n_samples, 1)
X2 = 3 * np.random.rand(n_samples, 1)
X3 = np.random.rand(n_samples, 1) - 1
X4 = np.random.rand(n_samples, 1) * 2 - 1
X5 = np.random.rand(n_samples, 1)
X6 = np.random.rand(n_samples, 1)

# Combine predictors into a matrix
X = np.c_[X1, X2, X3, X4, X5, X6]

# Generate response variable with some noise
y = 5 + 3*X1 + (-2)*X2 + 1*X3 + np.random.randn(n_samples, 1)

# Convert to pandas DataFrame for ease of use with statsmodels
df = pd.DataFrame(X, columns=['X1', 'X2', 'X3', 'X4', 'X5', 'X6'])
df['y'] = y

# Add a constant term (intercept) to the predictors
X = sm.add_constant(df.drop(columns='y'))

# Fit the model using statsmodels
model = sm.OLS(df['y'], X)
results = model.fit()

# Print the summary of the regression model
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.860
Model:                            OLS   Adj. R-squared:                  0.859
Method:                 Least Squares   F-statistic:                     1013.
Date:                Sun, 07 Jul 2024   Prob (F-statistic):               0.00
```

(continues on next page)

(continued from previous page)

```
Time:                        20:04:07   Log-Likelihood:                -1415.6
No. Observations:                1000   AIC:                             2845.
Df Residuals:                     993   BIC:                             2879.
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          5.0404      0.124     40.701      0.000       4.797       5.283
X1             2.9303      0.054     53.805      0.000       2.823       3.037
X2            -1.9891      0.035    -56.372      0.000      -2.058      -1.920
X3             0.9598      0.111      8.681      0.000       0.743       1.177
X4            -0.0396      0.055     -0.716      0.474      -0.148       0.069
X5            -0.1008      0.112     -0.897      0.370      -0.321       0.120
X6             0.1066      0.109      0.978      0.328      -0.107       0.321
==============================================================================
Omnibus:                        2.438   Durbin-Watson:                   2.039
Prob(Omnibus):                  0.296   Jarque-Bera (JB):                2.178
Skew:                           0.019   Prob(JB):                        0.337
Kurtosis:                       2.775   Cond. No.                         11.5
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

The `summary()` function from `statsmodels` provides detailed information about the regression model, including coefficients, standard errors, t-values, and p-values for each predictor.

How to read this table:

1. `Dep. Variable: y` This is the response variable that the model is trying to predict.
2. `R-squared: 0.860` indicates the proportion of the variance in the dependent variable that is predictable from the independent variables.
3. `Adj. R-squared: 0.859` adjusts for the number of predictors in the model.
4. `F-statistic: 1013.` tests the overall significance of the model, a very small p-value (`Prob (F-statistic): 0.00`) indicates that at least one predictor is significantly related to the response variable.
5. `Log-Likelihood: -1415.6` can be used in model comparison. Higher values (closer to zero) indicate a better fit.
6. `No. Observations: 1000` is total number of data points used in the regression.
7. `AIC: 2845.` and `BIC: 2879.` are information criteria used for model comparison. Lower values indicate a better fit.
8. `Df Residuals: 993` and `Df Model: 6` are the degrees of freedom for the residuals and the model, respectively.
9. The **coefficients table** is the main part of the output and it includes:
   - `coef` the coefficient estimates for the intercept term and for each predictor.
   - `std err` measure the variability of the coefficient estimates. Smaller values indicate more precise estimates.
   - `t` the t-values computed as the ratio of the coefficient to its standard error: $t = \frac{\hat{\beta}}{SE(\hat{\beta})}$.

- `P>|t|` the p-value which indicates the probability of observing a t-value at least as extreme as the computed one if the null hypothesis ($\beta = 0$) is true. A small p-value (typically < 0.05) suggests that the coefficient is significantly different from zero.
- `[0.025 0.975]` the confidence intervals provide a range of values within which the true coefficient is likely to fall with 95% confidence.

10. The **diagnostic tests** part include tests that check for various assumptions and properties of the residuals.
    - **Omnibus**: Tests for skewness and kurtosis. A small value indicates normality.
    - **Durbin-Watson**: Tests for autocorrelation in residuals. Values around 2 indicate no autocorrelation.
    - **Jarque-Bera**: Another test for normality. A small value indicates normality

2.6.2 Some background about hypothesis testing and confidence intervals

Test for overall significance of the model

The test for significance of regression is a test to determine if there is a linear relationship between the response variable y and a subset of the regressor variables $x_1, x_2, \dots, x_p$. The appropriate hypotheses are

$$H_0 : \beta_1 = \beta_2 = \dots = \beta_p = 0 \tag{2.30}$$

$$H_1 : \beta_j \neq 0 \quad \text{for at least one } j \tag{2.31}$$

This test procedure is called an analysis of variance (ANOVA) because it is based on a decomposition of the total variability in the response variable y. The total variability in the response variable is measured by the total sum of squares (SS_T), calculated as

$$\text{SS}_T = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{2.32}$$

which is used as a measure of overall variability in the data. Intuitively, this is reasonable because if we were to divide SS_T by the appropriate number of degrees of freedom, we would have the sample variance of the ys. This SS_T is partitioned into

$$\text{SS}_T = \text{SS}_M + \text{SS}_E \tag{2.33}$$

where SS_M is the sum of squares due to the regression model(SS_M), measuring the explained variation in y, calculated as $\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$; and SS_E is the sum of squares due to error (SS_E), measuring the unexplained variation in y, calculated as $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$. SS_M measures the explained variability in the dependent variable due to the independent variables in the regression model. Under the null hypothesis (which typically states that all regression coefficients except the intercept are zero), SS_M captures variability that is purely due to chance. To reject the null hypotheses and say that a significant part of the variability is explained by the model (thus justifying the existence of at least one coefficient), we would like SS_M to be significantly larger than SS_R. More formally, we use the test statistic is given by

$$F_0 = \frac{\text{SS}_M / p}{\text{SS}_E / (n - p - 1)} = \frac{\text{MS}_M}{\text{MS}_E} \tag{2.34}$$

Since the residuals are assumed to be normally distributed, SS_M and SS_E follow a chi-squared distribution, and their ratio follows an F-distribution. This is why the critical value for the test is given $F_{\alpha,p,n-p-1}$, where α is the confidence level. We reject the null hypothesis H_0 if $F_0 > F_{\alpha,p,n-p-1}$. \textbf{ANOVA one factor at the time:} If we have five factors (A, B, C, D, E) and we want to find which factors are significant for a given significance level α, we compare MS_i/MS_E for $i = A, \dots, E$ where the MS_E is usually the pure error that comes from replications (or the whole error?).

Tests on individual coefficients

Adding a variable to the regression model always causes the sum of squares for the regression model (SS_M) to increase and the error sum of squares to decrease. The hypotheses for testing the significance of any individual regression coefficient β_j are

$$H_0 : \beta_j = 0 \tag{2.35}$$
$$H_1 : \beta_j \neq 0 \tag{2.36}$$

If H_0 is not rejected, then this indicates that x_j can be deleted from the model. However, note that this is truly a partial or marginal test, because the regression coefficients depend on all the other regressor variables x_i, with $i \neq j$, that are in the model. Because we know that

$$\hat{\beta} \sim \mathcal{N}(\beta, \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}) \tag{2.37}$$

For the ith coefficient, we do have that

$$\hat{\beta}_i \sim \mathcal{N}(\beta_i, \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}_{ii}) \tag{2.38}$$

where $\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}_{ii}$ is the diagonal element of the covariance matrix $\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}$ corresponding to b_j. The test statistic is given by

$$t_0 = \frac{\hat{\beta}_i}{\sqrt{\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}_{ii}}} \tag{2.39}$$

The denominator $\sqrt{\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}_{ii}}$ is also called standard error of the regression coefficients $\hat{\beta}_i$, so we can also write it as

$$t_0 = \frac{\hat{\beta}_i}{\text{se}(\hat{\beta}_i)} \tag{2.40}$$

The distribution is a Student's t distribution because the coefficients itself is normally distributed, while the estimated variance has a Chi-square distribution. So the ratio between a normal and the squared root of a Chi-square follows a t distribution.

Confidence intervals on individual regression coefficients

A confidence interval, from the frequentist standpoint, is an interval estimate of a parameter β_j that, if the same data collection and analysis procedure were repeated many times, would contain the true parameter value a certain percentage (e.g., 95%) of the time β_j. This is because in frequentist statistics, parameters are considered fixed but unknown quantities. The data are random, which means the calculated confidence interval varies from sample to sample. Saying a 95% confidence interval for a parameter is $[a, b]$ means that if we were to repeat the study many times, 95% of such calculated intervals would contain the true parameter value. It does not mean there is a 95% probability that the true parameter lies within that specific interval for the observed data. Conversely, Bayesian methods treat parameters as random variables and provide a probability distribution (the posterior distribution) that quantifies uncertainty about parameter values given the observed data. The uncertainty about a parameter is directly quantified by its posterior distribution. A 95% credible interval from Bayesian analysis means there is a 95% probability that the parameter lies within this interval, given the data and the prior information. Frequentist confidence intervals do not allow for probabilistic statements about the parameter being within the interval for a given dataset. In contrast, Bayesian credible intervals provide a probability that the parameter lies within the interval, given the data and prior.

So far, we have demonstrated that $\hat{\beta} \sim \mathcal{N}(\beta, \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1})$. This also implied that the marginal distribution for any regression coefficient is $\hat{\beta}_j \sim \mathcal{N}(\beta_j, \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}_{jj})$. Then, we can say that

$$\frac{\hat{\beta}_j - \beta_j}{\sqrt{\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}_{jj}}} \sim \mathcal{N}(0, 1) \tag{2.41}$$

Since we do not know σ^2, we used its estimate $\hat{\sigma}^2$ obtained through the MS_E, which has a χ^2 distribution with $n-p$ degrees of freedom. Thus, we have

$$\frac{\hat{\beta}_j - \beta_j}{\sqrt{\hat{\sigma}^2(\mathbf{X}^\top\mathbf{X})^{-1}_{jj}}} \sim t_{n-p} \tag{2.42}$$

We can then define a $100(1-\alpha)\%$ confidence interval for the regression coefficient β_j, $j = 0, 1, \dots, p$, as

$$\hat{\beta}_j - t_{\alpha/2,n-p}\sqrt{\hat{\sigma}^2(\mathbf{X}^\top\mathbf{X})^{-1}_{jj}} \leq \beta_j \leq \hat{\beta}_j + t_{\alpha/2,n-p}\sqrt{\hat{\sigma}^2(\mathbf{X}^\top\mathbf{X})^{-1}_{jj}} \tag{2.43}$$

where $t_{\alpha/2,n-p}$ is the critical value from the t-distribution for $\alpha/2$ and $n-p$ degrees of freedom. This can also be written in terms of the standard error of the estimated coefficient b_j as

$$\hat{\beta}_j - t_{\alpha/2,n-p}\text{se}(\hat{\beta}_j) \leq \beta_j \leq \hat{\beta}_j + t_{\alpha/2,n-p}\text{se}(\hat{\beta}_j) \tag{2.44}$$

Confidence interval on the mean response and prediction variance

We can define a confidence interval on the mean response at a particular point $\mathbf{x}_0$. The fitted value at this particular point is

$$\hat{y}_0 = \mathbf{x}_0^\top\hat{\beta} \tag{2.45}$$

Because we have $\mathbf{y} \sim \mathcal{N}(\mathbf{X}\beta, \sigma^2\mathbf{I}_n)$, suggesting how $\mathbb{E}[y|\mathbf{X}] = \mathbf{X}\beta$, we also have that $\mathbb{E}[y|\mathbf{x}_0] = \mathbf{x}_0^\top\beta$. Then, we have

$$\mathbb{E}[\hat{y}_0] = \mathbb{E}[\mathbf{x}_0^\top\hat{\beta}] = \mathbf{x}_0^\top\beta \tag{2.46}$$

because $\mathbb{E}[\hat{\beta}] = \beta$. Thus, $\hat{y}_0$ is an unbiased estimator of $\mathbb{E}[y|\mathbf{x}_0]$. Since $\hat{y}_0$ is a linear combination of $\hat{\beta}$, which has a variance of $\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}$, the variance of $\hat{y}_0$ is given by

$$\text{Var}[\hat{y}_0] = \text{Var}\left[\mathbf{x}_0^\top\hat{\beta}\right] = \mathbf{x}_0^\top(\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1})\mathbf{x}_0 = \sigma^2\mathbf{x}_0^\top(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{x}_0 \tag{2.47}$$

Then, we can define the 100(1-α)% confidence interval on the mean response at point $\mathbf{x}_0$ as

$$\hat{y}_0 - t_{\alpha/2,n-p}\sqrt{\sigma^2\mathbf{x}_0^\top(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{x}_0} \leq \mathbb{E}[y|\mathbf{x}_0] \leq \hat{y}_0 + t_{\alpha/2,n-p}\sqrt{\sigma^2\mathbf{x}_0^\top(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{x}_0} \tag{2.48}$$

In general, we can define the **prediction variance (PV)** of the fitted value at location $\mathbf{x}_0$ as

$$\text{PV} = \text{Var}[\hat{y}_0] = \sigma^2\mathbf{x}_0^\top(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{x}_0 \tag{2.49}$$

Then, the **unscaled prediction variance (UPV)** is given by

$$\text{UPV} = \frac{\text{Var}[\hat{y}_0]}{\sigma^2} = \mathbf{x}_0^\top(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{x}_0 \tag{2.50}$$

The main benefit of the UPV is that it can be used during the design selection phase. The division by σ^2 means that \underline{UPV is a function of only the design matrix} and does not require data to have been collected. UPV provides a way to quantify the precision of the predictions made by a regression model at specific points. It tells us how much the predicted value at a certain point $\mathbf{x}_0$ is expected to vary, due to the variability in the estimated regression coefficients, but without considering the inherent variability of the response itself. The key intuition is that since $\hat{\beta}$ carries variance, any prediction made using these estimates also inherits variance (i.e., uncertainty propagation?). UPV connects parameter estimation to prediction variance by translating the uncertainty in the model's coefficients into the uncertainty of the model's predictions. If we have already collected data, we can try to estimate σ^2 using the MS_E. Then, we can define the **estimated prediction variance (EPV)** as

$$\text{EPV} = \hat{\sigma}^2\mathbf{x}_0^\top(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{x}_0 \tag{2.51}$$

2.7 Regularization methods

Regularization techniques are fundamental to the field of machine learning and statistics, primarily used to prevent overfitting, improve model generalization, and handle multicollinearity among predictors. We will now briefly introduce three primary methods for regularization.

2.7.1 Lasso Regression (L1 Penalty)

The least absolute shrinkage and selection operator (lasso) regression introduces a penalty term equal to the absolute value of the magnitude of coefficients to the loss function. The objective function of Lasso regression is given by:

$$\text{minimize} \left\{ \frac{1}{2n} \sum_{i=1}^{n} (y_i - x_i^\top \beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}, \tag{2.52}$$

where λ is a regularization parameter that controls the strength of the penalty.

Key aspects:

- Promotes sparsity in the model coefficients, effectively performing feature selection.
- Particularly useful when dealing with high-dimensional data or when feature selection is desired.
- Can result in models that are easier to interpret due to fewer predictors.

2.7.2 Ridge Regression (L2 Penalty)

Ridge regression adds a penalty equal to the square of the magnitude of coefficients to the loss function. The formulation of Ridge regression is as follows:

$$\text{minimize} \left\{ \frac{1}{2n} \sum_{i=1}^{n} (y_i - x_i^\top \beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}, \tag{2.53}$$

where λ is the regularization parameter.

Key aspects:

- Reduces model complexity by shrinking coefficients, but does not necessarily reduce them to zero.
- Particularly beneficial in situations with multicollinearity among predictors.
- Helps to stabilize the coefficient estimates and improve model generalization.

2.7.3 Elastic Net (combination of L1 and L2 penalties)

Elastic Net combines the penalties of Lasso and Ridge, incorporating both the L1 and L2 penalty terms. The objective function for Elastic Net is:

$$\text{minimize} \left\{ \frac{1}{2n} \sum_{i=1}^{n} (y_i - x_i^\top \beta)^2 + \lambda_1 \sum_{j=1}^{p} |\beta_j| + \frac{\lambda_2}{2} \sum_{j=1}^{p} \beta_j^2 \right\}, \tag{2.54}$$

where λ_1 and λ_2 are parameters that control the strength of the L1 and L2 penalties, respectively.

Key aspects:

- Combines the feature selection properties of Lasso with the ridge regression's ability to handle multicollinearity.

- Encourages a grouping effect, where correlated predictors either enter or leave the model together.
- Offers a balance between Lasso and Ridge regression by allowing for control over both penalties, making it versatile for various scenarios.
- Useful in high-dimensional data scenarios where Lasso might suffer due to high correlations among predictors.

Each of these regularization methods has its unique strengths and applications, and the choice among them should be guided by the specific characteristics of the data and the modeling objectives at hand.

CHAPTER

THREE

MACHINE LEARNING

This chapter provides a very quick overview of the main concepts in machine learning, mostly to refresh some of the concepts or models that will be used throughout this book.

3.1 Supervised learning vs. unsupervised learning

In supervised learning, the model is trained on a labeled dataset, meaning that each training example is composed by input features (**X**) and labels (**y**). The goal is to learn the function relating the labels to the input features. Common tasks in supervised learning include:

- **Regression**: predicting a continuous value. Example: Predicting house prices based on features like size, location, and age.
- **Classification**: predicting a discrete label. Example: Identifying whether an email is spam or not based on its content.

In unsupervised learning, the model is usually trained on unlabeled examples (only the features **X**). The goal can be to infer the latent structure present within a set of data points. Common tasks in unsupervised learning include:

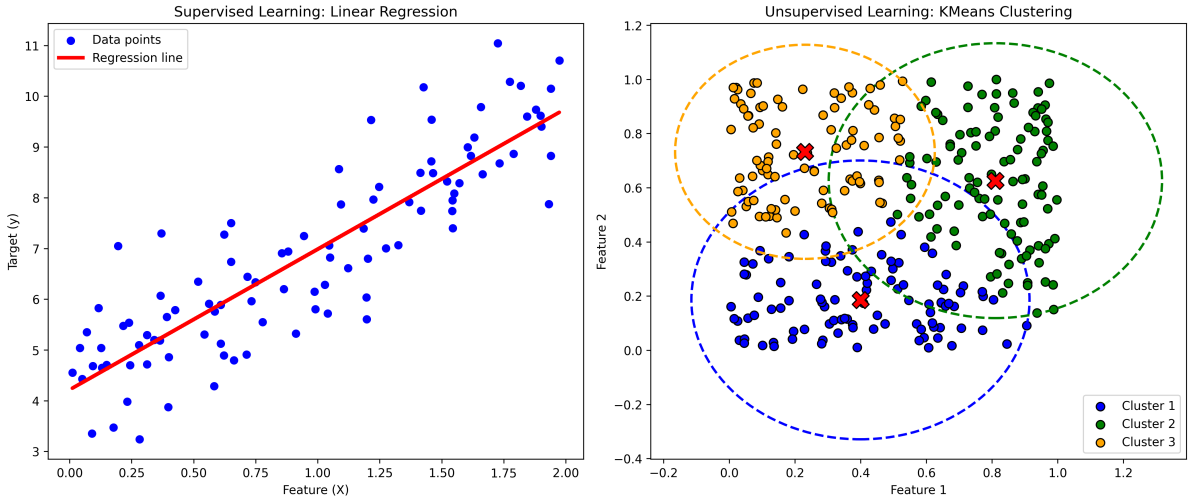
- **Clustering**: grouping similar data points together. Example: Customer segmentation based on purchasing behavior.
- **Dimensionality Reduction**: reducing the number of features while retaining the most important information. Example: Principal Component Analysis (PCA) for visualizing high-dimensional data.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.cluster import KMeans

# Generate synthetic data for regression (supervised learning)
np.random.seed(42)
X_reg = 2 * np.random.rand(100, 1)
y_reg = 4 + 3 * X_reg + np.random.randn(100, 1)

# Train a linear regression model
lin_reg = LinearRegression()
lin_reg.fit(X_reg, y_reg)
y_pred = lin_reg.predict(X_reg)

# Generate synthetic data for clustering (unsupervised learning)
np.random.seed(42)
X_cluster = np.random.rand(300, 2)
```

(continues on next page)

(continued from previous page)

```
# Define true cluster centers
true_centers = np.array([[0.2, 0.8], [0.8, 0.2], [0.5, 0.5]])
# Assign data points to clusters
y_cluster = np.argmin(np.linalg.norm(X_cluster[:, np.newaxis] - true_centers, axis=2),
 ↪ axis=1)

# Train a KMeans clustering model
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X_cluster)
y_kmeans = kmeans.predict(X_cluster)

# Plot regression results
plt.figure(figsize=(14, 6), dpi=300)

plt.subplot(1, 2, 1)
plt.scatter(X_reg, y_reg, color='blue', label='Data points')
plt.plot(X_reg, y_pred, color='red', label='Regression line', lw=3)
plt.title('Supervised Learning: Linear Regression')
plt.xlabel('Feature (X)')
plt.ylabel('Target (y)')
plt.legend()

# Plot clustering results
plt.subplot(1, 2, 2)
colors = ['blue', 'green', 'orange']
for i in range(3):
    plt.scatter(X_cluster[y_kmeans == i, 0], X_cluster[y_kmeans == i, 1], s=50,␣
 ↪c=colors[i], label=f'Cluster {i+1}', edgecolor='k')
    plt.scatter(kmeans.cluster_centers_[i, 0], kmeans.cluster_centers_[i, 1], s=200,␣
 ↪c='red', marker='X', edgecolor='k')

    # Draw circle around each cluster
    cluster_center = kmeans.cluster_centers_[i]
    cluster_points = X_cluster[y_kmeans == i]
    radius = np.max(np.linalg.norm(cluster_points - cluster_center, axis=1))
    circle = plt.Circle(cluster_center, radius, color=colors[i], fill=False,␣
 ↪linestyle='--', linewidth=2)
    plt.gca().add_patch(circle)

plt.title('Unsupervised Learning: KMeans Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.tight_layout()
plt.show()
```



3.2 Overfitting and underfitting

Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise and random fluctuations. This excessive learning leads the model to perform exceptionally well on the training data, but poorly on unseen or new data, such as validation or test datasets. Overfitting is more likely to happen when the model is overly complex relative to the amount of training data it has been given. Complexity can come from having too many features, using high-degree polynomial models, or incorporating too many parameters. The characteristics of overfitting are:

- **High training accuracy, low test accuracy**: the model fits the training data very well but generalizes poorly to new data.
- **Complex models**: models with a large number of parameters or high-degree polynomials are prone to overfitting.
- **Noise fitting**: the model captures noise and random variations in the training data as if they were true patterns.

Mitigation strategies might include:

- **Regularization**: techniques like Lasso (L1 penalty), Ridge (L2 penalty), and Elastic Net add a penalty for larger coefficients, thus discouraging overly complex models.
- **Pruning**: in decision trees, pruning helps by cutting off branches that have little importance.
- **Cross-validation**: using techniques like k-fold cross-validation ensures that the model performs well on different subsets of the data.
- **Simpler models**: choosing simpler models with fewer parameters can reduce the risk of overfitting.
- **More data**: increasing the size of the training data can help the model generalize better.

3.2.1 Underfitting

Underfitting occurs when a model is too simple to capture the underlying patterns and structure of the data. This simplicity can arise from using too few features, having a low-degree polynomial, or having an overly simplistic model structure. An underfitted model will perform poorly on both the training data and new data because it fails to capture the essential trends and relationships within the data. The characteristics of underfitting are:

- **Low training accuracy, low test accuracy**: the model does not perform well even on the training data, indicating it has not captured the underlying patterns.
- **Simple models**: models with too few parameters, low-degree polynomials, or insufficient features.
- **Bias**: the model has high bias and cannot adequately represent the complexity of the data.

Mitigation strategies might include:

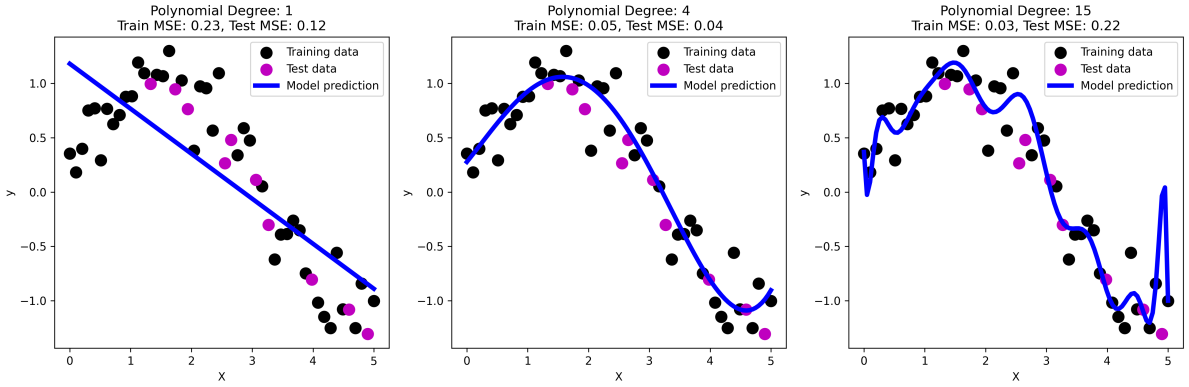
- **Complexity**: increase the complexity of the model by adding more features, using higher-degree polynomials, or more complex algorithms.
- **Feature engineering**: create new features that better capture the underlying patterns in the data.
- **Model tuning**: adjust hyperparameters to better fit the data.
- **Increase training**: ensure the model is adequately trained by allowing more iterations or using more sophisticated training techniques.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error

# Generate synthetic data
np.random.seed(0)
X = np.linspace(0, 5, num=50).reshape(-1, 1)
y = np.sin(X).ravel() + np.random.randn(50) * 0.2

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=42)

# Function to plot model
def plot_models(degrees, X_train, y_train, X_test, y_test):
    plt.figure(figsize=(15, 5), dpi=300)

    for i, degree in enumerate(degrees, start=1):
        polynomial_features = PolynomialFeatures(degree=degree)
        X_train_poly = polynomial_features.fit_transform(X_train)
        X_test_poly = polynomial_features.fit_transform(X_test)

        model = LinearRegression()
        model.fit(X_train_poly, y_train)
        y_train_pred = model.predict(X_train_poly)
        y_test_pred = model.predict(X_test_poly)

        # Plotting the fitted line using the whole range of X for a smoother curve
        X_plot = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
        X_plot_poly = polynomial_features.transform(X_plot)
        y_plot_pred = model.predict(X_plot_poly)

        plt.subplot(1, len(degrees), i)
        plt.scatter(X_train, y_train, label='Training data', c='k', s=100)
```

(continues on next page)

(continued from previous page)

```
        plt.scatter(X_test, y_test, label='Test data', c='m', s=100)
        plt.plot(X_plot, y_plot_pred, color='b', label='Model prediction', lw=4)
        plt.title(f'Polynomial Degree: {degree}\nTrain MSE: {mean_squared_error(y_
↪train, y_train_pred):.2f}, Test MSE: {mean_squared_error(y_test, y_test_pred):.2f}')
        plt.xlabel('X')
        plt.ylabel('y')
        plt.legend()

    plt.tight_layout()
    plt.show()

# Plotting models
degrees = [1, 4, 15]
plot_models(degrees, X_train, y_train, X_test, y_test)
```



The three plots above illustrate the concepts of underfitting, good fit, and overfitting using polynomial regression models of varying degrees.

1. **Underfitting**: the first plot shows a linear regression model (polynomial degree 1). The model is too simple to capture the underlying nonlinear relationship in the data, resulting in high bias. This is evident from the high training and test mean squared error (MSE). The model fails to capture the curve in the data, leading to underfitting.
2. **Good fit**: the second plot represents a polynomial regression model with degree 4. This model captures the underlying pattern of the data well, striking a balance between bias and variance. The training and test MSEs are relatively low, indicating that the model generalizes well to new data. This is an example of an optimal fit where the model complexity is appropriate for the data.
3. **Overfitting**: the third plot shows a polynomial regression model with degree 15. The model is overly complex, capturing not only the underlying pattern but also the noise in the training data. This leads to low training MSE but significantly higher test MSE, as the model does not generalize well to new data. This high variance results in poor performance on the test set, exemplifying overfitting.

These plots clearly demonstrate how model complexity affects the ability to generalize, emphasizing the importance of finding a balance to avoid both underfitting and overfitting.

3.3 Tree-based methods

So far, we mostly covered the use of linear models. Let's now briefly introduce another powerful class of models, that will be used in the book. Tree-based methods are powerful tools for both regression and classification tasks. They involve segmenting the predictor space into a number of simple regions, and making predictions based on these regions. Decision trees are a type of tree-based method that can handle complex data structures and relationships.

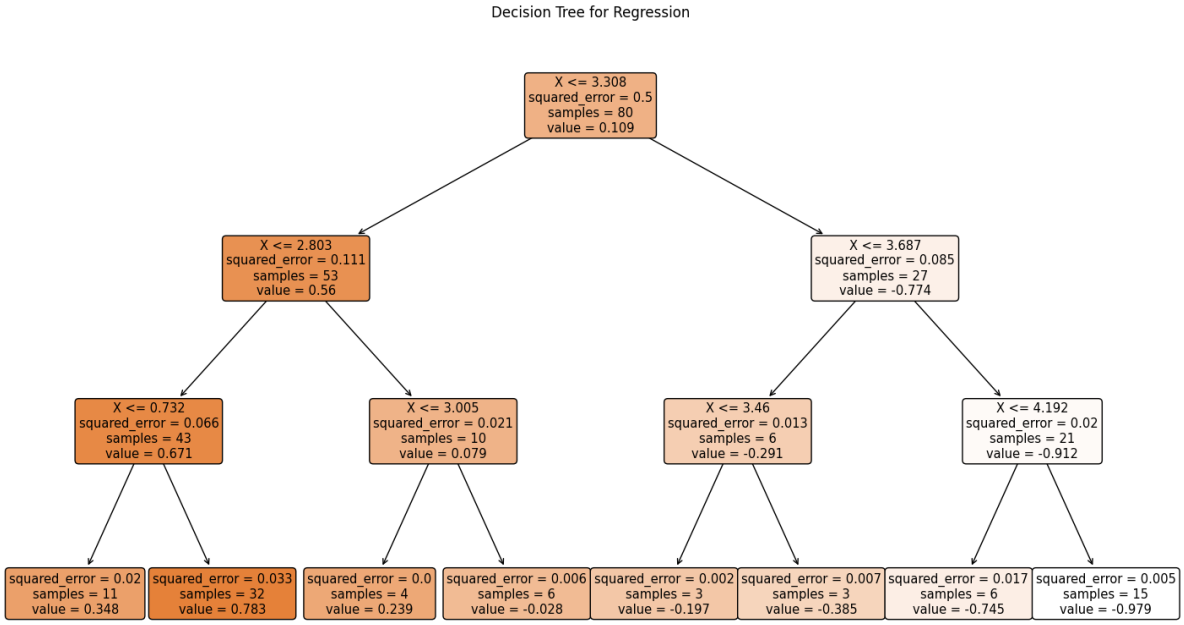
3.3.1 Decision trees

The decision tree is the most simple tree-based model. It is a non-parametric model that splits the data into subsets based on the value of the features, creating a tree-like structure. A decision tree is composed of nodes and branches. Each internal node represents a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a predicted value. The tree grows by splitting nodes into child nodes, based on the values of input features. The goal is to reduce the impurity at each step. The most common algorithm for regression trees is CART (Classification and Regression Trees). The quality of a split is measured using metrics like mean squared error (MSE) or mean absolute error (MAE). The key advantages of decision trees include:

- **Interpretability**: decision trees are easy to interpret and visualize.
- **Non-linearity**: they can capture non-linear relationships between features and the target variable.
- **Little data preprocessing**: they can handle both numerical and categorical data.
- **Feature importance**: trees provide insights into feature importance based on how often and effectively features are used to split the data.

Let's build and visualise a simple decision tree for a regression task using synthetic data.

```
from sklearn.tree import DecisionTreeRegressor, plot_tree

# Generate synthetic data
np.random.seed(42)
X = np.linspace(0, 5, 100).reshape(-1, 1)
y = np.sin(X).ravel() + np.random.randn(100) * 0.1

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=42)

# Fit a decision tree regressor
tree_reg = DecisionTreeRegressor(max_depth=3, random_state=42)
tree_reg.fit(X_train, y_train)

# Visualize the decision tree
plt.figure(figsize=(18, 10))
plot_tree(tree_reg, filled=True, feature_names=["X"], rounded=True)
plt.title('Decision Tree for Regression')
plt.show()
```



The decision tree plot shows how the data is split at each node based on the input feature X. Each internal node represents a decision based on the value of X, and each leaf node shows the predicted value of y. The tree provides a clear and interpretable way to understand how the predictions are made based on the input features.

3.3.2 Random Forests

Random Forests are an ensemble learning method that combines multiple decision trees to improve the predictive performance and control overfitting. The key idea is to build a "forest" of decision trees, where each tree is trained on a random subset of the data and a random subset of features. The key concepts behind random forests are:

1. **Bootstrap aggregating (bagging)**: random forests use bagging, where multiple subsets of the data are sampled with replacement to train each tree independently. Mathematically, if the original dataset has n samples, each tree is trained on a bootstrap sample of n samples, drawn with replacement.

$$\text{Bootstrap Sample} = \{x_i\}_{i=1}^{n} \quad \text{where} \quad x_i \sim \mathcal{D}$$

2. **Feature randomness**: at each split in the decision trees, a random subset of features is considered, introducing further randomness and reducing correlation between the trees. If there are p features, a typical number of features considered at each split is $m = \sqrt{p}$ for classification or $m = \frac{p}{3}$ for regression.

$$\text{Random Subset of Features} = \{X_j\}_{j=1}^{m} \quad \text{where} \quad m \leq p$$

3. **Voting mechanism**: for regression tasks, the final prediction is the average of the predictions from all individual trees, while for classification tasks, the majority vote is taken.

$$\text{Regression Prediction} = \frac{1}{T} \sum_{t=1}^{T} \hat{y}_t(x)$$

$$\text{Classification Prediction} = \text{mode}\left(\{\hat{y}_t(x)\}_{t=1}^{T}\right)$$

The ensemble approach employed by random forests reduces the variance and the risk of overfitting compared to individual decision trees. Indeed, aggregating multiple trees often leads to better predictive performance. Moreover, random forests provide insights into feature importance by evaluating the average decrease in impurity over all trees.

For classification, a common impurity measures are Gini impurity and entropy, which is defined for a node t as:

$$G(t) = 1 - \sum_{i=1}^{C} p_i^2$$

where p_i is the probability of class i at node t, and C is the number of classes.

For regression, the impurity measure is typically the **variance**, which is defined for a node t as:

$$\sigma^2(t) = \frac{1}{N_t} \sum_{i=1}^{N_t} (y_i - \bar{y})^2$$

where N_t is the number of samples in node t, y_i is the target value for the i-th sample, and $\bar{y}$ is the mean target value in node t.

Feature importance in random forests is calculated based on the mean decrease in impurity. For a feature X_j:

$$\text{Importance}(X_j) = \frac{1}{T} \sum_{t=1}^{T} \sum_{k \in \text{nodes}} \Delta I_k(X_j)$$

where $\Delta I_k(X_j)$ is the decrease in impurity at node k due to split on feature X_j, and T is the total number of trees in the forest.

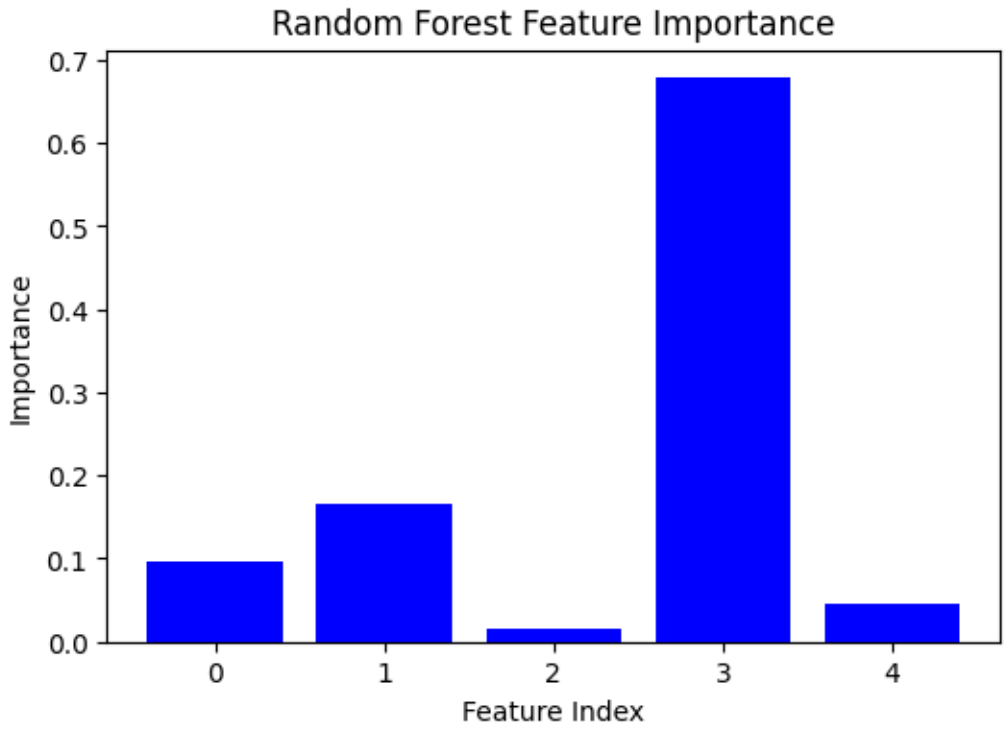
The bar plot below shows the importance scores of each feature used in a random forest model fitted to synthetic data. Each bar represents a feature, and its height indicates the relative importance of that feature in predicting the target variable. Higher bars correspond to features that are more important for the model. These features contribute more significantly to reducing the impurity of the splits in the decision trees. Understanding feature importance can provide insights into the underlying data structure and guide feature selection or engineering efforts.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor

# Generate synthetic data
np.random.seed(42)
X = np.random.rand(100, 5)
y = X @ [1.5, -2, 0, 3.5, -1] + np.random.randn(100) * 0.1

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=42)

# Fit a random forest regressor
forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(X_train, y_train)

# Extract and plot feature importance
importances = forest_reg.feature_importances_
features = np.arange(X.shape[1])
plt.figure(figsize=(6, 4))
plt.bar(features, importances, color='b', align='center')
plt.xlabel('Feature Index')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Importance')
plt.title('Random Forest Feature Importance')
plt.show()
```



3.3.3 XGBoost

XGBoost (Extreme Gradient Boosting) is an advanced implementation of gradient boosting, designed for speed and performance. It builds decision trees sequentially, where each tree attempts to correct the errors of the previous trees. The key concepts of XGBoost are:

1. **Gradient boosting**: the method sequentially builds trees by fitting the residual errors of the previous trees. Each tree tries to minimize a loss function by using gradient descent. The objective is to minimize the following loss function:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^{t} \Omega(f_k)$$

   where l is a differentiable loss function (e.g., mean squared error for regression, log loss for classification), $\hat{y}_i^{(t)}$ is the prediction at the t-th iteration, and Ω is a regularization term for the complexity of the trees f_k.

2. **Regularization**: XGBoost incorporates regularization to prevent overfitting, making it more robust than traditional gradient boosting methods. The regularization term Ω is given by:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

   where T is the number of leaves in the tree, w_j are the leaf weights, γ is a regularization parameter for the number of leaves, and λ is a regularization parameter for the leaf weights.

3. **Parallel processing**: XGBoost supports parallel processing and efficient handling of missing values, which enhances its performance. This is achieved by using a block structure for data storage and computation, allowing multiple operations to be carried out simultaneously.

XGBoost is known for its high predictive accuracy and efficiency. It can handle a variety of data types and is widely used in many industrial contexts. Moreover, it has built-in regularization parameters to control overfitting and improve generalization. Indeed, the objective function in XGBoost combines the loss function and the regularization term. For the t-th iteration, the objective function can be approximated using a second-order Taylor expansion:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

where $g_i = \frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$ and $h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)2}}$ are the first and second derivatives of the loss function, respectively.

The structure score of a tree is given by:

$$\mathcal{L}_{split} = -\frac{1}{2} \left(\frac{\sum_{i \in I_L} g_i}{\sum_{i \in I_L} h_i + \lambda} + \frac{\sum_{i \in I_R} g_i}{\sum_{i \in I_R} h_i + \lambda} - \frac{\sum_{i \in I} g_i}{\sum_{i \in I} h_i + \lambda} \right) + \gamma$$

where I_L and I_R are the instances in the left and right child nodes, respectively, and I is the set of all instances in the node before the split.

In the plot below, we show the training process, where the lines display the root mean squared error (RMSE) on both the training and test sets across the boosting rounds. The x-axis represents the number of boosting rounds (trees added), and the y-axis represents the RMSE.

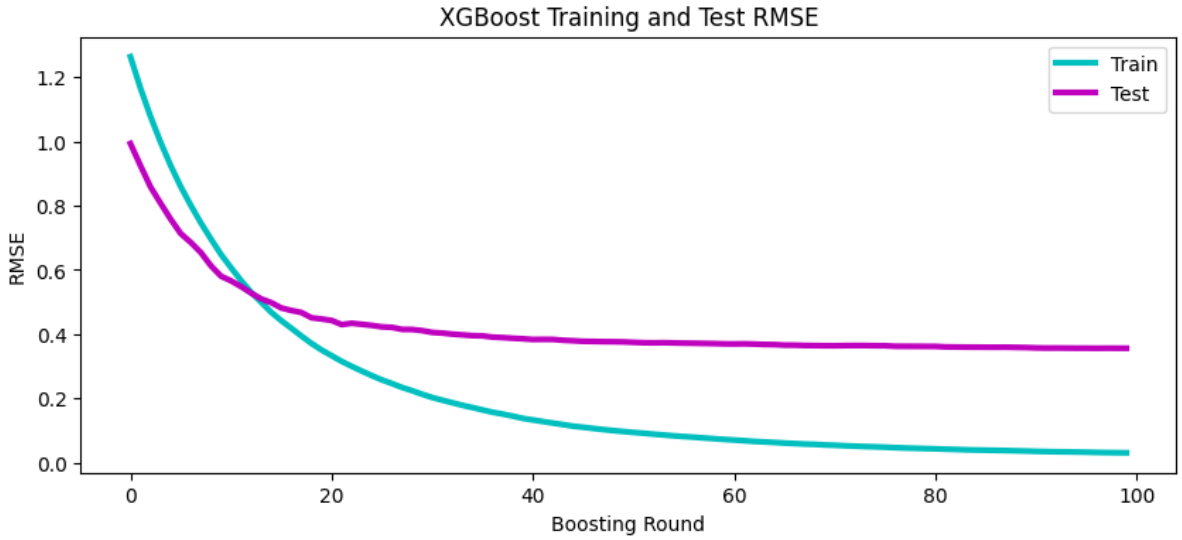
The plot helps in understanding how the model's performance evolves during training. Ideally, both training and test RMSE should decrease initially, indicating improved model performance. If the training RMSE continues to decrease while the test RMSE starts to increase, it suggests overfitting. The evaluation plot is a crucial diagnostic tool for tuning the parameters (e.g., learning rate, max depth) to achieve a balance between underfitting and overfitting.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import xgboost as xgb

# Generate synthetic data
np.random.seed(42)
X = np.random.rand(100, 5)
y = X @ [1.5, -2, 0, 3.5, -1] + np.random.randn(100) * 0.1

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=42)

# Convert the data into DMatrix format for XGBoost
train_dmatrix = xgb.DMatrix(X_train, label=y_train)
test_dmatrix = xgb.DMatrix(X_test, label=y_test)

# Define the model parameters
params = {
    "objective": "reg:squarederror",
    "max_depth": 3,
    "eta": 0.1,
    "eval_metric": "rmse"
}
```

(continues on next page)

(continued from previous page)

```
# Train the XGBoost model with evaluation sets
evals = [(train_dmatrix, 'train'), (test_dmatrix, 'test')]
num_rounds = 100
evals_result = {}
xg_reg = xgb.train(params, train_dmatrix, num_boost_round=num_rounds, evals=evals,␣
↪evals_result=evals_result, verbose_eval=False)

# Plot the evaluation results
epochs = len(evals_result['train']['rmse'])
x_axis = range(0, epochs)
plt.figure(figsize=(10, 4))
plt.plot(x_axis, evals_result['train']['rmse'], label='Train', lw=3, c='c')
plt.plot(x_axis, evals_result['test']['rmse'], label='Test', lw=3, c='m')
plt.xlabel('Boosting Round')
plt.ylabel('RMSE')
plt.title('XGBoost Training and Test RMSE')
plt.legend()
plt.show()
```



Part II

Introduction

CHAPTER

FOUR

MOTIVATION

Understanding causality is crucial in many fields, including economics, medicine, social sciences, and engineering. In the context of electricity markets, grasping causal relationships allows us to make informed decisions, optimise operations, and predict future outcomes with greater accuracy. By studying causality, we move beyond mere correlations, gaining the ability to make strategic decisions, design effective policies, and foster a deeper understanding of the intricacies of electricity markets. The potential advantages of grasping and utilising causal relationships in electricity markets include:

➔ **Enhancing understanding of complex systems**: electricity markets involve interactions between multiple entities, including generators, consumers, and regulators. Causal inference provides a structured approach to unravel these complexities, helping stakeholders understand how different factors interact and influence market outcomes.

➔ **Making better decisions**: causal inference helps differentiate between correlation and causation, which is essential for making informed and effective policy decisions. Without a proper understanding of causality, decisions can be influenced by biases such as omitted variable bias, where important factors that affect the outcome are not accounted for. By employing causal inference methods, policymakers can identify and correct for these biases, ensuring that their decisions are based on accurate and reliable evidence. This approach leads to more effective interventions and policies, avoiding the pitfalls of misleading correlations and enhancing the overall decision-making process.

➔ **Learning from interventions**: in experimental settings, such as randomised controlled trials, causality helps determine the effect of specific interventions. In electricity markets, experiments might involve testing new pricing schemes or demand response programmes, where understanding the causal impact is vital for scaling successful initiatives.

➔ **Predicting future trends**: causal models can be beneifcial for long-term forecasting because they help identify and understand the fundamental drivers of market behavior. While traditional forecasting models rely on historical data and correlations, these approaches can fall short when market conditions change or when predicting the impact of new interventions. For example, causal inference might support:

- **Understanding root causes**: by identifying the true causal relationships between variables, stakeholders can understand the root causes of market trends rather than just observing surface-level correlations. This allows for more accurate predictions, especially in changing environments.
- **Policy impact analysis**: causal models enable the evaluation of how policy changes will affect the market over time. For example, understanding how subsidies for renewable energy causally impact electricity prices can help predict future market conditions as these policies evolve.
- **Adaptation to new conditions**: markets are dynamic, and conditions often change due to technological advancements, regulatory shifts, or economic developments. Causal models help predict how these changes will influence long-term trends by focusing on the underlying mechanisms rather than past patterns alone.
- **Scenario planning**: causal inference allows for robust scenario analysis by manipulating key variables to see potential outcomes. This is crucial for strategic planning and risk management, providing insights into how different factors will interact over the long term.
- **Avoiding spurious predictions**: correlational models may lead to spurious predictions when unseen confounding variables are at play. Causal models help control for these confounders, ensuring that predictions are based on genuine causal effects rather than coincidental correlations.

CHAPTER

FIVE

WHAT TO EXPECT FROM EACH CHAPTER

The content is currently divided into five main parts. Each part aims to provide a set of alternatives to tackle **one key objective**. This is your map if you are not sure what you need.

Part	Name	Key Objective	Description
I	Basic Concepts	**Getting an overview of causal inference**	Introduction and basic concepts to understand the foundation of causal inference.
II	Causal Discovery	**Discovering causal relationships from data**	Techniques and methods for identifying causal structures and relationships.
III	Causal Inference	**Estimating causal effects accurately**	Methods for quantifying the strength and nature of causal relationships.
IV	Interpretability	**Interpreting machine learning models**	Techniques to understand and explain model predictions and decisions.
V	Experiments and Data Collection	**Designing effective experiments**	Approaches for planning, conducting, and analysing controlled experiments.

5.1 Detailed overviews

For parts II to V, we have included an overview chapter at the beginning of each section. These overview chapters are designed to provide a thorough introduction and a quick summary of the primary content, making it easier for you to understand the overall scope and focus of each part.

Part III

I. Basic Concepts

CHAPTER

SIX

CORRELATION VS. CAUSATION

When addressing real-world engineering or business challenges, a fundamental goal is to comprehend the underlying mechanisms governing a system and the **key drivers** affecting its performance. This understanding often stems from conducting experiments, which are foundational to scientific research. For example, if we aim to determine the effect of a new solar panel design on energy conversion efficiency, systematic experimentation might be the initial approach. Through controlled manipulation and observation, **experimental data** helps us establish clear cause-and-effect relationships.

> In the context of data analysis and research, a **cause** refers to an event or a variable that directly influences another event or variable. If a change in X (the cause) results in a change in Y (the effect), then X is said to cause Y. This relationship is often depicted as $X \rightarrow Y$, signifying that X is a causal factor for Y.

In many real-world settings, such as economics or public policy, performing experiments can be impractical, expensive, or unethical. Consequently, researchers and analysts often rely on **observational data**, which is collected without any direct manipulation of variables. Working with observational data introduces complexities, particularly in distinguishing between mere correlations and actual causal relationships. Observational data can reveal patterns and associations, but without the ability to control for all influencing factors, these correlations might lead to misleading conclusions about causality.

Causal inference bridges this gap. It is a branch of statistics focused on determining cause-and-effect relationships from data where no explicit experimentation has been conducted. Unlike mere correlation analysis, which only measures how variables move together, causal inference aims to uncover whether and how one variable influences another. This distinction is vital for informed decision-making across various domains, including economics, healthcare, and energy markets. The phrase "correlation does not imply causation" is frequently cited in statistical analysis to caution against prematurely concluding that one variable causes another simply because they are associated. Despite this, it is surprising how often business decisions are made based on perceived associations highlighted by correlation metrics alone.

In the electricity market, understanding causal relationships is crucial. It enables stakeholders to predict the effects of policy changes, modify pricing strategies, and assess the impact of technological advancements on demand and supply dynamics. Causal inference provides the tools to make predictions about the consequences of potential actions, thereby supporting more strategic and effective decision-making.

A Motivating Example

To illustrate the dichotomy between correlation and causation, consider a simple example involving three variables:

1. Temperature (°C)
2. Electricity load (MW)
3. Ice cream sales (GBP)

To better understand the relationships among these variables, we utilise a **causal graph**. In this scenario, we assume knowledge of the true causal structure of the data-generating process: temperature affects both electricity load and ice cream sales.

A causal graph is typically assumed to be a **directed acyclic graph (DAG)**, which is a graphical representation used to model and reason about the causal relationships between variables. In a DAG:

- **Nodes** represent variables or events.
- **Edges** (arrows) indicate causal influences from one variable to another.

When we say that the graph is **acyclic**, it means that it does not contain any loops, ensuring that causality flows in one direction and does not circle back on itself.

In the causal graph below, **temperature** is depicted as a parent node influencing two child nodes: **electricity load** and **ice cream sales**. This structure helps us visualise and understand that fluctuations in temperature are the underlying drivers for changes in electricity consumption and ice cream sales. Typically, we are also interested in understanding the **strength** of these causal connections. However, for now, let's not consider the specific impacts of these weights.

```
import graphviz
from IPython.display import display

# Create a new graph
dot = graphviz.Digraph()

# Add nodes
dot.node('T', 'Temperature')
dot.node('L', 'Electricity\nload')
dot.node('I', 'Ice cream\nsales')

# Add edges
dot.edge('T', 'L')
dot.edge('T', 'I')

# Display the graph in the notebook
display(dot)
```

```
<graphviz.graphs.Digraph at 0x104ae7a10>
```

Assume we have collected a dataset that follows the causal structure outlined in the previously discussed causal graph. However, let's also assume that we are unaware of this true causal structure and need to perform exploratory analysis to uncover potential correlations among the observations.

In this scenario, our goal is to explore the data, look for correlations, and attempt to infer possible causal relationships without prior knowledge of the underlying causal mechanisms. This approach mimics real-world situations where data scientists and researchers often work with complex datasets without a clear understanding of the dynamics that govern the relationships between variables.

Through statistical methods and visual analysis, we will examine the interactions between temperature, electricity load, and ice cream sales, exploring how these variables may be related and considering the implications of our findings.

First, let's generate some data according to the DAG shown above.

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

# Setting a random seed for reproducibility
np.random.seed(42)

# Generating synthetic data
n = 100000
temperatures = np.random.normal(20, 5, n)  # average temperature in Celsius
```

(continues on next page)

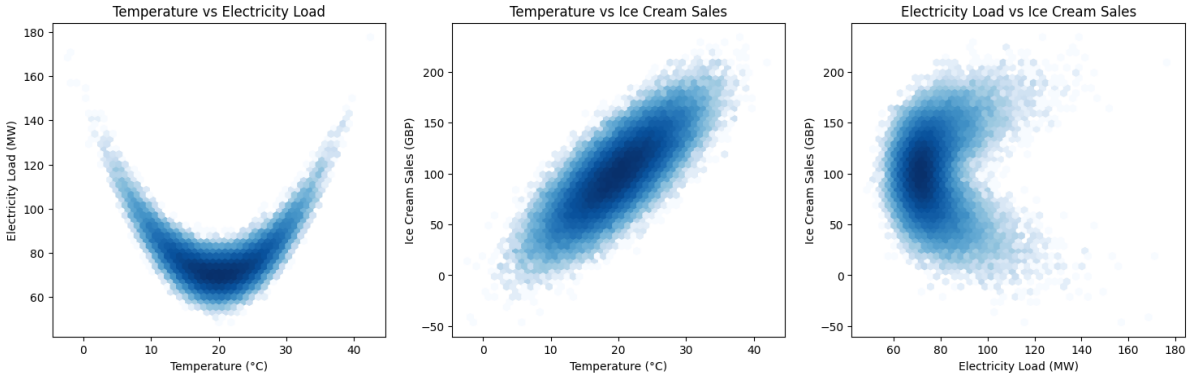
(continued from previous page)

```
electricity_load = 0.2 * (temperatures - 20)**2 + 70 + np.random.normal(0, 5, n)  #␣
↪Quadratic relationship for U-shape
ice_cream_sales = 5* temperatures + np.random.normal(0, 20, n)  # also influenced by␣
↪temperature

# Creating a DataFrame
data = pd.DataFrame({'Temperature': temperatures, 'Electricity Load': electricity_
↪load, 'Ice Cream Sales': ice_cream_sales})

data.head()
```

	Temperature	Electricity Load	Ice Cream Sales
0	22.483571	76.386598	143.654668
1	19.308678	64.318811	94.658826
2	23.238443	74.974688	89.601501
3	27.615149	78.501907	110.302984
4	18.829233	68.637125	87.293150

Now, let's plot the pairwise correlation plots to explore the dependencies.

```
# Setting up the plots
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Temperature vs Electricity Load with hexbin plot
hb1 = axes[0].hexbin(data['Temperature'], data['Electricity Load'], gridsize=50, cmap=
↪'Blues', bins='log')
axes[0].set_title('Temperature vs Electricity Load')
axes[0].set_xlabel('Temperature (°C)')
axes[0].set_ylabel('Electricity Load (MW)')

# Temperature vs Ice Cream Sales with hexbin plot
hb2 = axes[1].hexbin(data['Temperature'], data['Ice Cream Sales'], gridsize=50, cmap=
↪'Blues', bins='log')
axes[1].set_title('Temperature vs Ice Cream Sales')
axes[1].set_xlabel('Temperature (°C)')
axes[1].set_ylabel('Ice Cream Sales (GBP)')

# Electricity Load vs Ice Cream Sales with hexbin plot
hb3 = axes[2].hexbin(data['Electricity Load'], data['Ice Cream Sales'], gridsize=50,␣
↪cmap='Blues', bins='log')
axes[2].set_title('Electricity Load vs Ice Cream Sales')
axes[2].set_xlabel('Electricity Load (MW)')
axes[2].set_ylabel('Ice Cream Sales (GBP)')

plt.show()
```



By examining these plots, we might find ourselves tempted to conclude a correlation between electricity load and ice cream sales, especially in a more complex scenario where the causal structure is not known to us. This observation highlights a common challenge in data analysis: distinguishing between mere correlation and actual causation without prior knowledge of the underlying causal mechanisms.

CHAPTER

SEVEN

CAUSAL REPRESENTATIONS

Directed acyclic graphs (DAGs) are essential tools for understanding causal relationships between variables. They are widely used in statistics, machine learning, and causal inference.

The **key elements** of a DAG are:

1. **Node**: represents a variable or an event in the graph.
2. **Edge**: a directed arrow indicating a causal influence from one node (variable) to another.
3. **Parent node**: a node that has outgoing edges to one or more child nodes.
4. **Child node**: a node that has incoming edges from one or more parent nodes.

The **key properties** of a DAG are:

1. **Directed**: each edge in the graph has a direction, from one vertex to another.
2. **Acyclic**: there are no cycles; you cannot start at one vertex, follow a sequence of directed edges, and return to the starting vertex.
3. **Topological ordering**: the nodes of a DAG can be ordered in such a way that for every directed edge u → v, node u comes before node v in the ordering.

A DAG is a graph that is directed and has no cycles, meaning there is no way to start at one node and follow a consistent direction that leads back to the starting node.

> In a DAG, every parent is a direct cause of all its children.

To better understand DAGs, let's consider a simple example with the following variables:

1. **Temperature**
2. **Electricity load**
3. **Renewable energy production**
4. **Electricity price**

```
import graphviz
from IPython.display import display

# Create a new graph
dot = graphviz.Digraph()

# Add nodes
dot.node('T', 'Temperature')
dot.node('L', 'Electricity\nload')
dot.node('R', 'Renewable\nenergy production')
dot.node('P', 'Electricity\nprice')
```

(continues on next page)

(continued from previous page)

```
# Add edges
dot.edge('T', 'L')
dot.edge('T', 'R')
dot.edge('R', 'P')
dot.edge('L', 'P')

# Display the graph in the notebook
display(dot)
```

```
<graphviz.graphs.Digraph at 0x104280190>
```

Understanding the DAG

A DAG is used to visually represent the causal relationships between the key variables in the analysis. The key variables are represented as nodes, and the arrows represent the direction of influence. By examining the graph, we can understand how changes in one variable may impact others, forming a network of dependencies that govern market dynamics. In the example above, we have:

1. **Temperature → electricity load**: as temperature increases or decreases, it directly affects the electricity load. With warmer temperatures, the demand for electricity increases due to air conditioning usage. Similarly, colder temperatures can increase electricity demand for heating purposes.
2. **Temperature → renewable energy production**: temperature can affect the efficiency and output of renewable energy sources. For example, solar panels may produce more energy on sunny, warm days but less on extremely hot days when efficiency drops. Wind patterns, affected by temperature changes, can also impact wind energy production.
3. **Renewable energy production → electricity price**: the amount of renewable energy produced affects electricity prices. When renewable energy production is high, it can lead to lower electricity prices due to the abundance of cheaper energy sources. Conversely, low renewable energy production can increase reliance on more expensive, non-renewable energy sources, driving up prices.
4. **Electricity load → electricity price**: the demand for electricity (load) influences electricity prices. High electricity demand usually leads to higher prices because the supply must meet the increased load, often requiring more expensive or less efficient energy sources. Lower demand can result in lower prices due to decreased strain on the electricity grid and reliance on cheaper energy sources.

Additional Concepts in DAGs

d-separation

In addition to understanding direct causal relationships, it is also important to understand how to determine if variables are conditionally independent given other variables. This concept is known as **d-separation**, and it represents a criterion for deciding whether a set of nodes is independent of another set of nodes given a third set. It is a crucial concept in understanding the flow of information and causation in DAGs.

For example, in the context of our electricity market DAG:

- **Temperature** is d-separated from **electricity price** given **electricity load**. This means that if we know the electricity load, knowing the temperature provides no additional information about the electricity price.
- **Renewable energy production** and **electricity load** are not d-separated, indicating a direct causal relationship without any conditional independence given other variables.

By understanding d-separation, we can better interpret the dependencies and independencies in our DAG, leading to more accurate causal inferences.

Local Markov assumption

The Local Markov ssumption states that a node in a DAG is conditionally independent of its non-descendants given its parents. This assumption allows for the decomposition of the joint probability distribution of all variables in the DAG into simpler conditional distributions. For example, in our DAG, **electricity price** is conditionally independent of **temperature** given **renewable energy production** and **electricity load**. This implies that once we know the values of **renewable energy production** and **electricity load**, additional information about **temperature** does not change our understanding of **electricity price**.

Factorization

Factorization refers to the decomposition of a joint probability distribution into a product of conditional distributions. This is possible under the Local Markov Assumption. The joint distribution $P(T, L, R, P)$ can be factorized as:

$$P(T, L, R, P) = P(T) \cdot P(L|T) \cdot P(R|T) \cdot P(P|R, L) \tag{7.1}$$

This factorization simplifies the computation of the joint probability distribution by breaking it down into manageable parts. It allows us to understand the contribution of each variable to the overall distribution, aiding in both analysis and inference.

Minimality assumption

The Minimality Assumption asserts that the DAG representing the causal structure is minimal, meaning that removing any edge would violate the Markov condition for the observed data. It ensures that the model encodes only necessary dependency relationships. It means that the DAG includes only those edges that represent necessary causal relationships. Removing any edge would lead to a loss of information about the dependencies among variables, ensuring that the DAG is the simplest model that adequately represents the data.

Markov equivalence class

A Markov equivalence class contains all DAGS that encode the same conditional independencies. This means that within a Markov equivalence class, multiple DAGs can represent the same set of conditional independence relationships among variables. These DAGs are considered equivalent because they imply the same probabilistic dependencies, even if their structures differ.

Consider the following three DAGs involving three variables X, Y, and Z:

Graph 1:

```
dot = graphviz.Digraph()
dot.node('X')
dot.node('Y')
dot.node('Z')
dot.edge('X', 'Y')
dot.edge('Y', 'Z')
display(dot)
```

```
<graphviz.graphs.Digraph at 0x106159310>
```

Graph 2:

```
dot = graphviz.Digraph()
dot.node('X')
dot.node('Y')
dot.node('Z')
dot.edge('Y', 'X')
dot.edge('Y', 'Z')
display(dot)
```

```
<graphviz.graphs.Digraph at 0x10615a210>
```

Graph 3:

```
dot = graphviz.Digraph()
dot.node('X')
dot.node('Y')
dot.node('Z')
dot.edge('X', 'Y')
dot.edge('Z', 'Y')
display(dot)
```

```
<graphviz.graphs.Digraph at 0x106159610>
```

Each of these DAGs represents the same conditional independence relationships: $X \perp Z|Y$, indicating that once we know Y, X and Z are independent.

A completed partially directed acyclic graph (CPDAG), also known as an Essential Graph (EG), is a graphical representation that captures all DAGs within a Markov equivalence class. The CPDAG contains both directed and undirected edges:

- **Directed edges** represent causal relationships that are common to all DAGs in the equivalence class.
- **Undirected edges** represent relationships where the directionality is ambiguous among the DAGs in the equivalence class.

The CPDAG can be constructed by:

1. Identifying all DAGs that belong to the same Markov equivalence class.
2. Retaining directed edges that are common to all these DAGs.
3. Converting edges with uncertain directionality into undirected edges.

Let's consider a CPDAG for the equivalence class of the three DAGs shown above:

```
dot = graphviz.Digraph()
dot.node('X')
dot.node('Y')
dot.node('Z')
dot.edge('X', 'Y')
dot.edge('Z', 'Y')
dot.edge('X', 'Z', dir='none')  # undirected edge
display(dot)
```

```
<graphviz.graphs.Digraph at 0x10615bad0>
```

In this CPDAG:

- The edges $X \to Y$ and $Z \to Y$ are directed because all DAGs in the equivalence class have these directions.
- The edge $X - Z$ is undirected, indicating that the direction between X and Z is ambiguous within the equivalence class.

The CPDAG or EG provides a compact and comprehensive representation of the causal structure, capturing all possible DAGs that explain the observed data:

- **Identification of causal relationships**: directed edges in the CPDAG indicate robust causal relationships that hold across all DAGs in the equivalence class.

- **Ambiguity in directionality**: undirected edges highlight areas where additional data or assumptions are needed to resolve the direction of causality.
- **Simplified analysis**: analysing a CPDAG instead of multiple individual DAGs simplifies the process of understanding the underlying causal structure.

CHAPTER

EIGHT

BASIC CAUSAL STRUCTURES

Understanding basic causal structures is crucial in causal inference because they possess **unique properties** that aid in the discovery of causal graphs from observational data. Additionally, recognizing these structures is essential for correctly specifying **regression models** in statistical analysis. Misidentifying or incorrectly including variables based on these structures can lead to erroneous conclusions and model estimates.

8.1 Chains

A chain occurs when one variable causally affects a second, which in turn affects a third. This linear sequence shows the **transmission of causality** through an intermediary.

```
import numpy as np
from lingam.utils import make_dot

# Matrix of coefficients (weights)
m = np.array([[0.0, 0.0, 0.0],
              [1.5, 0.0, 0.0],
              [0.0, 1.2, 0.0]])

# Plotting causal graph
make_dot(m, labels=["weather \nfluctuations", "wind forecast \nerrors", "balancing \
↪ncosts"])
```

```
<graphviz.graphs.Digraph at 0x13d5cfe10>
```

This causal graph represents the following **structural causal model (SCM)**:

$$\text{weather fluctuations} = e_w \tag{8.1}$$

$$\text{wind forecast errors} = 1.5 \times \text{weather flucturations} + e_f \tag{8.2}$$

$$\text{balancing costs} = 1.2 \times \text{wind forecast errors} + e_b \tag{8.3}$$

$$\tag{8.4}$$

where the **coefficients** 1.5 and 1.2 represent the strenghts of the causal connections.

In this case, the **causal association flows** from the weather forecast to the balancing costs, and is (fully) **mediated** by the wind forecast errors.

8.2 Forks

A fork occurs when a single variable causally influences two other variables, making it a common cause to both. This structure typically indicates that the two downstream variables are **correlated due to a shared source** but do not causally influence each other. This coincides with the exmaple we saw in the previous chapter, where the common cause (temperature) gave the impression that electricity load and ice cream sales were correlated.

Another example, related to the balancing costs in the electricity markets, might be the effect of weather fluctuations on wind forecast errors and on electricity load due to increase heating (or cooling) demand.

```
# Matrix of coefficients (weights)
m = np.array([[0.0, 0.0, 0.0],
              [1.5, 0.0, 0.0],
              [0.7, 0.0, 0.0]])

# Plotting causal graph
make_dot(m, labels=["weather \nfluctuations", "wind forecast \nerrors", "heating \
 ↪\ndemand"])
```

```
<graphviz.graphs.Digraph at 0x13e88fb10>
```

This causal graph represents the following **SCM**:

$$\text{weather fluctuations} = e_w \tag{8.5}$$

$$\text{wind forecast errors} = 1.5 \times \text{weather flucturations} + e_f \tag{8.6}$$

$$\text{heating demand costs} = 0.7 \times \text{heating demand} + e_h \tag{8.7}$$

$$\tag{8.8}$$

where $e_w, e_f, e_h \sim \mathcal{N}(0, 1)$.

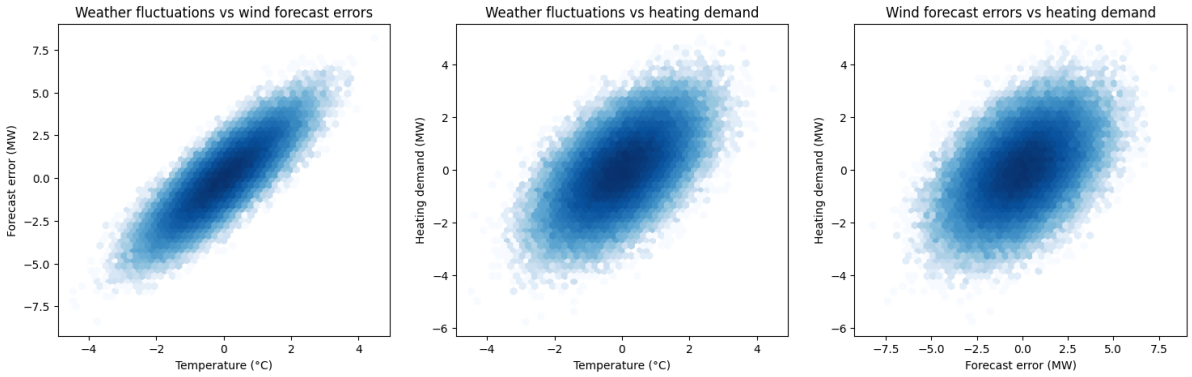
First, let's generate some data from this DAG.

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

# Setting a random seed for reproducibility
np.random.seed(42)

# Generating synthetic data
n = 100000
weather_fluctuations = np.random.normal(0, 1, n)  # average temperature in Celsius
forecast_errors = 1.5 * weather_fluctuations + np.random.normal(0, 1, n)  # Quadratic␣
 ↪relationship for U-shape
heating = 0.7* weather_fluctuations + np.random.normal(0, 1, n)  # also influenced by␣
 ↪temperature

# Creating a DataFrame
data = pd.DataFrame({'weather': weather_fluctuations, 'forecast errors': forecast_
 ↪errors, 'heating demand': heating})

data.head()
```

```
    weather  forecast errors  heating demand
0  0.496714         1.775666        1.909541
1 -0.138264        -1.362751       -0.191013
2  0.647689         1.546970       -0.876154
3  1.523030         1.665306       -0.322517
4 -0.234153        -0.678633       -0.506558
```

Now, let's plot the pairwise correlation plots.

```
# Setting up the plots
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Temperature vs Electricity Load with hexbin plot
hb1 = axes[0].hexbin(data['weather'], data['forecast errors'], gridsize=50, cmap=
↪'Blues', bins='log')
axes[0].set_title('Weather fluctuations vs wind forecast errors')
axes[0].set_xlabel('Temperature (°C)')
axes[0].set_ylabel('Forecast error (MW)')

# Temperature vs Ice Cream Sales with hexbin plot
hb2 = axes[1].hexbin(data['weather'], data['heating demand'], gridsize=50, cmap='Blues
↪', bins='log')
axes[1].set_title('Weather fluctuations vs heating demand')
axes[1].set_xlabel('Temperature (°C)')
axes[1].set_ylabel('Heating demand (MW)')

# Electricity Load vs Ice Cream Sales with hexbin plot
hb3 = axes[2].hexbin(data['forecast errors'], data['heating demand'], gridsize=50,␣
↪cmap='Blues', bins='log')
axes[2].set_title('Wind forecast errors vs heating demand')
axes[2].set_xlabel('Forecast error (MW)')
axes[2].set_ylabel('Heating demand (MW)')

plt.show()
```



We can see that, despites **not being causally related**, wind forecast errors and heating demand appears to be **positively correlated**. This is because they have a **common cause**, represented by the weather fluctuations.

8.3 Immoralities

An immorality happens when two variables independently cause a third variable, but there is no causal connection between the two independent variables. Conditioning on the common effect (**collider**) can introduce a **spurious association** between these independent variables.

```
# Matrix of coefficients (weights)
m = np.array([[0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0],
              [2.0, 1.5, 0.0]])

# Plotting causal graph
make_dot(m, labels=["forecast \nerrors", "outages", "balancing \ncosts"])
```

```
<graphviz.graphs.Digraph at 0x105a58e10>
```

This causal graph represents the following **SCM**:

$$\text{forecast errors} = e_f \tag{8.9}$$

$$\text{outages} = e_o \tag{8.10}$$

$$\text{balancing costs} = 2 \times \text{forecast errors} + 1.5 \times \text{outages} + e_b \tag{8.11}$$

$$\tag{8.12}$$

where $e_f, e_o, e_b \sim \mathcal{N}(0, 1)$.

We will now see the effect of the **Berkson's paradox**, which is the spurious correlation observed in the two parent variables, after conditioning on the collider.

First, let's generate some data.

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

# Set the random seed for reproducibility
np.random.seed(42)

# Generate data for the Immorality example
n = 100000
forecast_errors = np.random.normal(0, 1, n)
outages = np.random.normal(0, 1, n)
balancing_costs = 2 * forecast_errors + 1.5 * outages + np.random.normal(0, .1, n)  #␣
 ↪the collider

data_immorality = pd.DataFrame({'forecast errors': forecast_errors, 'outages':␣
 ↪outages, 'balancing costs': balancing_costs})

data_immorality.head()
```

	forecast errors	outages	balancing costs
0	0.496714	1.030595	2.695504
1	-0.138264	-1.155355	-2.018984
2	0.647689	0.575437	2.025579

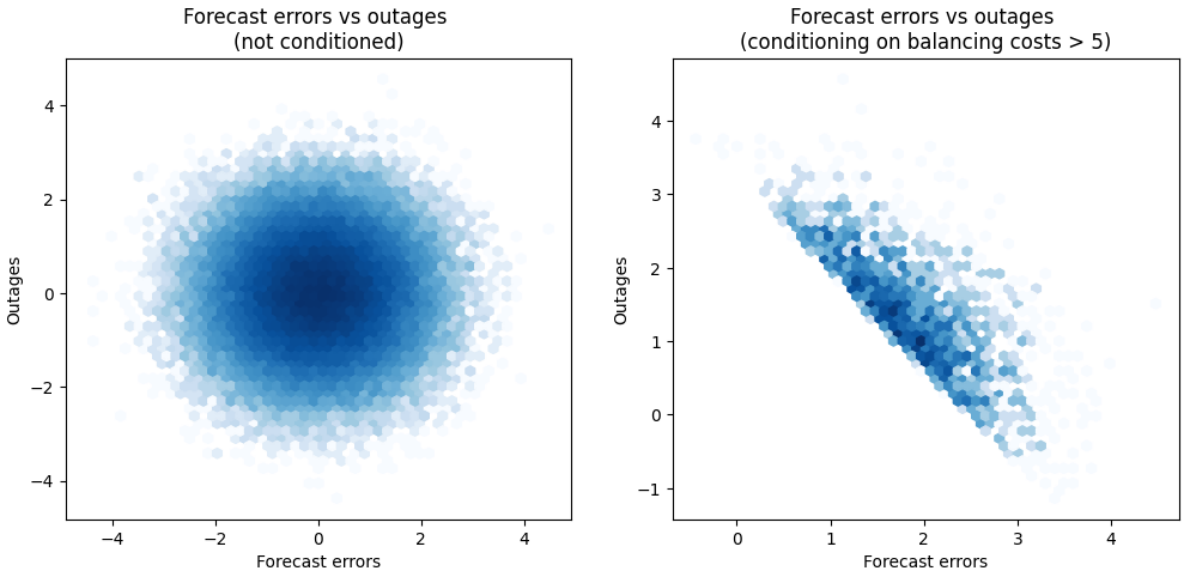
(continues on next page)

(continued from previous page)

```
3          1.523030 -0.619238          1.978338
4         -0.234153 -0.327403         -0.993676
```

Now, let's take a look at the correlations, after conditioning on the child variable.

```
# Setting up the plots
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# All the data
hb1 = axes[0].hexbin(data_immorality['forecast errors'], data_immorality['outages'],␣
↪gridsize=50, cmap='Blues', bins='log')
# cb1 = fig.colorbar(hb1, ax=axes[0])
axes[0].set_title('Forecast errors vs outages \n(not conditioned)')
axes[0].set_xlabel('Forecast errors')
axes[0].set_ylabel('Outages')

# After conditioning
data_immorality_conditioned = data_immorality[data_immorality['balancing costs'] >= 5]
hb2 = axes[1].hexbin(data_immorality_conditioned['forecast errors'], data_immorality_
↪conditioned['outages'], gridsize=50, cmap='Blues', bins='log')
# cb2 = fig.colorbar(hb2, ax=axes[1])
axes[1].set_title('Forecast errors vs outages \n(conditioning on balancing costs > 5)
↪')
axes[1].set_xlabel('Forecast errors')
axes[1].set_ylabel('Outages')

plt.show()
```



We can observe how conditioning on a collider introduces spurious correlations in the data. Outages and forecast errors are **not causally related** but, when conditioning on the balancing costs, they **appear to be correlated**.

This underscores the importance of being cautious when attempting to draw causal conclusions from observational data.

In practice, this means that merely including the collider as a predictor in a **regression model** can lead to misleading results, where the estimated coefficients deviate significantly from the true causal effects. This potential misstep highlights the necessity of carefully selecting variables for inclusion in regression models, especially when the goal is to uncover causal

relationships rather than mere associations.

CHAPTER

NINE

DEFINITIONS AND TERMINOLOGY

Understanding the terminology and concepts in causal inference is crucial for grasping the methodologies and their applications. This chapter is designed to be a vademecum that you can consult whenever you have a doubt. Whether you are just starting your causal inference course or revisiting the concepts after a few months, this glossary will provide you with a refresher on the terms you need to remember.

9.1 Causal structures and graphical representations

- **Causal graph**: a visual representation of causal relationships among a set of variables, often depicted as a directed acyclic graph (DAG).
- **Collider**: a variable that is influenced by two or more other variables in a causal graph, potentially introducing bias when conditioned upon.
- **Confounder**: a variable that influences both the treatment and the outcome, potentially biasing the estimated effect of the treatment.
- **Directed acyclic graph (DAG)**: a graphical representation of causal relationships between variables. Each node represents a variable, and each directed edge represents a causal effect from one variable to another. The graph is acyclic, meaning it does not contain any cycles.
- **D-separation**: a criterion used to determine conditional independence between two sets of nodes in a directed acyclic graph (DAG), given a third set of nodes. It helps identify whether paths between nodes are blocked by a conditioning set.
- **Instrumental variable (IV)**: a variable that is used to estimate causal relationships when controlled experiments are not feasible. The IV affects the treatment but has no direct effect on the outcome except through the treatment.
- **Local Markov assumption**: states that a node in a DAG is conditionally independent of its non-descendants given its parents. This assumption allows for the decomposition of the joint probability distribution of all variables in the DAG into simpler conditional distributions.
- **Mediator**: a variable that lies on the causal path between the treatment and the outcome, helping to explain the mechanism through which the treatment affects the outcome.
- **Minimality assumption**: Asserts that the DAG representing the causal structure is minimal, meaning that removing any edge would violate the Markov condition for the observed data. It ensures that the model encodes only necessary dependency relationships.

9.2 Experimental design and analysis

- **A/B testing**: a randomised controlled experiment used to compare two versions of a variable to determine which performs better, commonly used in marketing and product development.
- **Control group and treatment group**: in experimental design, the control group is the group that does not receive the treatment or intervention, while the treatment group is the group that receives the treatment or intervention. Comparing the outcomes of these two groups helps to estimate the causal effect of the treatment.
- **Counterfactuals**: this framework allows for the analysis of counterfactual questions—what would happen to one variable if another were changed, holding everything else constant.
- **Design of experiments (DoE)**: a systematic approach to planning, conducting, analysing, and interpreting controlled experiments to ensure valid, reliable, and replicable results.
- **Ignorability/exchangeability assumption**: sssumes that the way individuals are assigned to treatments can be ignored, allowing for the assumption of random assignment in observational studies. This is crucial for identifying causal effects.
- **Placebo effect**: a phenomenon where subjects experience a perceived or actual improvement in their condition despite receiving a non-active treatment, due to their expectations of the treatment's efficacy.
- **Randomisation**: the process of assigning subjects to treatment and control groups by chance, reducing bias and ensuring that the groups are comparable.
- **Regression discontinuity design (RDD)**: a quasi-experimental design that exploits a cutoff or threshold to assign treatments in order to estimate causal effects.
- **Selection bias**: a bias that occurs when the subjects included in a study are not representative of the population intended to be analysed, leading to incorrect conclusions.
- **Synthetic control method**: a method for estimating causal effects by comparing treated units to a weighted combination of untreated units that best approximates the characteristics of the treated units.

9.3 Effects and estimates

- **Average treatment effect (ATE)**: the difference in the average outcomes between the treatment group and the control group. It measures the overall impact of the treatment on the population.
- **Causal effect**: the change in an outcome directly attributable to a change in a treatment or exposure.
- **Conditional average treatment effect (CATE)**: the average treatment effect for a specific subset of the population, defined by certain characteristics or conditions. It provides a more granular understanding of how the treatment effect varies across different groups within the population.
- **Confounding bias**: a bias that occurs when the treatment effect is mixed with the effect of a confounder, leading to an incorrect estimation of the causal effect.
- **Endogeneity**: a situation in which an explanatory variable is correlated with the error term in a regression model, leading to biased and inconsistent parameter estimates.
- **Exogeneity**: a condition where an explanatory variable is not correlated with the error term, ensuring unbiased and consistent parameter estimates.
- **External validity**: the extent to which the results of a study can be generalised to other settings, populations, or time periods.
- **Heterogeneity**: the variation in treatment effects across different subgroups or individuals within a population.

- **Hidden common causes**: variables that are not observed but influence multiple observed variables, leading to dependent error terms. Addressing hidden common causes is crucial for accurate causal inference.
- **Internal validity**: the extent to which the observed effects in a study are due to the treatment and not to other factors.
- **Propensity score**: the probability of a unit being assigned to the treatment group given a set of observed covariates. Used to reduce bias in the estimation of treatment effects in observational studies.
- **Robustness check**: methods used to test the stability of the estimated effects under different model specifications or assumptions.
- **Sensitivity analysis**: a technique used to determine how the results of a study or model change when the assumptions or parameters are varied.
- **Spillover effect**: an effect where the treatment of one group or individual influences the outcomes of another group or individual, potentially biasing the estimated treatment effect.
- **SUTVA**: the stable unit treatment value assumption (SUTVA) is a fundamental assumption in causal inference and experimental design. It asserts that:
  - The treatment assigned to one unit does not affect the outcomes of other units.
  - Each unit's outcome depends only on its own treatment, not on the treatments assigned to other units.
  - There are no different versions of the treatment.
  - The treatment is applied uniformly across all treated units.

Part IV

II. Causal Discovery

CHAPTER

TEN

OVERVIEW

Causal discovery involves identifying causal relationships from data. This process uncovers the underlying causal structure without assuming prior knowledge of the direction or nature of causality. It typically employs statistical and computational methods to determine which variables influence others, providing a foundation for further causal analysis.

The key objective of causal discovery is often to retrieve and visualize a **causal graph from observational data**. In general, the causal graph for a specific problem can be retrieved relying on:

- **Domain knowledge**: in some cases, there are well-known principles and mechanisms governing the processes under investigation. We might be able to draw causal graphs relying on the expertise of practitioners or on the findings reported in previous studies.
- **Data-driven methods**: when we cannot assume prior knowledge about the process, we can obtain a causal graph using causal discovery algorithms such as the ones discussed in this chapter.
- **Hybrid approaches**: in many cases, the two aforementioned approaches are complementary. For example, we could:
  - **Start with domain knowledge**: if experts can only outline an incomplete causal graph, we can include this knowledge in the causal discovery algorithm to discover the missing relationships.
  - **Start with the data**: we might derive a tentative causal graph from a causal discovery algorithm and then iteratively validate and refine it through consultation with engineers or practitioners.

With regard to data-driven methods, there are two main approaches:

- **Independence-based causal discovery**: this class of methods involves analyzing basic DAG structures like **chains** and **forks** to interpret how variables influence each other. This approach relies on statistical tests to identify conditional independencies among variables, which can then be used to infer the structure of the causal graph. Methods such as the PC algorithm [SGS01], the Fast Causal Inference (FCI) algorithm [Spi01], or the PCMCI algorithm [RNK+19] fall into this category. One of the main drawbacks of these methods is that, in many cases, they can only identify the **Markov equivalence class** of the graph. This means that while we can identify sets of relationships that are consistent with the observed independencies, we cannot definitively determine the direction of causality between all the variables. For instance, if we have only two variables, which are independent, this method cannot specify whether one causes the other or if they are causally unrelated.
- **Semi-parametric causal discovery**: this approach involves making some assumptions about the functional form of the relationships between variables but remains flexible by not fully specifying a parametric model. Methods like the Linear Non-Gaussian Acyclic Model (LiNGAM) [SHHyvarinen+06] are examples of this approach. Going beyond the Markov equivalence class comes at the expenses of making more specific **assumptions about the functional forms** of relationships between variables. These assumptions allow for a more detailed discovery of the causal graph, often identifying specific causal directions rather than just equivalence classes. By assuming certain **non-linearities** or **specific distributions of errors**, semi-parametric methods can exploit asymmetries in the data that reveal the direction of causal effects. This approach is more powerful in that it can often discern the actual causal structure rather than just a set of possible structures. However, the trade-off is that these methods require stronger assumptions about the nature of the data and the relationships involved, which may not always hold true.

In the upcoming Chapters, we focus primarily on semi-parametric approaches, inspired by the methodologies presented in Statistical Causal Discovery: LiNGAM Approach. Most of the examples hereby presented have been implemented using the LiNGAM Python package [IIZ+23].

10.1 Content of Causal Discovery chapters

Chapter	Description
Linear Models	How to retrieve the causal graph from the data, if we can assume a linear model with non-Gaussian noise for the data-generating process.
Nonlinear Models	Extends the LiNGAM approach by considering nonlinear functional forms, and accomodating for Gaussian noise.
Time Series Models	Extends the LiNGAM approach by providing a method for detecting contemporaneous and lagged effects in time series data.
Structural Breaks	Highilight the challenges of trying to identifying causal graphs in dynamic and evolving environments.

CHAPTER

ELEVEN

LINEAR MODELS

If we are willing to make assumptions about the **functional form** of the structural causal model or the **distribution of the errors**, we can go beyond the Markov equivalence class and identify the complete causal structure of the data-generating process. In this first chapter of semi-parametric models, we introduce the **linear non-Gaussian acyclic model (LiNGAM)** [SHHyvarinen+06].

11.1 The LiNGAM model

The LiNGAM model for p observed variables $x_1, x_2, \dots, x_p$ is given by

$$x_i = \sum_{j \in \text{pa}(x_i)} b_{ij} x_j + e_i \quad (i = 1, \dots, p) \tag{11.1}$$

where each observed variable x_i is a linear sum of their parent variables $\text{pa}(x_i)$ plus some noise e_i. If the coefficient b_{ij} is zero, then there is no direct causal effect from x_j to x_i. The error variables e_i are **independent** and follow continuous **non-Gaussian** distributions. The independence means there are **no unobserved or hidden common causes**.

In matrix notation, the model is given by

$$\mathbf{x} = \mathbf{B}\mathbf{x} + \mathbf{e} \tag{11.2}$$

where $\mathbf{x}$ and $\mathbf{e}$ are p-dimensional vectors, and $\mathbf{B}$ is a $p \times p$ matrix that contains the b_{ij} coefficients, with $i, j = 1, \dots, p$.

Let us consider a simple **example** with three variables. To ease the introduction of the LiNGAM model, we will use the variables from the chain example of the previous chapters, where:

- **Wind forecast errors** are directly influences by sudden changes in the weather.
- **Balancing costs** are influenced by wind forecast errors. For example, if the system is short because it was expecting a larger production from wind farms, it might incur balancing costs to procure electricity from alternative sources.
- **Weather fluctuarions** are influenced by external factors and serve as an exogenous variable in this model.

11.1.1 Strucural equations

If we know the coefficients of those causal relations, the **SCM** can be written as

$$\text{wind forecast errors} = 1.5 \times \text{weather flucturations} + e_f \tag{11.3}$$

$$\text{balancing costs} = 1.2 \times \text{wind forecast errors} + e_b \tag{11.4}$$

$$\text{weather fluctuations} = e_w \tag{11.5}$$

$$\tag{11.6}$$

11.1.2 Matrix notation

Then, in matrix notation, this model is given by

$$\begin{bmatrix} \text{wind forecast errors} \\ \text{balancing costs} \\ \text{weather fluctuations} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1.5 \\ 1.2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \text{wind forecast errors} \\ \text{balancing costs} \\ \text{weather fluctuations} \end{bmatrix} + \begin{bmatrix} e_f \\ e_b \\ e_w \end{bmatrix} \tag{11.7}$$

11.1.3 Causal ordering

Typically, we **order** the the structural equations according to the **true causal order**, so that the matrix **B** is permuted to be a **lower triangular matrix** (this property is used later on to identify the true structure), with all the diagonal elements equal to zero (strictly lower triangular). This simply means that we rewrite the structural equations as

$$\text{weather fluctuations} = e_w \tag{11.8}$$

$$\text{wind forecast errors} = 1.5 \times \text{weather flucturations} + e_f \tag{11.9}$$

$$\text{balancing costs} = 1.2 \times \text{wind forecast errors} + e_b \tag{11.10}$$

$$\tag{11.11}$$

that, in matrix notation, becomes

$$\begin{bmatrix} \text{weather fluctuations} \\ \text{wind forecast errors} \\ \text{balancing costs} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1.5 & 0 & 0 \\ 0 & 1.2 & 0 \end{bmatrix} \begin{bmatrix} \text{weather fluctuations} \\ \text{wind forecast errors} \\ \text{balancing costs} \end{bmatrix} + \begin{bmatrix} e_w \\ e_f \\ e_b \end{bmatrix} \tag{11.12}$$

Now, we can rewrite the LiNGAM model as

$$x_i = \sum_{j:k(j)<k(i)} b_{ij} x_j + e_i \quad (i = 1, \ldots, p) \tag{11.13}$$

where $k(\cdot)$ represents the causal ordering, meaning that each variable x_i is a linear sum of the x_j variables observed earlier in the causal graph ($k(j) < k(i)$), plus its own error variable e_i. We can now plot the **causal graph**, using the LiNGAM library.

```
import numpy as np
from lingam.utils import make_dot

# Matrix of coefficients (weights)
m = np.array([[0.0, 0.0, 0.0],
              [1.5, 0.0, 0.0],
              [0.0, 1.2, 0.0]])

# Plotting causal graph
make_dot(m, labels=["weather \nfluctuations", "wind forecast \nerrors", "balancing \
 ↪ncosts"])
```

```
<graphviz.graphs.Digraph at 0x13d439f10>
```

11.2 Estimation methods

Now that we explained the basic formulation, we explain how to **estimate the model from the data**. There are two main approaches to estimate a LiNGAM model: a direct estimation method (**DirectLiNGAM**) through a series of regressions and independence tests, and a method based on independent component analysis (ICA) (**ICA-based LinGAM**). In the case of linear models, the key assumption is that the **errors are non-Gaussians**. This is the most important assumption as it allows us to identify the causal structure, both using DirectLiNGAM and ICA-based LiNGAM.

Now, we will explain the intuition behing DirectLiNGAM, an iterative procedure that employs a series of regressions and independence tests to directly identify the causal order among the observed variables. It leverages the non-Gaussianity of the data and the assumption of linear causality to systematically test for independence between variables and their residuals from regressions.

The key intuition is that, with non-Gaussian errors, the **residuals in the anti-causal direction** will not be independent of the predictor. We will see how this works with a practical example.

11.2.1 Example with Gaussian errors

Let us consider a simple example, like the one we saw above, but for simplicity we only keep two variables:

$$\text{weather fluctuations} = e_w \tag{11.14}$$

$$\text{wind forecast errors} = 1.5 \times \text{weather flucturations} + e_f \tag{11.15}$$

$$\tag{11.16}$$

We now assume Gaussian errors, so we have that $e_w, e_f \sim \mathcal{N}(0, 1)$.

We will show that, in this case, by analysing the residuals we cannot understand the true causal direction, because we can both predict the wind from the prices and the prices from the wind.

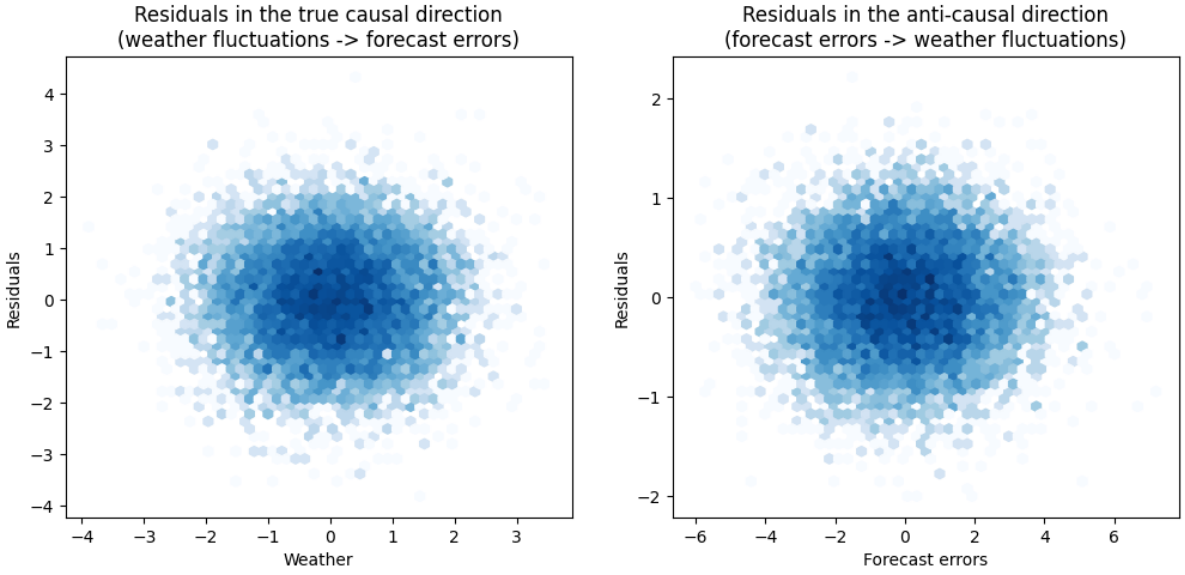
```
import statsmodels.api as sm
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

# Synthetic data generation based on the given equations
n = 10000
weather_fluctuations = np.random.normal(0, 1, size=n)  # Predictor variable
forecast_errors = 1.5 * weather_fluctuations + np.random.normal(0, 1, size=n)  #␣
↪Response variable with non-Gaussian noise (uniform)

# Prepare the data for regression
weather_fluctuations_with_const = sm.add_constant(weather_fluctuations)  # Add a␣
↪constant term for the intercept
forecast_errors_with_const = sm.add_constant(forecast_errors)  # Add a constant term␣
↪for the intercept

# Linear regression in the true causal direction (wind -> price)
true_model = sm.OLS(endog=forecast_errors, exog=weather_fluctuations_with_const)
true_model_results = true_model.fit()
forecast_errors_pred = true_model_results.predict(weather_fluctuations_with_const)
forecast_errors_residuals = forecast_errors - forecast_errors_pred

# Linear regression in the anti-causal direction (price -> wind)
anti_model = sm.OLS(endog=weather_fluctuations, exog=forecast_errors_with_const)
anti_model_results = anti_model.fit()
```

(continues on next page)

(continued from previous page)

```
weather_fluctuations_pred = anti_model_results.predict(forecast_errors_with_const)
weather_fluctuations_residuals = weather_fluctuations - weather_fluctuations_pred

# Setting up the plots
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# All the data
hb1 = axes[0].hexbin(weather_fluctuations, forecast_errors_residuals, gridsize=50,␣
↪cmap='Blues', bins='log')
axes[0].set_title('Residuals in the true causal direction\n(weather fluctuations ->␣
↪forecast errors)')
axes[0].set_xlabel('Weather')
axes[0].set_ylabel('Residuals')

# After conditioning
hb2 = axes[1].hexbin(forecast_errors, weather_fluctuations_residuals, gridsize=50,␣
↪cmap='Blues', bins='log')
axes[1].set_title('Residuals in the anti-causal direction\n(forecast errors ->␣
↪weather fluctuations)')
axes[1].set_xlabel('Forecast errors')
axes[1].set_ylabel('Residuals')
```

```
Text(0, 0.5, 'Residuals')
```



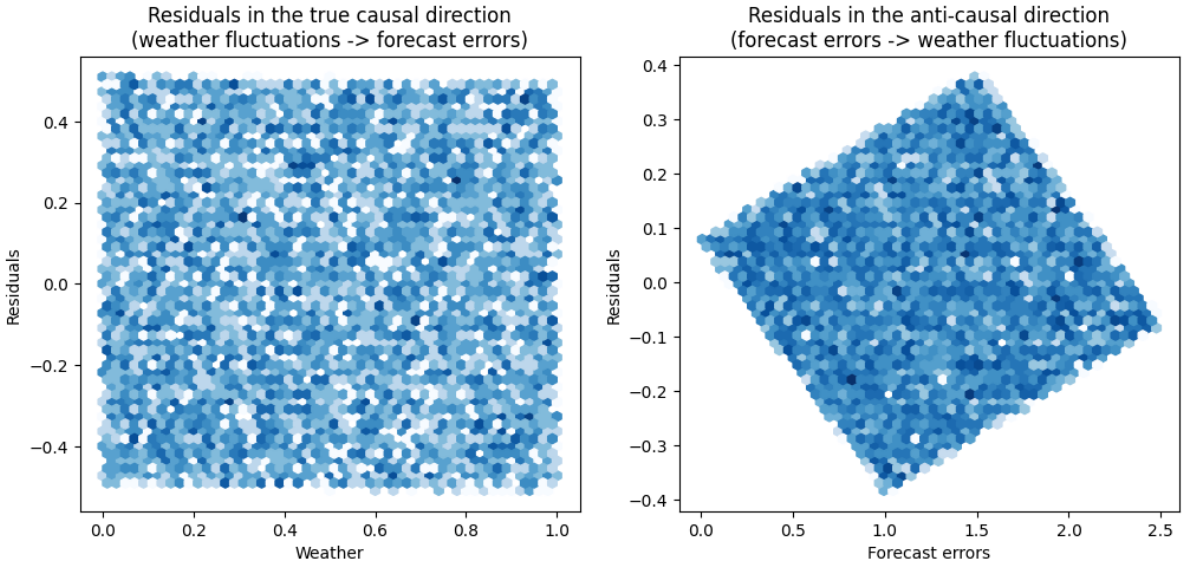
Where we can see how the residuals are independent of the input variable in both the true and anti-causal directions.

11.2.2 Example with non-Gaussian errors

Now, by simply assuming the errors are uniformly distributed, we can wee how it will be possible to detect the true causal direction by testing the independence of the residuals from the input variable.

```
# Synthetic data generation based on the given equations
n = 10000
weather_fluctuations = np.random.uniform(0, 1, size=n)  # Predictor variable
forecast_errors = 1.5 * weather_fluctuations + np.random.uniform(0, 1, size=n)  #␣
↪Response variable with non-Gaussian noise (uniform)

# Prepare the data for regression
weather_fluctuations_with_const = sm.add_constant(weather_fluctuations)  # Add a␣
↪constant term for the intercept
forecast_errors_with_const = sm.add_constant(forecast_errors)  # Add a constant term␣
↪for the intercept

# Linear regression in the true causal direction (wind -> price)
true_model = sm.OLS(endog=forecast_errors, exog=weather_fluctuations_with_const)
true_model_results = true_model.fit()
forecast_errors_pred = true_model_results.predict(weather_fluctuations_with_const)
forecast_errors_residuals = forecast_errors - forecast_errors_pred

# Linear regression in the anti-causal direction (price -> wind)
anti_model = sm.OLS(endog=weather_fluctuations, exog=forecast_errors_with_const)
anti_model_results = anti_model.fit()
weather_fluctuations_pred = anti_model_results.predict(forecast_errors_with_const)
weather_fluctuations_residuals = weather_fluctuations - weather_fluctuations_pred

# Setting up the plots
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# All the data
hb1 = axes[0].hexbin(weather_fluctuations, forecast_errors_residuals, gridsize=50,␣
↪cmap='Blues', bins='log')
axes[0].set_title('Residuals in the true causal direction\n(weather fluctuations ->␣
↪forecast errors)')
axes[0].set_xlabel('Weather')
axes[0].set_ylabel('Residuals')

# After conditioning
hb2 = axes[1].hexbin(forecast_errors, weather_fluctuations_residuals, gridsize=50,␣
↪cmap='Blues', bins='log')
axes[1].set_title('Residuals in the anti-causal direction\n(forecast errors ->␣
↪weather fluctuations)')
axes[1].set_xlabel('Forecast errors')
axes[1].set_ylabel('Residuals')
```

```
Text(0, 0.5, 'Residuals')
```



11.2.3 Full LiNGAM estimation example

Now, we try a full estimation example, where we have a slightly more complex causal structure, that we will need to retrieve from observational data. We now add to the model:

- **Outages**: an exogenous variable affecting the balancing costs.
- **heating demand**: a variable affected by changes in the weather, which affects the balancing costs.

Assume the following **structural equations**:

$$\text{weather fluctuations} = e_w \tag{11.17}$$

$$\text{outages} = e_o \tag{11.18}$$

$$\text{heating demand} = 0.3 \times \text{weather flucturations} + e_h \tag{11.19}$$

$$\text{wind forecast errors} = 1.5 \times \text{weather flucturations} + e_f \tag{11.20}$$

$$\text{balancing costs} = 1.5 \times \text{outages} + 0.7 \times \text{heating demand} + 1.2 \times \text{wind forecast errors} + e_b \tag{11.21}$$

$$\tag{11.22}$$

where the errors are now assumed to be **uniformly distributed**.

Then, the true **causal graph** is given by

```
# Matrix of coefficients (weights)
B = np.array([
    [0.0, 0.0, 0.0, 0.0, 0.0],    # Weather fluctuations
    [0.0, 0.0, 0.0, 0.0, 0.0],    # Outages
    [0.3, 0.0, 0.0, 0.0, 0.0],    # Heating demand
    [1.5, 0.0, 0.0, 0.0, 0.0],    # Wind forecast errors
    [0.0, 1.5, 0.7, 1.2, 0.0]     # Balancing costs
])

# Plotting causal graph
labels = ["weather \nfluctuations", "outages", "heating \ndemand", "wind forecast \
 ↪\nerrors", "balancing \ncosts"]
make_dot(B, labels=labels)
```

```
<graphviz.graphs.Digraph at 0x16a0c0d10>
```

Let us now generate some data from this causal graph.

```
import pandas as pd

# Set the random seed for reproducibility
np.random.seed(42)

# Number of samples
n = 1000

# Generating synthetic data based on the specified equations with uniform errors
weather_fluctuations = np.random.uniform(0, 1, size=n)
outages = np.random.uniform(0, 1, size=n)
forecast_errors = 1.5 * weather_fluctuations + np.random.uniform(0, 1, size=n)
heating = 0.3 * weather_fluctuations + np.random.uniform(0, 1, size=n)
balancing_costs = 0.7 * heating + 1.2 * forecast_errors + 1.5 * outages + np.random.
↪uniform(0, 1, size=n)

# Creating a DataFrame to hold the generated data
data = pd.DataFrame({'weather fluctuations': weather_fluctuations, 'outages': outages,
↪ 'wind forecast errors': forecast_errors,
                     'heating demand': heating, 'balancing costs': balancing_costs})
```

Now, we fit the **DirectLiNGAM** model and plot the estimated causal structure.

```
import lingam

direct_model = lingam.DirectLiNGAM()
direct_model.fit(data)
make_dot(direct_model.adjacency_matrix_, labels=labels)
```

```
<graphviz.graphs.Digraph at 0x16a01a4d0>
```

We can see how we successfully recovered the true casual structure, even though the model did not find the exact coefficients for the wind forecast errors and heating demand.

CHAPTER

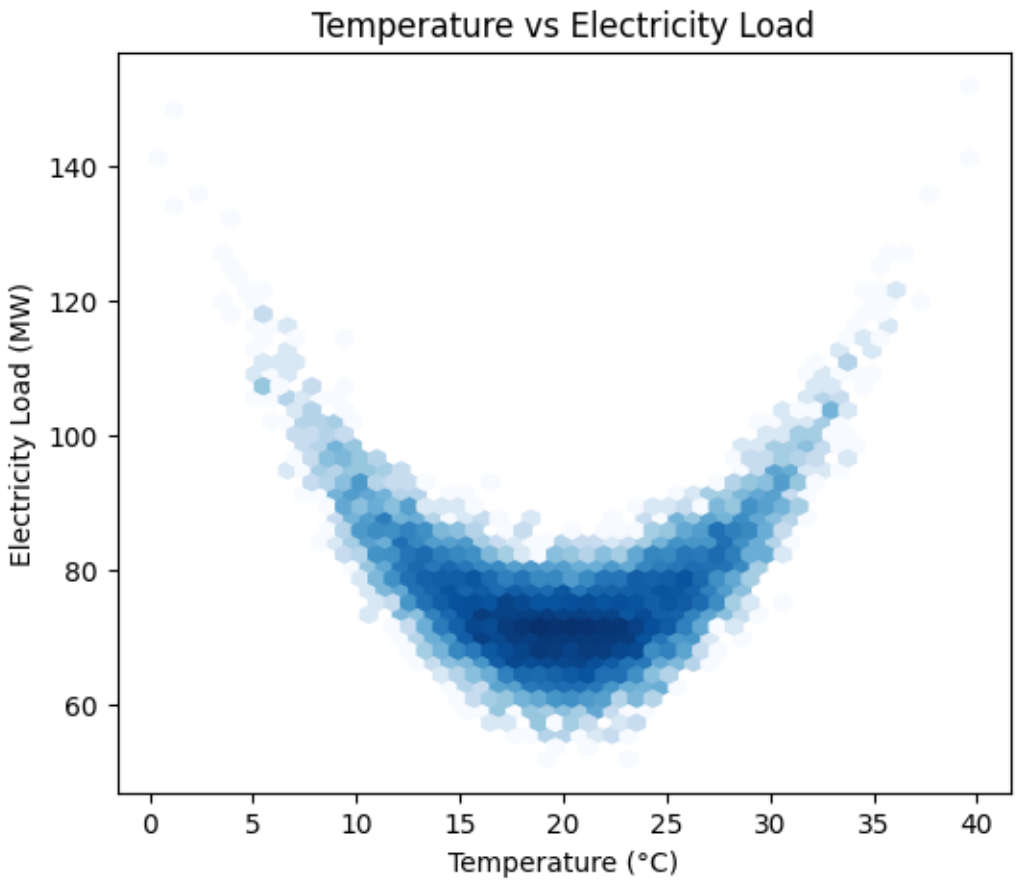
TWELVE

NONLINEAR MODELS

In many scenarios, assuming a linear relationship between observed variables may not accurately reflect the true dynamics of the system. For instance, consider the relationship between temperature and electricity load. This relationship is characteristically nonlinear, exhibiting increased electricity consumption at both colder and warmer temperatures, with consumption dipping during mild temperature days.

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

# Setting a random seed for reproducibility
np.random.seed(42)

# Generating synthetic data
n = 10000  # number of days for a year
temperatures = np.random.normal(20, 5, n)  # average temperature in Celsius
electricity_load = 0.2 * (temperatures - 20)**2 + 70 + np.random.normal(0, 5, n)  #␣
↪Quadratic relationship for U-shape

# Creating a DataFrame
data = pd.DataFrame({'Day': range(1, n+1), 'Temperature': temperatures, 'Electricity␣
↪Load': electricity_load,})

# Setting up the plots
plt.figure(figsize=(6, 5))

# Temperature vs Electricity Load with hexbin plot
hb1 = plt.hexbin(data['Temperature'], data['Electricity Load'], gridsize=50, cmap=
↪'Blues', bins='log')
plt.title('Temperature vs Electricity Load')
plt.xlabel('Temperature (°C)')
plt.ylabel('Electricity Load (MW)')
```

```
Text(0, 0.5, 'Electricity Load (MW)')
```



It turns out that **the presence of nonlinearities is beneficial for the causal discovery**. Just like non-Gaussian error distributions, nonlinear functional relations introduce asymmetries in the data distributions that can be exploited to infer the causal structure. Indeed, non-invertible functional relationships between the observed variables can provide clues to the generating causal model.

12.1 Additive noise models (ANMs)

One of the most popular approaches to causal discovery with nonlinear models is to use additive noise models (ANMs) [PMJScholkopf14]. In the case of an ANM, each observed variable x_i is obtained as a function of its parent variables, plus independent additive noise

$$x_i = f_i(\mathbf{x}_{\text{pa}(i)}) + e_i \tag{12.1}$$

where f_i is an arbitrary function (possibly different for each i), and the noise variables e_i are jointly independent and with arbitrary probability densities $p_{e_i}(e_i)$. We can easily see that when the functions f_i are linear and the $p_{e_i}(e_i)$ are non-Gaussian, we are in the standard LiNGAM settings.

12.2 Estimation

The model can be estimated using the **regression with subsequent independence test (RESIT)** approach, which follows the same intuition we saw in the DirectLiNGAM case. Indeed, we will see how the residuals in the anti-causal direction are not independent of the input variable if the function is not linear is non-Gaussian. The main steps of RESIT are:

1. Test whether x and y are statistically independent.
2. Test whether a model $y = f(x) + e$ is consistent with the data, by:
    1. Doing a **nonlinear regression** of y on x to get an estimate $\hat{f}$.
    2. Computing the residuals $\hat{e} = y - \hat{f}(x)$.
    3. Testing whether $\hat{e}$ is independent of x.
3. Test in a similar way if the **reverse model** $x = g(y) + e$ fits the data.

12.2.1 Example with nonlinear functions

Let us consider the simple example with two variables of temperature and electricity load, where

$$\text{temperature} = e_t \tag{12.2}$$

$$\text{electricity load} = 0.2 \times (\text{temperature} - 20)^2 + 70 + e_l \tag{12.3}$$

$$\tag{12.4}$$

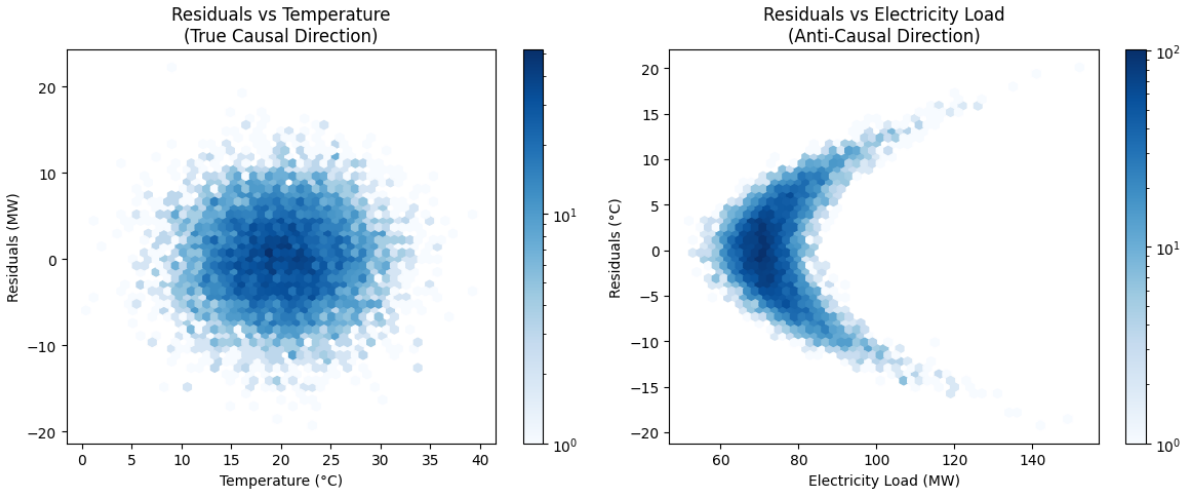
We will now try to fit a **nonlinear regression model** in the two directions and **check the residuals**.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

# Fit a quadratic model (true causal direction: Temperature -> Electricity Load)
model_true = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())
model_true.fit(X=data[['Temperature']], y=data['Electricity Load'])
electricity_pred_true = model_true.predict(X=data[['Temperature']])
residuals_true = data['Electricity Load'] - electricity_pred_true

# Fit a quadratic model (anti-causal direction: Electricity Load -> Temperature)
model_anti = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())
model_anti.fit(X=data[['Electricity Load']], y=data['Temperature'])
temperature_pred_anti = model_anti.predict(X=data[['Electricity Load']])
residuals_anti = data['Temperature'] - temperature_pred_anti

# Plotting residuals vs input variables
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

# Residuals for true causal direction
hb1 = ax[0].hexbin(data['Temperature'], residuals_true, gridsize=50, cmap='Blues',␣
↪bins='log')
cb1 = fig.colorbar(hb1, ax=ax[0])
ax[0].set_title('Residuals vs Temperature\n(True Causal Direction)')
ax[0].set_xlabel('Temperature (°C)')
ax[0].set_ylabel('Residuals (MW)')

# Residuals for true causal direction
hb1 = ax[1].hexbin(data['Electricity Load'], residuals_anti, gridsize=50, cmap='Blues
```

(continues on next page)

(continued from previous page)

```
↪', bins='log')
cb1 = fig.colorbar(hb1, ax=ax[1])
ax[1].set_title('Residuals vs Electricity Load\n(Anti-Causal Direction)')
ax[1].set_xlabel('Electricity Load (MW)')
ax[1].set_ylabel('Residuals (°C)')

plt.tight_layout()
```



12.2.2 Full RESIT example

We now try to apply the RESIT approach to a slightly more complex example, where some of the relationships are **nonlinear**.

Let us assume the following **structural equations**:

$$\text{temperature} = e_t \tag{12.5}$$

$$\text{electricity load} = 0.2 \times (\text{temperature} - 20)^2 + 70 + e_l \tag{12.6}$$

$$\text{thermal constraints} = 1.5 \times \text{electricity load} + \sin(\text{temperature}) + e_c \tag{12.7}$$

$$\tag{12.8}$$

where the errors e_i are now assumed to **Gaussian**.

The **causal graph** is then given by

```
from lingam.utils import make_dot

# Matrix of coefficients (weights)
B = np.array([
    [0.0, 0.0, 0.0],    # Temperatures
    [1.0, 0.0, 0.0],    # Electricity load
    [1.0, 1.0, 0.0]     # Thermal constraints
])

# Plotting causal graph
make_dot(B, labels=["temperatures", "electricity load", "thermal constraints"])
```

```
<graphviz.graphs.Digraph at 0x166403ad0>
```

where the coefficient of 1 is only an indication of causal relation (and its direction) between two variables

Let us now generate some observations according to that causal graph.

```
# Setting random seed for reproducibility
np.random.seed(42)

# Generating synthetic data
n = 1000
temperatures = np.random.normal(20, 5, n)  # average temperature in Celsius
electricity_load = 0.2 * (temperatures - 20)**2 + 70 + np.random.normal(0, 5, n)  #␣
↪Quadratic relationship for U-shape
thermal_constraints = 1.5* electricity_load + np.sin(temperatures) + np.random.
↪normal(0, 1, n)

# Creating DataFrame
data = pd.DataFrame({
    'temperatures': temperatures,
    'electricity load': electricity_load,
    'thermal constraints': thermal_constraints
})
```

We can now try to estimate the causal structure with the **RESIT** approach.

The interesting part is that we can **use any nonlinear regression model**. For example, we can try to use RESIT with a **random forest** regression model.

```
import lingam
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(n_estimators=100, max_depth=2, random_state=42)
model = lingam.RESIT(regressor=rf)
model.fit(data)
make_dot(model.adjacency_matrix_, labels=list(data.columns))
```

```
<graphviz.graphs.Digraph at 0x165e931d0>
```

Bootstrapping

While our initial results approached the true causal graph, there is always inherent **uncertainty** in any estimation process derived from a single dataset. To enhance the robustness and reliability of our causal estimates, we can employ bootstrapping, a powerful statistical **resampling** method.

Bootstrapping involves repeatedly sampling from the original data set with replacement to create multiple "bootstrap" samples. By applying the causal estimation process to each of these samples, we can observe how our estimates vary across different versions of the data. This approach not only helps in assessing the stability of our causal inference but also in improving the accuracy of our estimates by averaging the results.

So, we can estimate multiple DAGs, for each bootstrap replica of our dataset, and get a **probability score associated to each causal connection**.

```
from lingam.utils import print_causal_directions, print_dagc

replications = 10
result = model.bootstrap(data, n_sampling=replications)
cdc = result.get_causal_direction_counts(n_directions=8, min_causal_effect=0.1, split_
 ↪by_causal_effect_sign=True)
print_causal_directions(cdc, replications, labels=list(data.columns))
```

```
electricity load <--- temperatures (b>0) (100.0%)
thermal constraints <--- temperatures (b>0) (100.0%)
electricity load <--- thermal constraints (b>0) (50.0%)
thermal constraints <--- electricity load (b>0) (50.0%)
```

```
dagc = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.1,␣
 ↪split_by_causal_effect_sign=True)
print_dagc(dagc, replications, labels=list(data.columns))
```

```
DAG[0]: 50.0%
        electricity load <--- temperatures (b>0)
        thermal constraints <--- temperatures (b>0)
        thermal constraints <--- electricity load (b>0)
DAG[1]: 50.0%
        electricity load <--- temperatures (b>0)
        electricity load <--- thermal constraints (b>0)
        thermal constraints <--- temperatures (b>0)
```

CHAPTER

THIRTEEN

TIME SERIES MODELS

In many cases, the **temporal dimension** introduces complexities that traditional causal models may not adequately address. Particularly in fields like economics, environmental science, and engineering, understanding how variables influence each other both **instantaneously and across time lags** is crucial for predicting future states and designing effective interventions.

To capture both the immediate and delayed interactions among multiple time-dependent variables, it is important to employ models that can account for these dynamics. We will now introduce an extension of the LiNGAM model, based on **vector autoregression (VAR)** models, which has been introduced in [HyvarinenSH08] and [HyvarinenZSH10].

13.1 VARLiNGAM model

Assume that at each time step t we observe a vector $\mathbf{x}(t)$ with p variables

$$\mathbf{x}(t) = [\mathbf{x}_1(t), \mathbf{x}_2(t), \dots, \mathbf{x}_p(t)]^\top \tag{13.1}$$

Then, we use the following structural causal model, which considers both the instantaneous ($\tau = 0$) and lagged ($\tau \geq 1$) causal effects

$$\mathbf{x}(t) = \sum_{\tau=0}^{k} \mathbf{B}_\tau \mathbf{x}(t-\tau) + \mathbf{e}(t) \tag{13.2}$$

where:

- $\mathbf{B}$ is the coefficient matrix representing the magnitudes of the contemporaneous ($\mathbf{B}_0$) and lagged ($\mathbf{B}_\tau$, $\tau = 1, \dots, h$) causal effects.
- $\mathbf{e}(t)$ are the non-Gaussian mutually independent (both of each other and over time) errors.

A sufficient condition for the coefficient matrices $\mathbf{B}_0$ and $\mathbf{B}_\tau$ to be identifiable is that the contemporaneous causal relations represented by the matrix $\mathbf{B}_0$ is acyclic and the error variables $e_i(t)$, with $i = 1, \dots, p$ and $t = 1, \dots, T$, are non-Gaussian and independent. This independence implies **no hidden common cause** between the observed variables at the same point in time or between the observed variables at different points in time.

For example, if we consider the following variables:

- **Solar generation** ($x_1(t)$): amount of electricity generated from solar power at time t.
- **Transmission capacity usage** ($x_2(t)$): percentage of transmission capacity utilized at time t.
- **Balancing costs** ($x_3(t)$): costs incurred to ensure grid stability and balance the electricity supply with demand at time t.

At time t, we will observe the vector given by

$$\mathbf{x}(t) = \begin{bmatrix} \text{solar generation}(t) \\ \text{transmission capacity usage}(t) \\ \text{balancing costs}(t) \end{bmatrix} \tag{13.3}$$

and we can then model the time series data with the following relationships

$$\mathbf{x}(t) = \mathbf{B}_0\mathbf{x}(t) + \mathbf{B}_1\mathbf{x}(t-1) + \mathbf{e}(t) \tag{13.4}$$

where $\mathbf{B}_0$ is the matrix of the **contemporaneous effects**, given by

$$\begin{bmatrix} 0 & 0 & 0 \\ -1.0 & 0 & 0 \\ 1.5 & 0.2 & 0 \end{bmatrix} \tag{13.5}$$

implying that current solar generation negatively affects the transmission capacity usage due to intermittency issues. Increased solar output at the same time increases the need for grid balancing due to the variable nature of solar power, which can affect grid stability.

Instead, $\mathbf{B}_1$ is the matrix of the **lagged effects**, given by

$$\begin{bmatrix} 0 & 0 & 0 \\ 0.0 & 0 & 0 \\ 0.5 & 1.5 & 0 \end{bmatrix} \tag{13.6}$$

capturing how solar generation and transmission capacity usage from the previous time step $(t-1)$ influence current balancing costs. It suggests that previous fluctuations in solar output and transmission constraints can lead to increased current balancing costs.

Finally, $\mathbf{e}(t)$ represent the non-Gaussian error terms representing other unmodeled influences.

13.2 Estimation

One possible estimation method is to combine a traditional least-squares estimation of an autoregressive (AR) model with the LiNGAM estimation. The key intuition is that the model shown in the introduction is essentially **a LiNGAM model for the residuals of the predictions made by a traditional VAR model** that only considers the lagged effects and not the contemporaneous ones (where $\tau > 0$). The estimation method is based on three steps:

1. Obtain a least-squares estimate $\widehat{\mathbf{M}}_\tau$ of the AR model given by

$$\mathbf{x}(t) = \sum_{\tau=1}^{k} \mathbf{M}_\tau \mathbf{x}(t-\tau) + \mathbf{n}(t) \tag{13.7}$$

   $\widehat{\mathbf{M}}_\tau$ can be thought of as a matrix of the preliminary guesses for the lagged effects.

2. Compute the residuals of the AR model, which are an estimate of the true errors $\mathbf{n}(t)$

$$\hat{\mathbf{n}}(t) = \mathbf{x}(t) - \sum_{\tau=1}^{k} \widehat{\mathbf{M}}_\tau \mathbf{x}(t-\tau) \tag{13.8}$$

   These residuals are the differences between the actual observed data and the predictions from the AR model. They should contain the influences not captured by the past values (potentially the instantaneous effects we are interested in).

3. Perform a LiNGAM analysis on the residuals, which returns an **estimate of the coefficient matrix for the instantaneous causal effects $\mathbf{B}_0$**

$$\hat{\mathbf{n}}(t) = \mathbf{B}_0\hat{\mathbf{n}}(t) + \mathbf{e}(t) \tag{13.9}$$

   The idea is that if there are instantaneous effects, they would show up in the residuals of the AR model, because the AR model did not account for them.

4. Once $\mathbf{B}_0$ has been estimated, the next step is to disentangle the lagged causal effects from the total observed effects captured in the AR model. So, we compute the **estimates of the coefficient matrices for the lagged causal effects**, $\mathbf{B}_\tau$ for $\tau > 0$

$$\widehat{\mathbf{B}_\tau} = (\mathbf{I} - \widehat{\mathbf{B}}_0)\widehat{\mathbf{M}}_\tau \tag{13.10}$$

13.3 Full VARLiNGAM example

We will no generate observations of the variables introduced before: **solar generation**, **transmission capacity usage**, and **balancing costs**.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import make_dot, print_causal_directions, print_dagc
import warnings
warnings.filterwarnings('ignore', category=UserWarning)
import matplotlib.pyplot as plt
%matplotlib inline
```

The matrices explaining the contemporaneous and lagged effects are given by

```
B0 = [[0.0, 0.0, 0.0],
      [-1.0, 0.0, 0.0],
      [1.5, 2.0, 0.0]]

B1 = [[0.0, 0.0, 0.0],
      [0.0, 0.0, 0.0],
      [0.5, 1.5,0.0]]

causal_order = [0, 1, 2]
```

13.3.1 Contemporaneous effects

As we can see, at time t balancing costs are influenced by the current usage of the transmission capacity, and by the current solar generation. We can also see how the transmission capacity is negatively impacted by the solar power generation.

```
labels_0 = ["solar power \ngeneration (t)", "transmission capacity \nusage (t)",
 ↪"balancing \ncosts (t)"]
make_dot(B0, labels=labels_0)
```

```
<graphviz.graphs.Digraph at 0x13caa5390>
```

13.4 Lagged effects

Given the time-dependent nature of the data, balancing costs at time t are also dependent upon the solar generation and spare capacity observed at the previous time step. However, there are no lagged causal associations between solar generation and transmission capacity.

```
labels_1 = ["solar power \ngeneration (t-1)", "transmission capacity \nusage (t-1)",
 ↪"balancing \ncosts (t)"]
make_dot(B1, labels=labels_1)
```

```
<graphviz.graphs.Digraph at 0x13e718c10>
```

13.5 Putting all together

We can now see the complete causal structure of the data-generating process, including both contemporaneous and lagged effects.

```
labels = ["solar power \ngeneration (t)", "transmission capacity \nusage (t)",
 ↪"balancing \ncosts (t)",
         "solar power \ngeneration (t-1)", "transmission capacity \nusage (t-1)",
 ↪"balancing \ncosts (t-1)"]
make_dot(np.hstack((B0, B1)), ignore_shape=True, lower_limit=0.05, labels=labels)
```

```
<graphviz.graphs.Digraph at 0x13ddaa3d0>
```

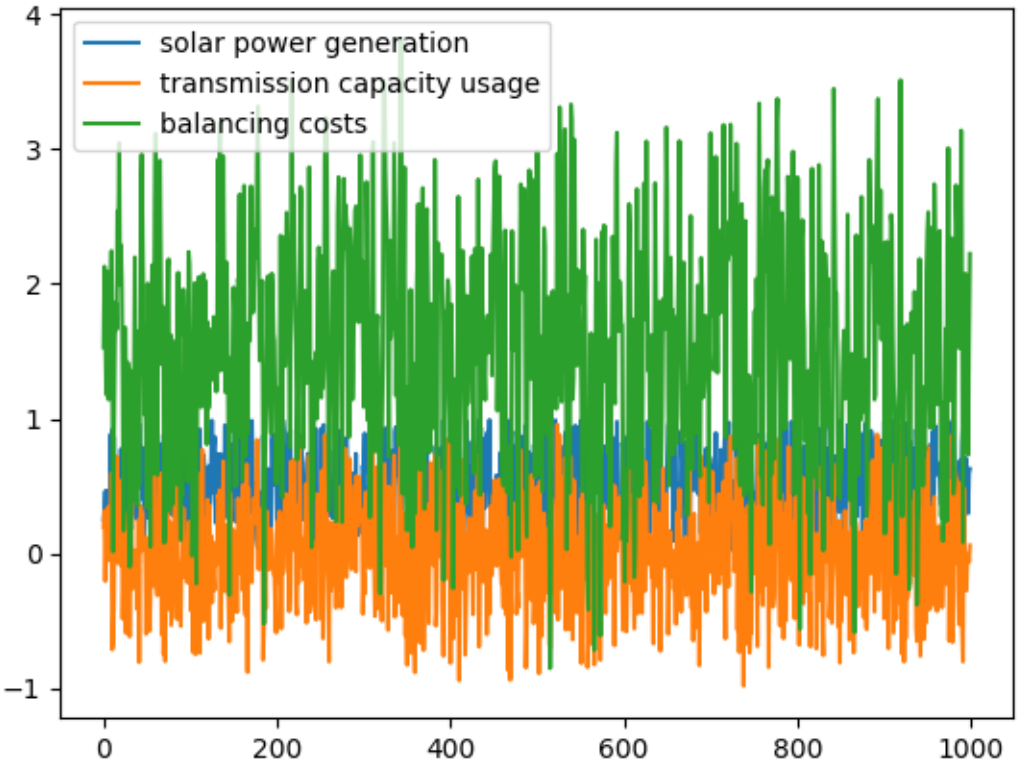
We will now **generate some observations** using the previous causal graph, and then try to retrieve the true causal structure using **VARLiNGAM**.

```
# Generating data according to this causal graph
B0 = np.array(B0)
B1 = np.array(B1)

# Sample size
n_features = len(causal_order)
sample_size = 1000

# Initialize the data matrix with zeros (considering t and t-1)
data = np.zeros((sample_size + 1, n_features))  # +1 to accommodate initial values at␣
 ↪t=-1

# Add non-Gaussian noise for each variable and each time point
for i in range(n_features):
    data[:, i] += np.random.uniform(size=sample_size + 1)

# Generate data according to true causal graph
for t in range(1, sample_size + 1):
    for var in causal_order:
        data[t, var] += np.dot(B0[var, :], data[t, :]) + np.dot(B1[var, :], data[t-1,␣
 ↪:])

# Drop the initial row used for t=-1 values
data = data[1:, :]

# Convert to DataFrame for easier handling
cols = ["solar power generation", "transmission capacity usage", "balancing costs"]
X = pd.DataFrame(data, columns=cols)
```

We can now plot the generated time series

```
for i in range(X.shape[1]):
    plt.plot(X.index, X.iloc[:, i], label=cols[i])
plt.legend()
```

```
<matplotlib.legend.Legend at 0x13f5d4890>
```



13.5.1 Estimating contemporaneous effects

This process is relatively simple, and includes:

1. **Fitting a VAR model** to the time series, to get rid of autocorrelation and lagged effects.
2. Applying traditional **LiNGAM model to the residuals**.

This allows us to estimate the matrix of contemporaneous effects, B_0.

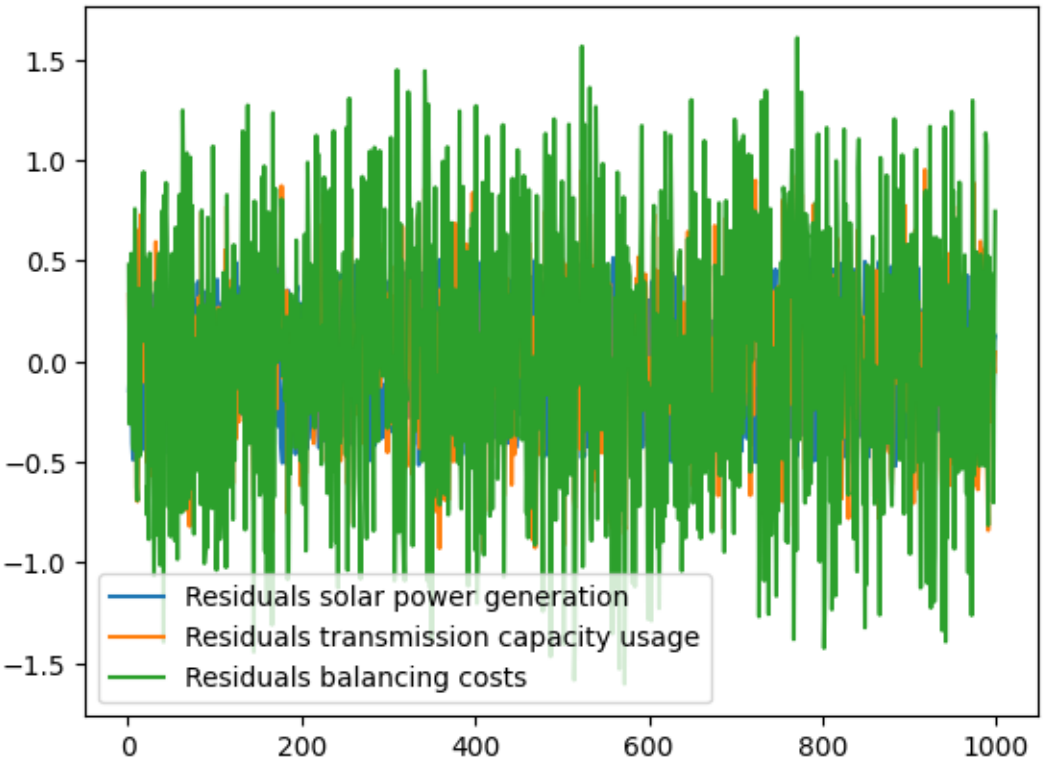
Here we fit a VAR model

```
from statsmodels.tsa.vector_ar.var_model import VAR

model = VAR(X)
results = model.fit(1)  # Fit VAR(1)
residuals = results.resid
```

let's plot the residuals

```
for i in range(residuals.shape[1]):
    plt.plot(residuals.index, residuals.iloc[:, i], label=f'Residuals {cols[i]}')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x13feb9610>
```



Now, we can simply apply LiNGAM (as we did in the initial chapter) to the extracted residuals.

```
direct_model = lingam.DirectLiNGAM()
direct_model.fit(residuals)
make_dot(direct_model.adjacency_matrix_, labels=labels_0)
```

```
<graphviz.graphs.Digraph at 0x13e132c50>
```

We can see how we got extremely close to the true contemporaneous effects.

13.5.2 Estimating lagged effects

We can now estimate the matrix of lagged effects, B_1. The process (included in the VARLiNGAM method) involves correcting the matrix estimated by the VAR model to take into account the effects already considered in B_0.

```
model = lingam.VARLiNGAM(lags=1)
model.fit(X)
make_dot(model.adjacency_matrices_[1], labels=labels_1) # B1
```

```
<graphviz.graphs.Digraph at 0x13fec5290>
```

Unfortunately, this time the causal graph is not precisely equal to the true one for the lagged effects. However, the largest lagged effects (transmission capacity and solar generation on balancing costs) have been properly identified, with the correct magnitude and sign.

13.5.3 Complete causal graph

We can now create the complete causal structure by simply stacking the two coefficient matrices B_0 and B_1.

```
# Combined estimated graph
make_dot(np.hstack([direct_model.adjacency_matrix_, model.adjacency_matrices_[1]]),␣
↪ignore_shape=True, lower_limit=0.05, labels=labels)
```

```
<graphviz.graphs.Digraph at 0x13caf83d0>
```

CHAPTER

FOURTEEN

STRUCTURAL BREAKS

Another challenges related to time series analysis and causal inference, besides the presence of lagged effects, is the possibility of the system to be in **multiple states or regimes**. This is a problem well known to economists, since financial data often exhibits distinct behaviour depending on the specific state of the system (e.g., growth or recession).

In electricity markets, we can also witness something similar. In particular, in the **balancing market**, the system is constantly switching between two states [BIM20].:

- **Short**, when there is a shortage of energy.
- **Long**, when there is an excess of energy.

Here, we will just provide an example of the challenges of trying to infer the causal structure of the system, if there is a **concept drift** affecting the system. In this case, we refer to concept drift as an alteration of the data-generating process, where the coefficients relating different variables change over time. For example:

- A **market policy** might significantly affect the behaviour or market partecipants.
- Newly introduced **subsidies** might alter the cost structures of energy generators.
- Advances in **battery technology** might introduce new variables.

14.1 Scenario with two regimes

Let's consider a scenario in the electricity market where three variables are impacted by a significant **policy change**, such as the introduction of a new subsidy for renewable energy. This policy change affects the relationships among these variables, illustrating how concept drift can occur in response to external changes.

14.1.1 Pre-subsidy structural model:

Before the introduction of subsidies, the causal relationships might be modeled as follows:

$$\mathbf{x}(t) = \mathbf{B}_{\text{pre}}\mathbf{x}(t-1) + \mathbf{e}(t) \tag{14.1}$$

where $\mathbf{B}_{\text{pre}}$ is:

$$\mathbf{B}_{\text{pre}} = \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ -1 & 0.05 & 0 \end{bmatrix} \tag{14.2}$$

14.1.2 Post-subsidy structural model:

After the introduction of subsidies, the relationships change as follows:

$$\mathbf{x}(t) = \mathbf{B}_{\text{post}}\mathbf{x}(t-1) + \mathbf{e}(t) \tag{14.3}$$

where $\mathbf{B}_{\text{post}}$ is:

$$\mathbf{B}_{\text{post}} = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ 5 & 0 & 0 \end{bmatrix} \tag{14.4}$$

This change in coefficients represents a concept drift, where the underlying data-generating process has altered due to external policy intervention. The challenge in causal inference here is to accurately detect and adapt to these changes, ensuring that models remain valid over time despite the evolving relationships. Failure to account for such drifts can lead to inaccurate predictions and misguided policy or business decisions.

14.1.3 Contemporaneous effects

In **regime 1** the causal graph is represnted by

```
from lingam.utils import make_dot

B0_regime1 = [[0.0, 0.0, 0.0],
              [2.0, 0.0, 0.0],
              [-1.0, 0.5,0.0]]

make_dot(B0_regime1)
```

```
<graphviz.graphs.Digraph at 0x1615d51d0>
```

Instead, in **regime 2** we have

```
B0_regime2 = [[0.0, 0.0, 0.0],
              [-1.0, 0.0, 0.0],
              [5.0, 0.0,0.0]]

make_dot(B0_regime2)
```

```
<graphviz.graphs.Digraph at 0x162fba090>
```

(You might have now recognised this as a simple **fork**)

14.1.4 Lagged effects

To slightly render the causal discovery problem more difficult, we also introduced some autocorrelation, just like in the **VARLiNGAM** example we saw in the previous chapter.

This simply means we now also have two $\mathbf{B}_1$ matrices, one for each regime.

```
B1_regime1 = [[0.7, 0.0, 0.0],
              [0.2, -0.6, 0.0],
              [0.1, 0.0,0.1]]
```

(continues on next page)

(continued from previous page)

```
B1_regime2 = [[0.0, 0.0, 0.0],
              [0.2, 0.2, 0.0],
              [0.5, 0.0,0.1]]

causal_order = [0, 1, 2]
```

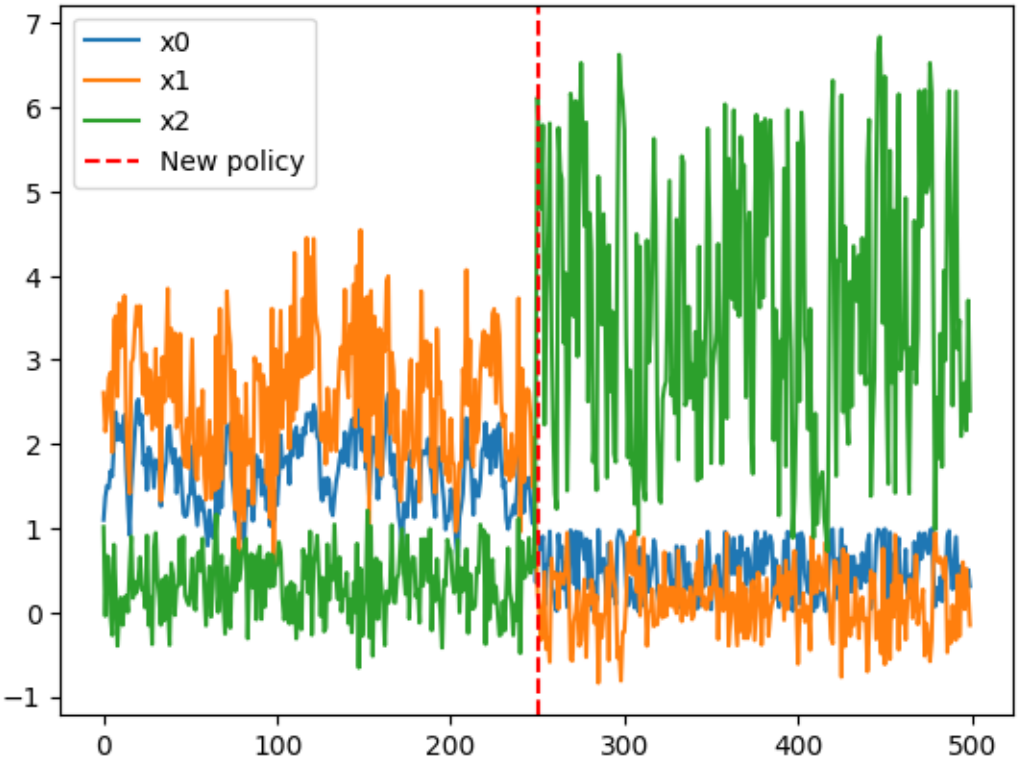
We now **generate some observations** using this four matrices

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

# Sample size
np.random.seed(0)
n_features = len(causal_order)
sample_size = 500
switch_point = 250  # Midpoint switch

# Coefficients for two different regimes
B0_regime1 = np.array(B0_regime1)
B0_regime2 = np.array(B0_regime2)
B1_regime1 = np.array(B1_regime1)
B1_regime2 = np.array(B1_regime2)

# Initialize the data matrix with zeros (considering t and t-1)
data = np.zeros((sample_size + 1, n_features))  # +1 to accommodate initial values at␣
↪t=-1

# Data matrix to accommodate initial values at t=-1
for i in range(n_features):
    data[:, i] += np.random.uniform(size=sample_size + 1)

# Generate data according to VARLiNGAM model with switching regimes
for t in range(1, sample_size + 1):
    if t < switch_point:
        for var in causal_order:
            data[t, var] += np.dot(B0_regime1[var, :], data[t, :]) + np.dot(B1_
↪regime1[var, :], data[t-1, :])
    else:
        for var in causal_order:
            data[t, var] += np.dot(B0_regime2[var, :], data[t, :]) + np.dot(B1_
↪regime2[var, :], data[t-1, :])

# Drop the initial row used for t=-1 values
data = data[1:, :]

# Convert to DataFrame for easier handling
X = pd.DataFrame(data, columns=[f"x{i}" for i in range(n_features)])
```

Let's now plot the data to see what is happening

```
for i in range(X.shape[1]):
    plt.plot(X.index, X.iloc[:, i], label=f'x{i}')
plt.axvline(x=switch_point, color='r', linestyle='--', label='New policy')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x163dbdf50>
```



We can clearly see how the introduction of the new policy altered the time series.

14.2 Estimation

To show the difficulty of doing causal discovery using data collected from drifting data streams, we will show what happens in two cases:

1. We fit just one model to the whole dataset.
2. We fit two separate models, one for each regime.

14.2.1 Fitting one model to all the data

As we did before, we fit a VAR model, compute the residuals, and apply LiNGAM to its residuals.

```
from statsmodels.tsa.vector_ar.var_model import VAR
import lingam

# Fit one VAR model
var_model = VAR(X)
var_results = var_model.fit(1)
residuals = var_results.resid

# Apply LiNGAM to the residuals
```

(continues on next page)

(continued from previous page)

```
lingam_model = lingam.DirectLiNGAM()
lingam_model.fit(residuals)
make_dot(lingam_model.adjacency_matrix_)  # B0
```

```
<graphviz.graphs.Digraph at 0x108df9fd0>
```

We can see how we got a **wrong estimation**. The causal structure is like the first regime, but the coefficients are wrong.

14.2.2 Fitting two separate models

We now show the ideal situation, where we assume we were able to **discern the two regimes** and fit distinct models.

```
# Split the data into two regimes
x_regime1 = X.iloc[:switch_point, :]
x_regime2 = X.iloc[switch_point:, :]

# Fit VAR model to each regime
var_model1 = VAR(x_regime1)
var_results1 = var_model1.fit(1)  # VAR(1) for simplicity
residuals_regime1 = var_results1.resid

var_model2 = VAR(x_regime2)
var_results2 = var_model2.fit(1)  # VAR(1) for simplicity
residuals_regime2 = var_results2.resid

# Apply LiNGAM for each regime
lingam_model1 = lingam.DirectLiNGAM()
lingam_model1.fit(residuals_regime1)

lingam_model2 = lingam.DirectLiNGAM()
lingam_model2.fit(residuals_regime2)
```

```
<lingam.direct_lingam.DirectLiNGAM at 0x164221890>
```

For the first regime, we now have

```
make_dot(lingam_model1.adjacency_matrix_)  # B0 regime 1
```

```
<graphviz.graphs.Digraph at 0x163dbc190>
```

and for the second regime

```
make_dot(lingam_model2.adjacency_matrix_) # B0 regime 2
```

```
<graphviz.graphs.Digraph at 0x164210e10>
```

We can clearly see how, taking into account the **policy change** we have been able to identify the **true causal structures**.

Part V

III. Causal Inference

CHAPTER

FIFTEEN

OVERVIEW

Causal inference focuses on estimating the strength and nature of identified causal relationships. This part uses statistical techniques to quantify the effect of one variable on another, given that the causal structure is known or assumed. If you do not know the causal graph, you can take a look at Part II about Causal Discovery to see methods to unveil the causal graph from observational data.

Knowing the causal graph is crucial because it helps identify the relationships and dependencies between variables. This knowledge allows for accurate estimation of causal effects, essential for making informed decisions and predictions. Here are some key reasons why understanding the causal graph is important:

- **Correct identification of relationships**: the causal graph helps in correctly identifying which variables directly affect the outcome of interest. This is important for estimating the true causal effect of a treatment variable.
- **Avoiding Bias**: without a proper causal graph, estimates of causal effects can be biased due to confounding variables. For instance, failing to account for confounders can lead to incorrect conclusions about the relationship between variables.
- **Understanding mediators and colliders**: the causal graph helps in identifying mediators (variables that lie on the causal path between the treatment and outcome) and colliders (variables influenced by two or more other variables). Misunderstanding these roles can lead to incorrect model specifications and biased estimates.

The following Chapters will try to present some key techniques to obtain precise and reliable estimates of causal effects by accounting for the presence of confounders and exogenous factors.

15.1 Content of Causal Inference chapters

Chapter	Description
Instrumental Variables	How to estimate causal effects when we might suffer from omitted variable bias, or when the explanatory variable and response variable influence each other.
Propensity Score Matching	How to control for confounders and avoid the selection bias.
Double Machine Learning	How to estimate causal inference using machine learnign models and adjusting for the presence of a high-dimensional set of confounders.
Difference-in-Differences	How to estimate the effect of a treatment on time series data by comparing changes in outcomes over time between treated and control groups.
Interrupted Time Series	How to estimate treatment effects in time series without the presence of a control group.

CHAPTER

SIXTEEN

INSTRUMENTAL VARIABLES

Instrumental variables (IV) refer to a method used in econometrics to estimate causal relationships when controlled experiments are not feasible and an explanatory variable is correlated with the error term, leading to **endogeneity issues**. Saying that a variable is correlated with the error term means that the variable and the error term share some common influence or that the variable captures some part of the variability in the dependent variable that should be attributed to the error term. When we say a variable is correlated with the error term, it means that both the variable and the error term are influenced by some common external factors. In other words, the variable captures some variability in the outcome that should be attributed to these external influences, which introduces bias in our estimates. This happens because the variable not only shows its direct effect on the outcome but also the effect of those omitted external factors.

16.1 Endogeneity

Endogeneity can occur due to:

1. **Omitted variable bias**: unobserved variables that affect both the explanatory variable and the outcome.
2. **Simultaneity**: when the explanatory variable and the outcome influence each other
3. **Measurement error**: Inaccuracies in measuring the explanatory variable.

For instance, consider trying to understand how wind power production impacts wholesale electricity prices. However, if wind power production and prices are also affected by factors like solar power or broader economic trends, these external influences can bias our estimates. Here, we can use an IV, like wind speed forecasts, which affects wind power production but doesn't directly influence electricity prices except through its effect on wind power. This helps us isolate the true effect of wind power production on prices, giving a clearer picture of the causal relationship.

An IV has two main properties:

1. **Relevance**: the IV needs to be something that is related to the variable you're interested in studying.
2. **Exogeneity**: the IV should not be influenced by other factors that affect the outcome you're studying.

Here is a representation of a case where a candidate IV (wind speed forecast) can help us isolate the effect of an explanatory variable (wind power production), on the response variable (price).

```
import graphviz
from IPython.display import display

# Create a new graph
dot = graphviz.Digraph()

# Add nodes
dot.node('Z', 'Wind speed\nforecasts')
dot.node('X', 'Wind power\nproduction')
```

(continues on next page)

(continued from previous page)

```
dot.node('Y', 'Electricity \nprices')
dot.node('U', 'Unobserved\nconfounders')

# Add edges
dot.edge('Z', 'X')
dot.edge('X', 'Y')
dot.edge('U', 'X')
dot.edge('U', 'Y')

# Display the graph in the notebook
display(dot)
```

```
<graphviz.graphs.Digraph at 0x1043bc9d0>
```

16.2 Two-stage least squares (2SLS)

Two-stage least squares (2SLS) is an econometric approach used to address endogeneity using IVs. It is performed in two steps:

1. Regress the endogenous explanatory variable on the IV to obtain predicted values.
2. Regress the dependent variable on the predicted values obtained from the first stage.

The key idea is that if we use the IV (wind speed forecast) to predict the endogenous explanatory variable (wind power production), and then use these predicted values to estimate its effect on the response variable (price), we essentially use only the part of the variation in the explanatori variable that is "clean" of the confounding effects of the unobserved confounders (e.g., solar generation, macroeconomic trends). This allows us to estimate a causal effect that is not biased.

If we want to visualize this effect, this corresponds to removing the edge from the unobserved confounders to the endogenous explaantory variable.

```
# Create a new graph
dot = graphviz.Digraph()

# Add nodes
dot.node('Z', 'Wind speed\nforecasts')
dot.node('X', 'Predicted wind \npower production \nusing the IV')
dot.node('Y', 'Electricity \nprices')
dot.node('U', 'Unobserved\nconfounders')

# Add edges
dot.edge('Z', 'X')
dot.edge('X', 'Y')
dot.edge('U', 'Y')

# Display the graph in the notebook
display(dot)
```

```
<graphviz.graphs.Digraph at 0x104411cd0>
```

Let's now generate some data to show how this works in practice. Since the confounders are, by definition, **unobserved**, we assume to only have at our disposal the wind speed forecast, the wind power production, and the electricity prices.

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)

# Simulate data
n = 1000
true_effect = 2  # True causal effect of wind power production on electricity prices
wind_speed_forecasts = np.random.uniform(0, 10, n)  # IV
unobserved_confounders = np.random.normal(0, np.sqrt(10), n)  # Unobserved factors␣
↪with variance 10
wind_power_production = 2 * wind_speed_forecasts + unobserved_confounders + np.random.
↪normal(0, 1, n)
electricity_prices = true_effect * wind_power_production + 1.5 * unobserved_
↪confounders + np.random.normal(0, 1, n)

# Create DataFrame
data = pd.DataFrame({
    'WindSpeedForecasts': wind_speed_forecasts,
    'WindPowerProduction': wind_power_production,
    'ElectricityPrices': electricity_prices
})

data.head()
```

	WindSpeedForecasts	WindPowerProduction	ElectricityPrices
0	3.745401	6.646425	15.010277
1	9.507143	14.708451	22.433042
2	7.319939	14.337450	29.275135
3	5.986585	14.664067	31.182353
4	1.560186	4.973025	12.114167

16.2.1 Ignoring confounders

If we simply ignore the problem of having unobserved confounders, we might be tempted to simply fit a model on the available data. Let's try that.

```
# Perform OLS regression
X_ols = sm.add_constant(data['WindPowerProduction'])
ols_model = sm.OLS(data['ElectricityPrices'], X_ols).fit()
print("OLS Regression Results:")
print(ols_model.summary())
```

```
OLS Regression Results:
                            OLS Regression Results
==============================================================================
Dep. Variable:      ElectricityPrices   R-squared:                       0.925
Model:                            OLS   Adj. R-squared:                  0.925
Method:                 Least Squares   F-statistic:                 1.225e+04
Date:                Sun, 07 Jul 2024   Prob (F-statistic):               0.00
```

(continues on next page)

(continued from previous page)

```
Time:                        20:03:36   Log-Likelihood:                -2886.7
No. Observations:                1000   AIC:                             5777.
Df Residuals:                     998   BIC:                             5787.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================================
                             coef    std err          t      P>|t|      [0.025      0.
↪975]
----------------------------------------------------------------------------------------------
↪----
const                     -2.5490      0.251    -10.149      0.000      -3.042      -
↪2.056
WindPowerProduction        2.2967      0.021    110.677      0.000       2.256       
↪2.337
==============================================================================
Omnibus:                        0.663   Durbin-Watson:                   1.919
Prob(Omnibus):                  0.718   Jarque-Bera (JB):                0.609
Skew:                          -0.059   Prob(JB):                        0.737
Kurtosis:                       3.022   Cond. No.                         22.2
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
↪specified.
```

16.2.2 2SLS

Now, we use the 2SLS approach and fit two models, the first one to predit the explanatory variable and the second one using the predicted variable to predict the response.

```
# First stage of 2SLS: Regress WindPowerProduction on WindSpeedForecasts
X_first_stage = sm.add_constant(data['WindSpeedForecasts'])
first_stage_model = sm.OLS(data['WindPowerProduction'], X_first_stage).fit()
data['FittedWindPowerProduction'] = first_stage_model.fittedvalues

# Second stage of 2SLS: Regress ElectricityPrices on the fitted values from the first
↪stage
X_second_stage = sm.add_constant(data['FittedWindPowerProduction'])
second_stage_model = sm.OLS(data['ElectricityPrices'], X_second_stage).fit()
print("\n2SLS Regression Results:")
print(second_stage_model.summary())
```

```
2SLS Regression Results:
                            OLS Regression Results
==============================================================================
Dep. Variable:      ElectricityPrices   R-squared:                       0.509
Model:                            OLS   Adj. R-squared:                  0.508
Method:                 Least Squares   F-statistic:                     1034.
Date:                Sun, 07 Jul 2024   Prob (F-statistic):          3.21e-156
Time:                        20:03:36   Log-Likelihood:                -3824.2
No. Observations:                1000   AIC:                             7652.
Df Residuals:                     998   BIC:                             7662.
Df Model:                           1
```

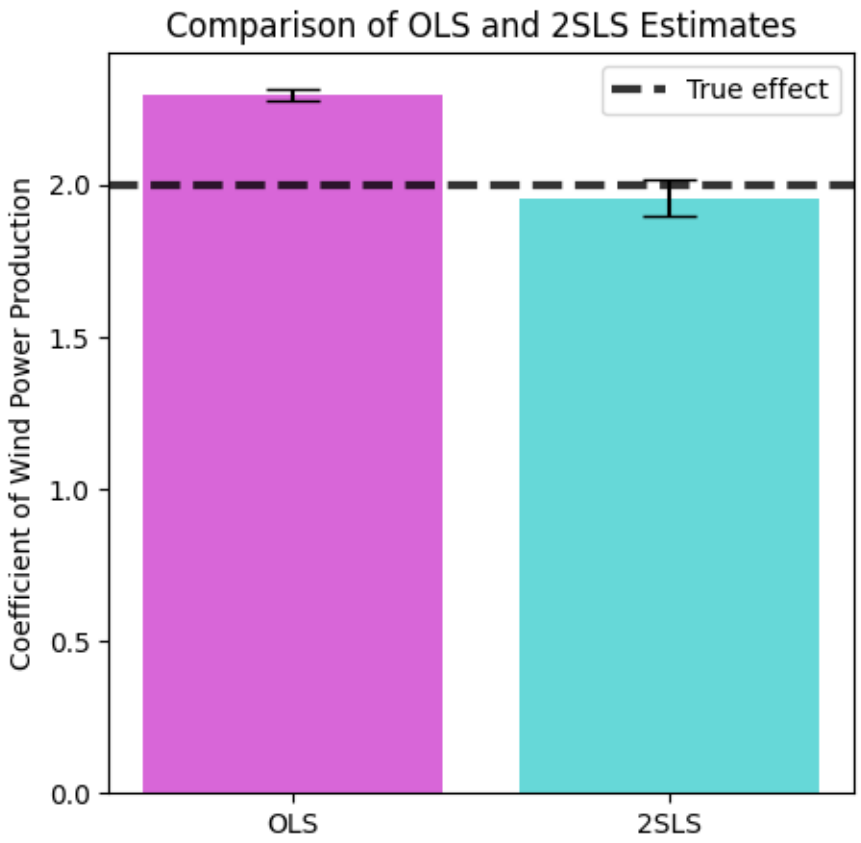
(continues on next page)

(continued from previous page)

```
Covariance Type:            nonrobust
==============================================================================================
                                 coef    std err          t      P>|t|      [0.025 ␣
 ↪    0.975]
----------------------------------------------------------------------------------------------
 ↪----------
const                          0.9002      0.709      1.269      0.205      -0.492 ␣
 ↪     2.292
FittedWindPowerProduction      1.9563      0.061     32.149      0.000       1.837 ␣
 ↪     2.076
==============================================================================
Omnibus:                        0.082   Durbin-Watson:                   2.015
Prob(Omnibus):                  0.960   Jarque-Bera (JB):                0.119
Skew:                          -0.020   Prob(JB):                        0.942
Kurtosis:                       2.964   Cond. No.                         23.7
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
 ↪specified.
```

Let's now **compare the results** obtained with the naive approach, ignoring the presence of confounders, and the 2SLS modelling.

```
# Extract coefficients and standard errors
ols_coef = ols_model.params.iloc[1]
ols_se = ols_model.bse.iloc[1]
second_stage_coef = second_stage_model.params.iloc[1]
second_stage_se = second_stage_model.bse.iloc[1]

# Plot the comparison
labels = ['OLS', '2SLS']
coefficients = [ols_coef, second_stage_coef]
errors = [ols_se, second_stage_se]

plt.figure(figsize=(5, 5), dpi=100)
plt.bar(labels, coefficients, yerr=errors, capsize=10, color=['m', 'c'], alpha=0.6)
plt.axhline(y=true_effect, color='k', linestyle='--', lw=3, alpha=0.8,  label='True␣
 ↪effect')
plt.ylabel('Coefficient of Wind Power Production')
plt.title('Comparison of OLS and 2SLS Estimates')
plt.legend()
plt.show()
```



We can now see tha simply using a model on all the observed variables leads us to a **biased estimate**. Instead, using 2SLS we can get much closer to the real effect.

CHAPTER

SEVENTEEN

PROPENSITY SCORE MATCHING

In many cases, we are interested in assessing the effectiveness of a campaign or other types of treatments. For example, let's assume we are interested in evaluating the impact of a **demand-response programme** on the electricity consumption of a set of households. Suppose that households can join the demand-response programme by filling out a specific form. For those who join, the programme offers reduced pricing during off-peak hours. Our goal is to assess the effectiveness of this programme by comparing the electricity consumption of people who joined the programme (our **treatment group**) to those who did not (our **control group**).

In an ideal scenario, if we were able to conduct a completely randomised experiment, we would have a balanced and diverse set of customers in both groups. In such a situation, if the possibility of joining the programme was as random as flipping a coin, any significant differences in consumption patterns between the two groups could be attributed to the special pricing offered by the programme.

However, in real life, things are more complicated. Because people are not randomly assigned to the two groups, there may be characteristics of the people who voluntarily joined the programme that are the true causes of changes in consumption. For example, if a majority of the people who agreed to join the programme are retired, it is likely that they will use electricity during off-peak hours due to their lifestyle and not solely because of the reduced pricing. This is an example of **selection bias**.

Whenever the treatment and control assignment is not random, there may be demographic or personal factors that affect the outcome of the experiment. These factors are referred to as **confounders**. This problem is particularly prevalent in observational data (see [RR83]).

Propensity score matching (PSM) is a statistical technique used to estimate the effect of a treatment by accounting for covariates that predict receiving the treatment. In our case, this means we can try to estimate the effect of the demand-response programme by considering the covariates that might have influenced who joined the programme. Essentially, if we can estimate the probability of someone joining the programme based on demographic factors, we can **match** treated and untreated subjects with similar propensity scores. By comparing these matched groups, we can compute an adjusted difference that helps mimic a randomised experimental design, reducing selection bias and allowing for a more accurate estimation of the treatment effect.

In mathematical terms, we can define the propensity score $e(x)$ as the probability of a unit (e.g., customer) joining the demand-response programme given a set of covariates x:

$$e(x) = P(T = 1 \mid X = x) \tag{17.1}$$

where T is a binary indicator of treatment assignment (1 if joined the programme, 0 if not).

Propensity scores are typically estimated using **logistic regression**, although other classification methods can be used. For example:

$$\hat{e}(x) = \hat{P}(T = 1 \mid X = x) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p))} \tag{17.2}$$

After estimating the propensity scores, we can proceed with the matching process. There are several ways to perform matching, such as:

1. **Nearest-neighbor matching**: each treated unit is matched with the control unit that has the closest propensity score.
2. **Caliper matching**: similar to nearest-neighbor matching but with a tolerance level (caliper) for how close the propensity scores must be.
3. **Stratification (or interval matching)**: units are divided into strata based on their propensity scores, and comparisons are made within each stratum.
4. **Kernel matching**: Weights all control units according to their distance from the treated unit's propensity score.

17.1 Example

Let's walk through an example using Python to illustrate how to implement propensity score matching, estimate the treatment effect, and visualize the results.

17.1.1 Data generation

We will use a synthetic dataset to simulate our scenario. Suppose our dataset includes the following columns:

- `joined_program`: A binary indicator (1 if joined the program, 0 if not).
- `peak_consumption`: The electricity consumption of the household during peak hours.
- `age`: Age of the household head.
- `income`: Income level of the household.
- `household_size`: Number of members in the household.

We also introduce some **bias in the treatment assignment** to simulate a real-world scenario where older people are more likely to join the programmw and lower-income households are less likely to join.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import NearestNeighbors
import seaborn as sns

np.random.seed(0)
n = 1000
age = np.random.normal(50, 12, n).astype(int)
income = np.random.normal(50000, 15000, n).astype(int)
household_size = np.random.randint(1, 6, n)

# Bias in treatment assignment
noise = np.random.normal(0, 10, n)
prob_join = 1 / (1 + np.exp(-(0.5 * age - 0.0001 * income + 0.2 * household_size - 5␣
↪+ noise)))
joined_program = (np.random.rand(n) < prob_join).astype(int)

peak_consumption = (200 - 0.001 * income + 0.2 * household_size - 2 * joined_program␣
↪+ np.random.normal(0, 1, n))

data = pd.DataFrame({
    'joined_program': joined_program,
```

(continues on next page)

(continued from previous page)

```
    'peak_consumption': peak_consumption,
    'age': age,
    'income': income,
    'household_size': household_size
})


data.head()
```

	joined_program	peak_consumption	age	income	household_size
0	1	140.595562	71	58339	3
1	1	134.926546	54	63387	2
2	1	154.885678	61	43665	4
3	1	147.252513	76	51570	1
4	1	146.337150	72	53420	1

17.1.2 Propensity scores estimation

In this step, we use logistic regression to estimate the propensity scores, which represent the probability of joining the programme given the covariates (age, income, and household size).

```
covariates = ['age', 'income', 'household_size']
X = data[covariates]
y = data['joined_program']

log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X, y)
data['propensity_score'] = log_reg.predict_proba(X)[:, 1]

data.head()
```

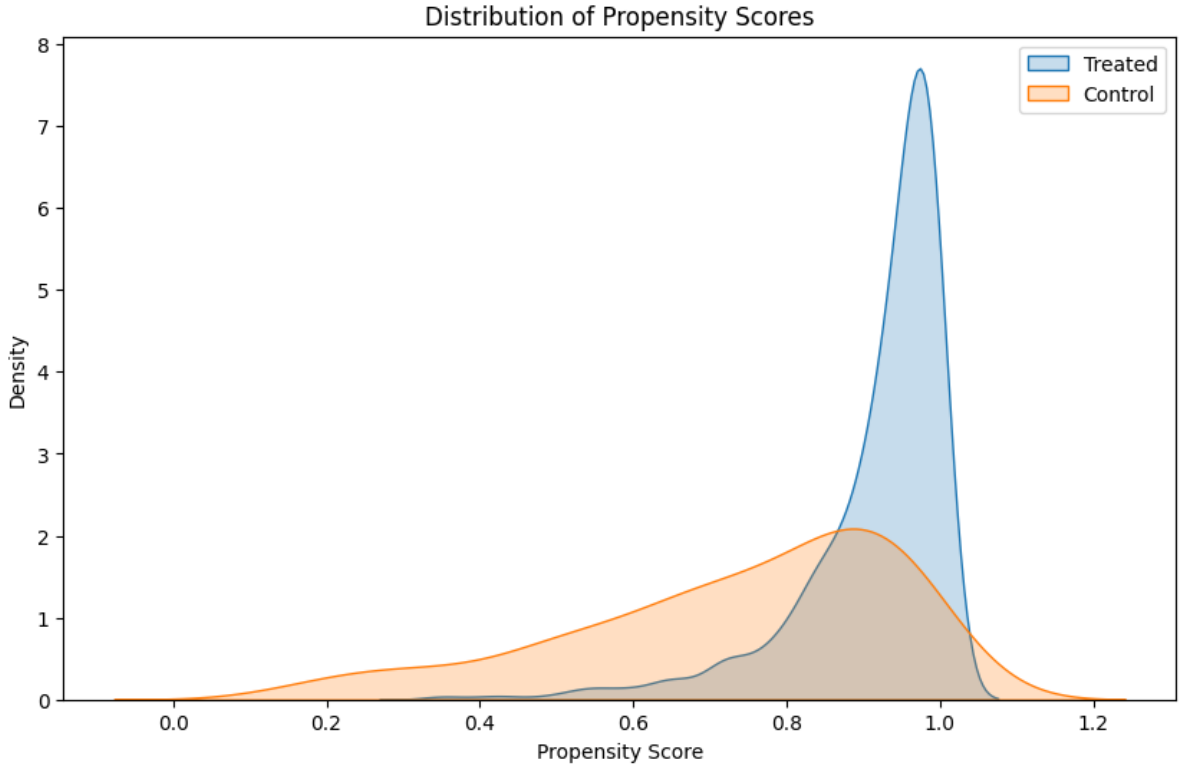
	joined_program	peak_consumption	age	income	household_size	propensity_score
0	1	140.595562	71	58339	3	0.994963
1	1	134.926546	54	63387	2	0.959616
2	1	154.885678	61	43665	4	0.988158
3	1	147.252513	76	51570	1	0.997573
4	1	146.337150	72	53420	1	0.995956

```
plt.figure(figsize=(10, 6))
sns.kdeplot(data['propensity_score'][data['joined_program'] == 1], label='Treated',␣
 ↪fill=True)
sns.kdeplot(data['propensity_score'][data['joined_program'] == 0], label='Control',␣
 ↪fill=True)
plt.title('Distribution of Propensity Scores')
plt.xlabel('Propensity Score')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Density')
plt.legend()
plt.show()
```



In this plot, the extent to which the treated and control **propensity score distributions overlap** is crucial:

- Significant overlap indicates that for many households, there are comparable counterparts in both groups, which is good for matching.
- Minimal overlap suggests that there are many treated households with no comparable controls (or vice versa), which can be problematic for matching.

Before matching, it is common to see differences in the distributions, especially if certain covariates strongly influence whether households join the program. The initial (pre-matching) plot might show that the treated group has higher propensity scores on average, indicating that those who joined the program had characteristics that made them more likely to join. This bias is what matching aims to adjust for by finding control units with similar propensity scores.

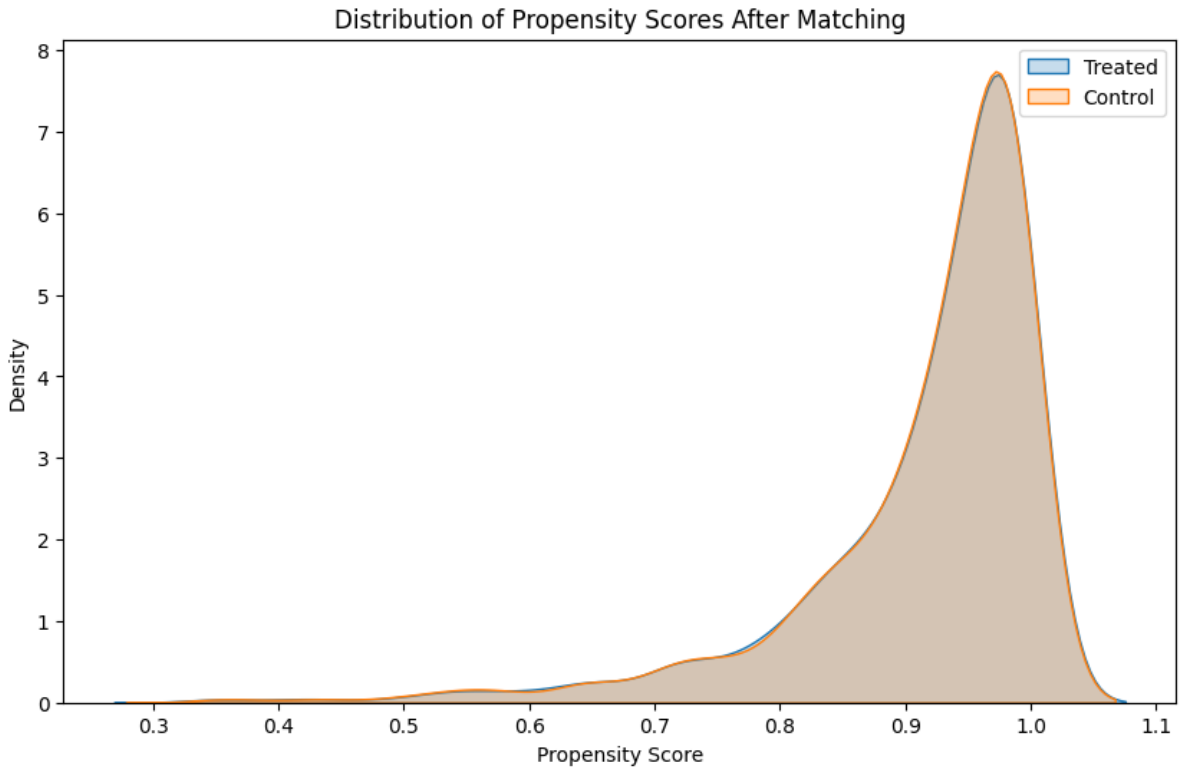
17.1.3 Nearest-neighbor matching

We separate the treated (joined the program) and control (did not join the program) groups. Using the nearest-neighbor matching technique, we match each treated unit with the control unit that has the closest propensity score. The goal here is to match each treated unit (household that joined the programme) with one or more control units (households that did not join the programme) that have similar propensity scores. Here's a detailed explanation of how we can do it:

- We separate the treated and control groups based on whether they joined the program or not.
- We find the closest control units for each treated unit based on their propensity scores.
- We combine the treated units and their matched control units into a single dataset for further analysis.

```
# Step 3: Perform Matching
treated = data[data['joined_program'] == 1]
control = data[data['joined_program'] == 0]

nbrs = NearestNeighbors(n_neighbors=1, algorithm='ball_tree').fit(control[[
 ↪'propensity_score']])
distances, indices = nbrs.kneighbors(treated[['propensity_score']])

# Get the matched control units
matched_control_indices = indices.flatten()
matched_control = control.iloc[matched_control_indices]

matched_pairs = pd.concat([treated.reset_index(drop=True), matched_control.reset_
 ↪index(drop=True)], axis=1)
matched_pairs.columns = ['t_' + col if i < len(treated.columns) else 'c_' + col for i,
 ↪ col in enumerate(matched_pairs.columns)]
```

```
# Distribution of propensity scores after matching
plt.figure(figsize=(10, 6))
sns.kdeplot(matched_pairs['t_propensity_score'], label='Treated', fill=True)
sns.kdeplot(matched_pairs['c_propensity_score'], label='Control', fill=True)
plt.title('Distribution of Propensity Scores After Matching')
plt.xlabel('Propensity Score')
plt.ylabel('Density')
plt.legend()
plt.show()
```



As we see here, after matching, we ideally want the distributions to overlap substantially, indicating that the treated and control groups are similar with respect to the covariates used to estimate the propensity scores.

17.1.4 Treatment effect estimation

We calculate the Average Treatment Effect on the Treated (ATT) by comparing the average electricity consumption of the treated group to that of the matched control group.

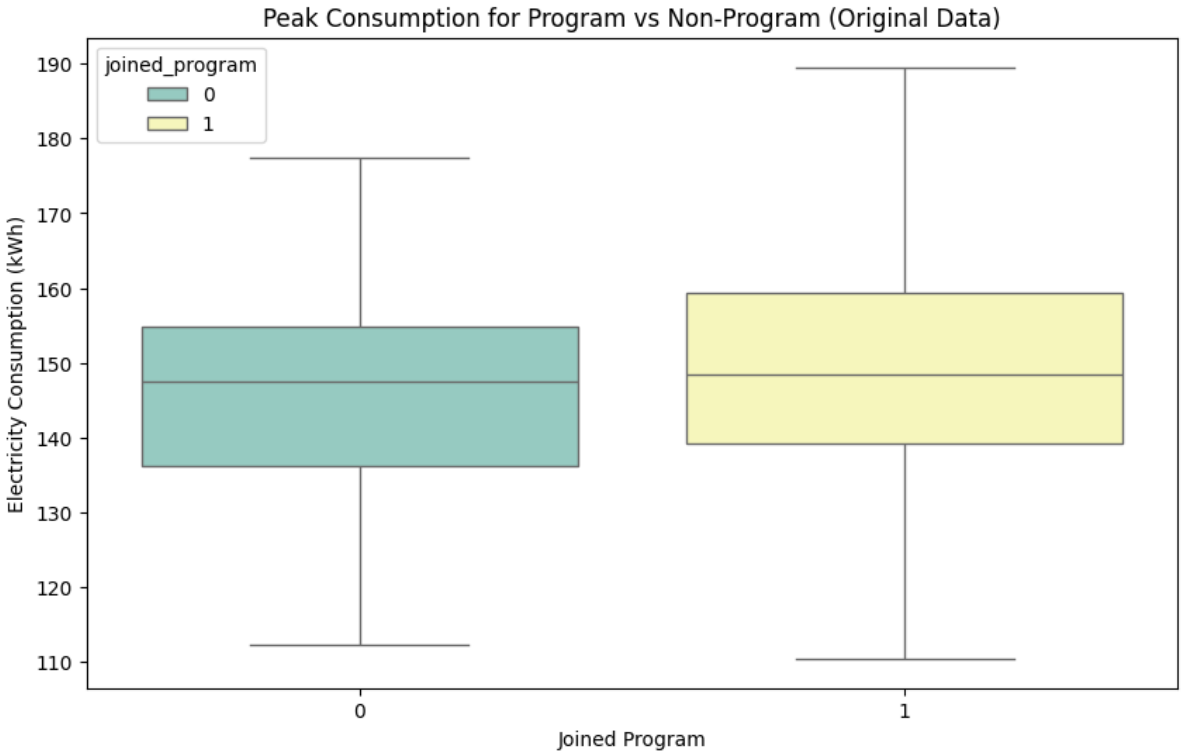
```
treatment_effect = matched_pairs['t_peak_consumption'].mean() - matched_pairs['c_peak_
↪consumption'].mean()
print(f'Estimated Treatment Effect: {treatment_effect:.3f} KWh')
```

```
Estimated Treatment Effect: -2.583 KWh
```

17.1.5 Visualising results

We can create some visualisations to compare the distributions of propensity scores and electricity consumption before and after matching.

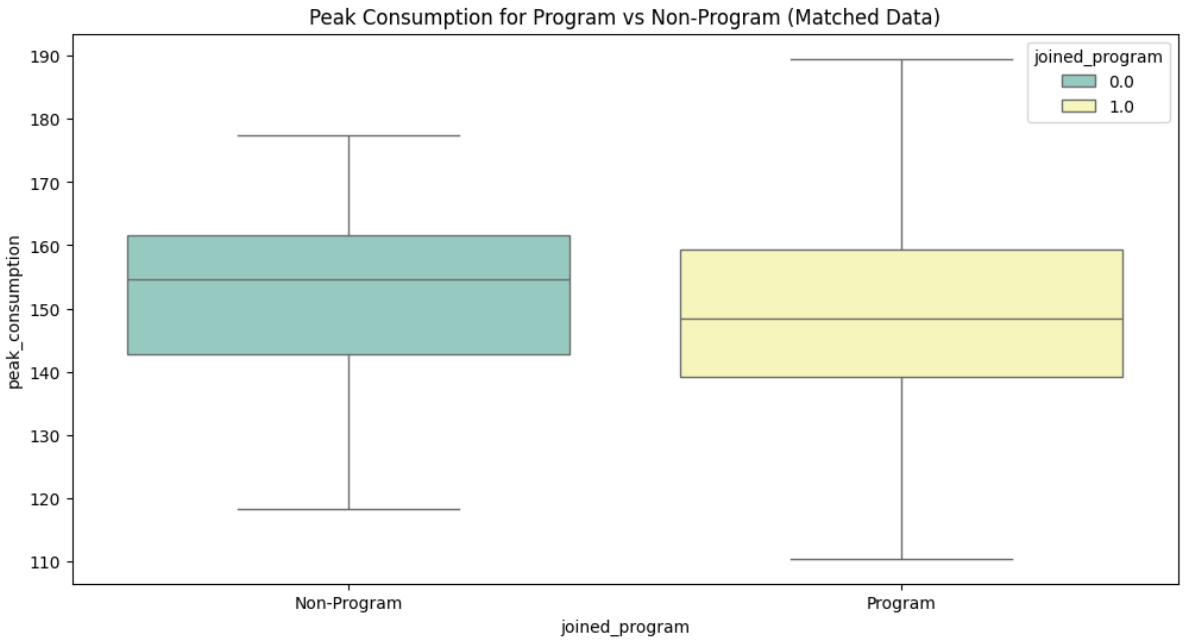
```
# Electricity consumption before and after matching
plt.figure(figsize=(10, 6))
sns.boxplot(x='joined_program', y='peak_consumption', data=data, palette='Set3', hue=
↪'joined_program', showfliers=False)
plt.title('Peak Consumption for Program vs Non-Program (Original Data)')
plt.xlabel('Joined Program')
plt.ylabel('Electricity Consumption (kWh)')
plt.show()
```



We can see how, looking at the original data, we would get the impression that people who joined the programme consume more lectricity during peak hours. The situation is much different when we look at the consumptions for the matched

groups, where we see that joining the programme actually has a positive effect in reducing electricity consumption during peak hours.

```
# Violin plot of consumption for matched data
# Boxplot of consumption for matched data
matched_data = pd.DataFrame({
    'joined_program': np.concatenate([np.ones(len(matched_pairs)), np.
 ↪zeros(len(matched_pairs))]),
    'peak_consumption': np.concatenate([matched_pairs['t_peak_consumption'], matched_
 ↪pairs['c_peak_consumption']])
})

plt.figure(figsize=(12, 6))
sns.boxplot(x='joined_program', y='peak_consumption', data=matched_data, palette='Set3
 ↪', hue='joined_program', showfliers=False)
plt.title('Peak Consumption for Program vs Non-Program (Matched Data)')
plt.xticks([0, 1], ['Non-Program', 'Program'])
plt.show()
```



This example demonstrates the basic steps of performing propensity score matching and evaluating the treatment effect. By matching treated and control units with similar propensity scores, we reduce the bias due to confounding variables and can make a more accurate assessment of the treatment's impact.

CHAPTER

EIGHTEEN

DOUBLE MACHINE LEARNING

We have already seen that we can use Propensity Score Matching (PSM) to adjust for confounders when estimating treatment effects. In high-dimensional settings or when using machine learning model, things can get more complicated. For these circumstances, Chernozhukov et al. [CCD+18] proposed the Double Machine Learning (DML) is a framework, to estimate causal effects when many confounding variables are present. It combines machine learning techniques with econometric methods to control for these confounders and obtain unbiased estimates of treatment effects. For example, let's assume we are interested in unveiling the effect of wind power (WP) production or solar power (SP) production on electricity prices. In this case, WP and SP production would be our **treatment variables**, while the electricity price would be our **response**. We know that these factors have an effect in reducing prices due to their low marginal costs. However, we also know that there are many other factors affecting electricity prices (e.g., demand, gas prices, macroeconomic trends). These are the **confounders**. We can also assume that some these confounders have an effect both on our treatment and our response variable. For example, the season we are in or the specific hour of the day will certainly affect the generation of SP, but will also have an effect on the demand (hence, the prices) because of the well-known daily and season consumption profiles. Ignoring the effect of these confounders might lead to biased estimates.

With DML, we are trying to isolate the effect of the treatment variables on the response. This framework assumes that the response y (e.g., the prices) is a function of the treatment w and other confounding variables x:

$$y = g(w, x) + \epsilon \tag{18.1}$$

where g is an arbitrary function (linear or nonlinear) and ϵ is the error term.

Similarly, since we assumed that the treatment is also affected by other confounding variables, we have that w can be modeled as a function of x:

$$w = m(x) + \nu \tag{18.2}$$

where m is an arbitrary function (linear or nonlinear) and ν is the error term.

Now, the DML framework involves two main stages:

1. **Nuisance parameter estimation**: use a machine learning model to estimate the functions $\widehat{g}(w, x)$ and $\widehat{m}(x)$.
2. **Orthogonalization and estimation**: use the estimated functions to "remove" the effect of the confounding variables from both w and y. Then, we estimate the causal effects by regressing the residuals of the response on the residuals of the treatment.

The **key intuition** is that if we remove the effect of other confounders from the tratment and the response, the variation that remains in the residuals is only due to the treatment itself. It should be noted that this approach assumes we already know the causal graph, and that there are no omitted variables.

18.1 The Partially Linear Case

For simplicity, we now consider a partially linear case where the relationship between the outcome y and the treatment w can be expressed linearly, while allowing for a potentially complex, nonlinear relationship between the confounders x and both the treatment and outcome.

In this case, the model is specified as follows:

$$y = \beta w + g(x) + \epsilon \tag{18.3}$$

$$w = m(x) + \nu \tag{18.4}$$

Here:

- β is the coefficient capturing the causal effect of the treatment w on the outcome y. This is what we are trying to estimate!
- $g(x)$ is an unknown function capturing the effect of the confounders x on the outcome.
- $m(x)$ is an unknown function capturing the effect of the confounders x on the treatment.

The **key steps** to implement the DML in the partially linear case are:

1. **Split the data**: randomly split the data into K folds.
2. **Train predictive models**: for each fold k (where $k \in \{1, 2, ..., K\}$):
    - **Treatment model**: train a machine learning model $\hat{m}_{-k}(x)$ using $K-1$ folds to predict w from x.
    - **Outcome model**: train a machine learning model $\hat{g}_{-k}(x)$ using $K-1$ folds to predict y from x.
3. **Generate residuals**:
    - Use the models trained on $K-1$ folds to predict the held-out fold k.
    - Compute the residuals for the treatment and outcome models:

$$\hat{V}_W = W - \hat{W}, \quad \hat{V}_Y = Y - \hat{Y} \tag{18.5}$$

4. **Regress residuals**: regress the residualized outcome $\hat{V}_Y$ on the residualized treatment $\hat{V}_W$ to estimate the causal effect β: \begin{equation} \hat{\beta}_k = \text{coef}\left(\hat{V}_Y \sim \hat{V}_W \right) \end{equation}
5. **Average estimates**: repeat steps 2-4 for each fold and average the resulting K estimates to obtain the final causal estimate: \begin{equation} \hat{\beta} = \frac{1}{K} \sum_{k=1}^{K} \hat{\beta}_k \end{equation}
6. **Robustness**: for more robustness with respect to random partitioning in finite samples, repeat the algorithm multiple times (e.g., 100 times) with different random splits and report the median estimate.

This algorithm ensures that the estimation of the treatment effect is orthogonal to the nuisance parameters (the confounders), thereby removing bias due to overfitting and ensuring that the estimated treatment effect is unbiased.

18.2 Electricity example

Let's now consider a practical case where we want to estimate the effect of day-ahead wind power and solar power forecasts on spot prices. We know that the forecasts available before gate closure [JonssonPM10] are crucial to determine the equilibrium price. Because of the merit-order effect, the low marginal costs of renewable energy sources acts like a shifter in reducing the equilibrium price. We now assume to be in a simplified setting where we want to quantify this effect. We consider a simple **causal graph** where the spot prices are linearly affected by the forecasted WP and SP

production levels, and nonlinearly affected by other endogenous factors and market conditions. In particular, we assume to be in the following **partially linear case**:

$$\text{spot price} = -0.3 * \text{WP} - 0,3 * \text{SP} + f_L(\text{load}) + f_G(\text{gas}) + f_C(\text{coal}) + f_{CO2}(\text{carbon pricing}) + \epsilon \tag{18.6}$$

where $f_L, \ldots, f_{CO2}$ are arbitrary nonlinear functions, and ϵ is the error term.

Our goal is to estimate the coefficients -0.3 as accurately as possible, having collected data from the following causal graph.

18.2.1 The causal graph

```
import graphviz
from IPython.display import display

# Create a new graph
dot = graphviz.Digraph()

# Add nodes
dot.node('S', 'Daylight hours \n(proxy for season)', color='gray')
dot.node('WP', 'Forecasted \nWP \nproduction', penwidth='2', color='green')
dot.node('SP', 'Forecasted \nSP \nproduction', penwidth='2', color='green')
dot.node('L', 'Load forecast', color='gray')
dot.node('G', 'Gas price', color='gray')
dot.node('C', 'Coal price', color='gray')
dot.node('CO2', 'Carbon pricing', color='gray')
dot.node('P', 'Day-ahead spot prices', penwidth='2', color='green')

# Define coefficients for the edges
coefficients = {
    ('S', 'L'): '',
    ('S', 'WP'): '',
    ('S', 'SP'): '',
    ('WP', 'P'): '-0.3',
    ('SP', 'P'): '-0.3',
    ('L', 'P'): '',
    ('G', 'P'): '',
    ('CO2', 'P'): '',
    ('C', 'P'): ''
}

# Add edges with coefficients as labels and make only WP->P and SP->P green
for (start, end), coeff in coefficients.items():
    edge_color = 'green' if (start, end) in [('WP', 'P'), ('SP', 'P')] else 'gray'
    dot.edge(start, end, label=coeff, color=edge_color, penwidth='2' if (start, end)␣
↪in [('WP', 'P'), ('SP', 'P')] else '1')


# Display the graph in the notebook
display(dot)
```

```
<graphviz.graphs.Digraph at 0x107d7f6d0>
```

18.2.2 The data-generating process

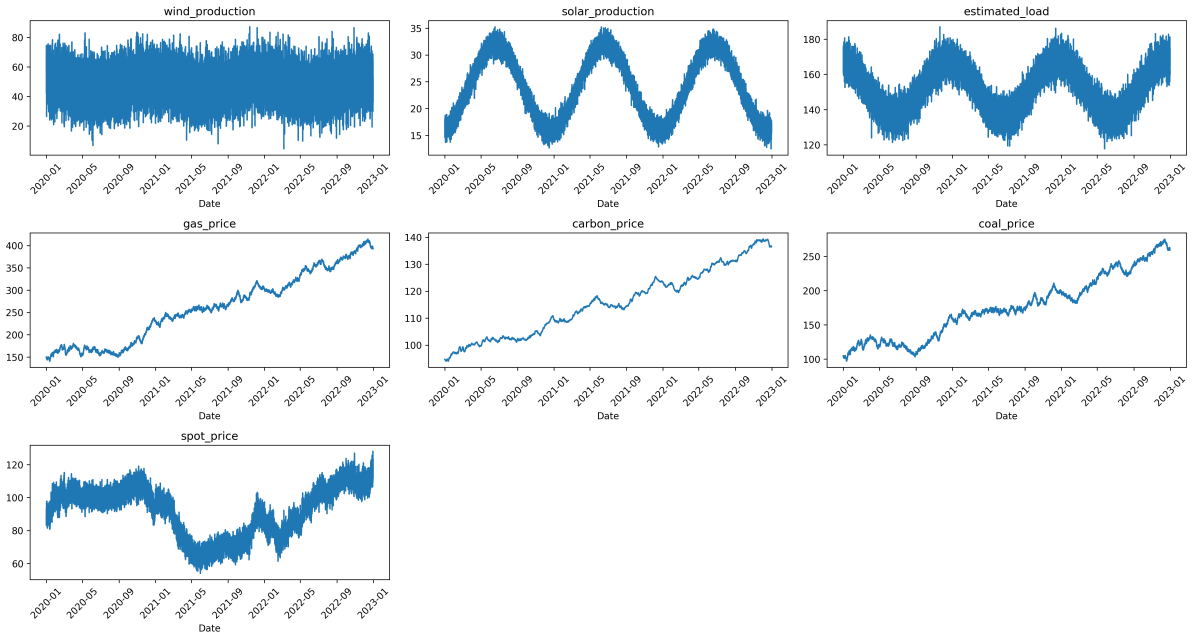
We can now create a simple simulator for this kind of data, and plot the results of a simulation run, to show the data:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from pygam import LinearGAM, s, f
from sklearn.model_selection import KFold
from scipy.ndimage import gaussian_filter
from tqdm import tqdm

def generate_data(n_years=1, random_seed=None, plot_time_series=False):
    if random_seed is not None:
        np.random.seed(random_seed)

    hours_per_day = 24
    days_per_year = 365
    n = hours_per_day * days_per_year * n_years

    date_range = pd.date_range(start='2020-01-01', periods=n, freq='h')
    time = np.arange(n)
    day_of_year = (time // hours_per_day) % days_per_year
    D = 8 + 4 * np.sin(2 * np.pi * (day_of_year - 80) / days_per_year) + 4

    WP = 60 - 1 * D + np.random.normal(0, 10, n)
    SP = 2 * D + np.random.normal(0, 1, n)
    L = 200 - 4 * D + np.random.normal(0, 5, n)
    G = 150 + np.cumsum(np.random.normal(0, 0.5, n))
    G = np.maximum(G, 50) + np.random.normal(0, 0.01, n)
    C = 80 + 0.1 * G + 0.05 * np.cumsum(np.random.normal(0, 1, n))
    C = gaussian_filter(np.maximum(C, 40), sigma=10) + np.random.normal(0, 0.01, n)
    O = 0.7 * G + 0.1 * np.cumsum(np.random.normal(0, 1, n))

    noiseless_P = (-0.3 * WP - 0.3 * SP  + (L**5)/10e9 +
                   -50 / (1 + np.exp(0.2 * (G - 100))) + 20 * np.sin(C/5) + 10 * np.
↪log(O + 1))
    P = 50 + noiseless_P + np.random.normal(0, .1, n)

    data = pd.DataFrame({
        'daylight_hours': D,
        'wind_production': WP,
        'solar_production': SP,
        'estimated_load': L,
        'gas_price': G,
        'carbon_price': C,
        'coal_price': O,
        'spot_price': P
    }, index=date_range)

    if plot_time_series:
        plt.figure(figsize=(18, 12), dpi=300)
        for i, var in enumerate(data.columns[1:], 1):
            plt.subplot(4, 3, i)
            plt.plot(data[var])
            plt.title(var)
```

(continues on next page)

(continued from previous page)

```
            plt.xlabel('Date')
            plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()

    return data

data = generate_data(n_years=3, random_seed=0, plot_time_series=True)
```



We can see that the simulator simply tries to emulate the seasonality in load and renewables production, with higher demands in colder months and higher SP production during summer months.

18.2.3 The DML algorithms

We can now implement the DML framework we have seen before, where we try to remove the effect of confounders by fitting a nonlinear model, in this case an generalized additive model (GAM) on the confounders. Here, we are in a peculiar DML setting, where the confounders affecting the treatment variables and the response variable are not the same. Indeed, we have:

- Confounder for the treatments: daylight hours.
- Confounders for the response: daylight hours, estimated load, gas price, coal price, carbon pricing.

Here is the code for implementing DML:

```
def dml_algorithm(data, n_splits=5, n_repeats=1, seed=42):
    results = []
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=seed)

    for _ in range(n_repeats):
        fold_coefs = []
        for train_index, test_index in kf.split(data):
            train, test = data.iloc[train_index], data.iloc[test_index]
```

(continues on next page)

(continued from previous page)

```
            confounders = ['gas_price', 'coal_price', 'carbon_price', 'estimated_load
↪', 'daylight_hours']

            # Residualize using GAM
            gam_wp = LinearGAM(s(0)).fit(train[['daylight_hours']], train['wind_
↪production'])
            gam_sp = LinearGAM(s(0)).fit(train[['daylight_hours']], train['solar_
↪production'])
            gam_spot_price = LinearGAM(s(0) + s(1) + s(2) + s(3) + s(4)).
↪fit(train[confounders], train['spot_price'])

            test_wp_residuals = test['wind_production'] - gam_wp.predict(test[[
↪'daylight_hours']])
            test_sp_residuals = test['solar_production'] - gam_sp.predict(test[[
↪'daylight_hours']])
            test_spot_price_residuals = test['spot_price'] - gam_spot_price.
↪predict(test[confounders])

            # Step 4: Regress residuals
            model = LinearRegression().fit(np.vstack([test_wp_residuals, test_sp_
↪residuals]).T, test_spot_price_residuals)
            fold_coefs.append(model.coef_)

        results.append(np.mean(fold_coefs, axis=0))

    return np.median(results, axis=0)
```

To have a **benchmark**, we also compare the DML method with two simpler approaches:

1. Two features: a simple approach where we fit a linear regression model only on the two treatment variables
2. ALl the features: an all-inclusive approach where we fit a linear regression model on all the available features.

```
def fit_models(data):
    results = {}

    # Method (i) Using only WP and SP as predictors
    X_wp_sp = data[['wind_production', 'solar_production']]
    y = data['spot_price']
    model_wp_sp = LinearRegression().fit(X_wp_sp, y)
    results['wp_sp']  = model_wp_sp.coef_

    # Method (ii) Using all variables as predictors
    X_all = data.drop(columns=['spot_price'])
    model_all = LinearRegression().fit(X_all, y)

    # Extracting coefficients for wind_production and solar_production specifically
    coef_all = model_all.coef_
    wp_idx = list(X_all.columns).index('wind_production')
    sp_idx = list(X_all.columns).index('solar_production')
    results['all'] = np.array([coef_all[wp_idx], coef_all[sp_idx]])

    return results
```

Let's now run replicate the data generation and parameter estimation experiment using the three approaches (DML and the two benchmarks):

```
n_iterations = 5
coefficients = {
    'wp_sp': [],
    'all': [],
    'DML': []
}

# Loop to generate data, residualize covariates, and fit models
for i in range(n_iterations):
    df = generate_data(n_years=3, random_seed=i)
    # df_residualized = residualize_data(df)
    df_residualized = df.copy()
    res1 = fit_models(df)
    res2 = dml_algorithm(df_residualized, seed=i)

    for key in res1:
        coefficients[key].append(res1[key])
    coefficients['DML'].append(res2)

# Convert to DataFrame for analysis
coefficients_df = {key: np.array(value) for key, value in coefficients.items()}
coefficients_df = pd.DataFrame({
    'Two features_wp': coefficients_df['wp_sp'][:, 0],
    'Two features_sp': coefficients_df['wp_sp'][:, 1],
    'All the features_wp': coefficients_df['all'][:, 0],
    'All the features_sp': coefficients_df['all'][:, 1],
    'DML_wp': coefficients_df['DML'][:, 0],
    'DML_sp': coefficients_df['DML'][:, 1]
})
```

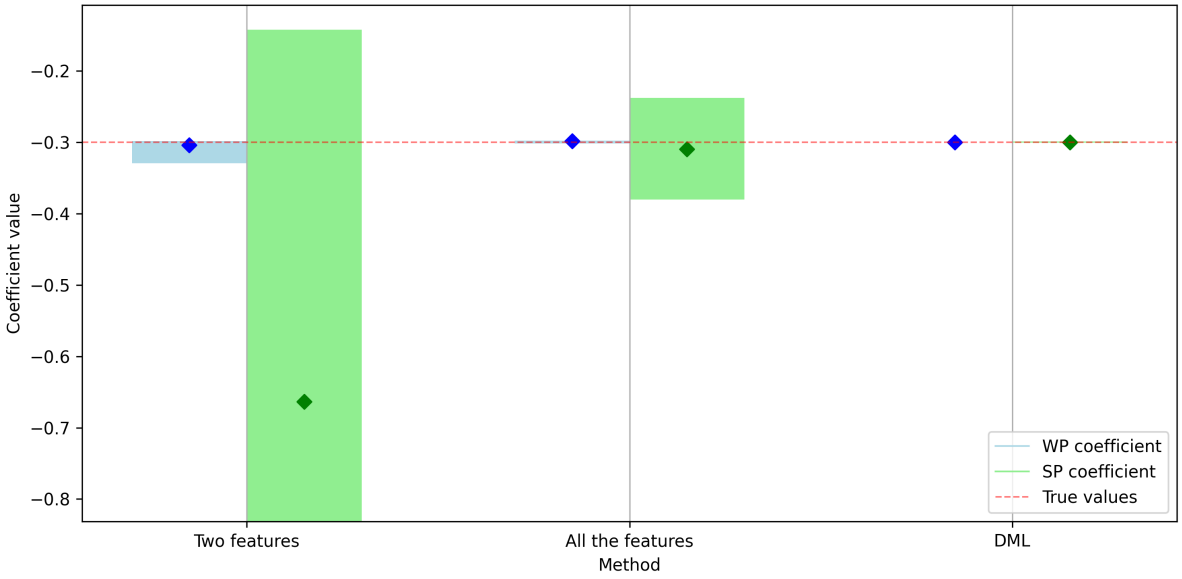
Let's now plot the results:

```
# Define the labels and colors
labels = ['Two features', 'All the features', 'DML']
colors = ['lightblue', 'lightgreen']
markers = ['blue', 'green']

# Calculate new quantiles (P10, P50, P90)
quantiles = coefficients_df.quantile([0.10, 0.5, 0.90]).T
quantiles.columns = ['P10', 'P50', 'P90']
quantiles = quantiles.reset_index().rename(columns={'index': 'Coefficient'})

# Plot the quantiles
fig, ax = plt.subplots(figsize=(10, 5), dpi=300)

bar_width = 0.3
bins = np.arange(len(labels))

for i, label in enumerate(labels):
    wp_label = f'{label}_wp'
    sp_label = f'{label}_sp'
    wp_data = quantiles[quantiles['Coefficient'] == wp_label]
    sp_data = quantiles[quantiles['Coefficient'] == sp_label]
    offset_wp = i - bar_width / 2
    offset_sp = i + bar_width / 2

    # WP bars and points
```

(continues on next page)

(continued from previous page)

```
    ax.bar(offset_wp, wp_data['P90'].values - wp_data['P10'].values, bottom=wp_data[
↪'P10'].values, width=bar_width, color='lightblue', label='WP coefficient' if i == 0␣
↪else "")
    ax.scatter(offset_wp, wp_data['P50'].values, color='blue', marker='D')

    # SP bars and points
    ax.bar(offset_sp, sp_data['P90'].values - sp_data['P10'].values, bottom=sp_data[
↪'P10'].values, width=bar_width, color='lightgreen', label='SP coefficient' if i ==␣
↪0 else "")
    ax.scatter(offset_sp, sp_data['P50'].values, color='green', marker='D')

# Set x-axis labels
ax.set_xticks(bins)
ax.set_xticklabels(labels)

# Simplified legend
handles = [
    plt.Line2D([0], [0], color='lightblue', lw=1, label='WP coefficient'),
    plt.Line2D([0], [0], color='lightgreen', lw=1, label='SP coefficient'),
    plt.Line2D([0], [0], color='red', linewidth=1, alpha=0.5, ls='--', label='True␣
↪values')
]

# Add horizontal line for the true value
ax.axhline(y=-0.3, color='red', linestyle='--', linewidth=1, alpha=0.5)
# ax.set_title('Quantiles of Estimated Coefficients for WP and SP')
ax.set_xlabel('Method')
ax.set_ylabel('Coefficient value')
ax.grid(axis='x')
ax.legend(handles=handles, loc='lower right')
plt.tight_layout()
plt.show()
```



We can see how the DML framework proves very effective in reducing the effect of the confounders on the estimation process. We can also see how the two-feature approach performs poorly, suffering from the **omitted variable bias**. Also, the all-inclusive approach, widely used in econometrics, has a much higher estimation variance. This is also due to the

fact the specified model is linear while some of the effects were actually nonlinear.

CHAPTER

NINETEEN

DIFFERENCE-IN-DIFFERENCES

Difference-in-differences (DiD) is a statistical technique used in econometrics to estimate the causal effect of a treatment on a time series. For example, it might be used to **evaluate the effect of a new policy** on the electricity prices, in a specific region.

Let us suppose we collected some data from electricity prices over time, and we know that at a certain point in time a new policy was introduced. Let us also suppose that we observed a change in the trend, and we would like to attribute that change to the newly introduced policy. To be sure that the change in price time series is indeed due to the new policy, we would need to know the **counterfactual**. The counterfactual represents what would have happened to the prices had the new policy not been introduced. Then, by comparing the prices observed under the new policy with what the prices would have been without the policy, we would finally be able to say that the prices changed because of the policy. Without the counterfactual, we might not be able to accurately determine the true effect of the policy because of:

- Other influencing factors: in the real world, multiple factors can influence electricity prices simultaneously. These could include changes in fuel prices, demand fluctuations, economic conditions, and other regulatory changes. Without a counterfactual, it is difficult to isolate the impact of the specific policy from these other variables.
- Temporal trends: electricity prices may follow certain trends over time regardless of the policy intervention. For instance, prices might be declining due to improvements in technology or increasing due to rising demand. The counterfactual helps to control for these underlying trends, providing a clearer picture of what the prices would have looked like in the absence of the policy.

Unfortunately, in observational data, we cannot observe the counterfactual, since we only have access to what has happened under the new policy. DiD tries to tackle this problem by comparing the changes in the prices over time before a treatment group and a control group, where:

- The **treatment group** is, for example, the data collected from a region where the new policy was introduced.
- The **control group** might refer to another region, where the new policy was not introduced.

While doing so, DiD relies a crucial assumption known as the **parallel trends assumption**. This assumption asserts that the treatment and control groups would have followed the same trajectory over time in the absence of the treatment. Using the parallel trends assumption, we can use the change of the prices of the control group as a counterfactual for the treatment group in the absence of the treatment. n simple terms, DiD means that we are looking at:

$$\begin{aligned} \text{DiD} =& (\text{price in treatment group before policy} - \text{price in treatment group after policy}) && (19.1)\\ & - (\text{price in control group before policy} - \text{price in control group after policy}) && (19.2) \end{aligned}$$

In other words, we are checking if there is a **difference between the two individual differences**. The first difference measures the change in prices for the treatment group before and after the policy, while the second difference measures the change in prices for the control group before and after the policy. By subtracting these two differences, we can isolate the effect of the policy from other factors that might influence electricity prices over time.

In practice, a common approach to DiD is to specify a linear regression model for the outcome of interest (in this case the price) as in:

$$y = \beta_0 + \beta_1 \,\text{group} + \beta_2 \,\text{period} + \beta_3 \,(\text{group} \times \text{period}) + \varepsilon \tag{19.3}$$

where:

- β_0, β_1, β_2, and β_3 are the regression coefficients.
- "group" is a dummy variable (0 or 1) that indicates whether the observation is from the control group (0) or the treatment group (1).
- "period" is a dummy variable (0 or 1) that indicates whether the observation is from the period before the policy implementation (0) or after (1).
- "group $\times$ period" is an interaction term to count for the **DiD causal effect**. This variable captures the combined effect of being in the treatment group and being in the post-policy period.

The model can be estimated with traditional methods such as ordinary least squares (OLS). DiD can also be extended to nonlinear or semi-parametric settings.

To provide a practical **example**, we will now generate some data for electricity prices over time for two regions: one where the policy was implemented (treatment group) and one where it was not (control group). We will include a change in the prices of the treatment gorup by adding a **policy effect** that reduces the prices of 10 units after the policy has been introduced.

```
import numpy as np
import pandas as pd

# Set random seed for reproducibility
np.random.seed(42)

# Generate time series data
n_periods = 100
time = np.arange(n_periods)
policy_start = 50  # Time when the policy starts

# Generate prices for control group
control_prices = 0.5 * time + np.random.normal(scale=2, size=n_periods)

# Generate prices for treatment group with a policy effect
treatment_prices = 50 + 0.5 * time + np.random.normal(scale=2, size=n_periods)
treatment_prices[policy_start:] += -10  # Policy effect

# Create a DataFrame
data = pd.DataFrame({
    'time': np.tile(time, 2),
    'price': np.concatenate([control_prices, treatment_prices]),
    'group': np.repeat(['control', 'treatment'], n_periods),
    'period': np.concatenate([np.zeros(policy_start), np.ones(n_periods - policy_
 ↪start), np.zeros(policy_start), np.ones(n_periods - policy_start)])
})

data
```

```
     time       price      group  period
0       0    0.993428    control     0.0
1       1    0.223471    control     0.0
2       2    2.295377    control     0.0
3       3    4.546060    control     0.0
4       4    1.531693    control     0.0
..    ...         ...        ...     ...
195    95   88.270635  treatment     1.0
```

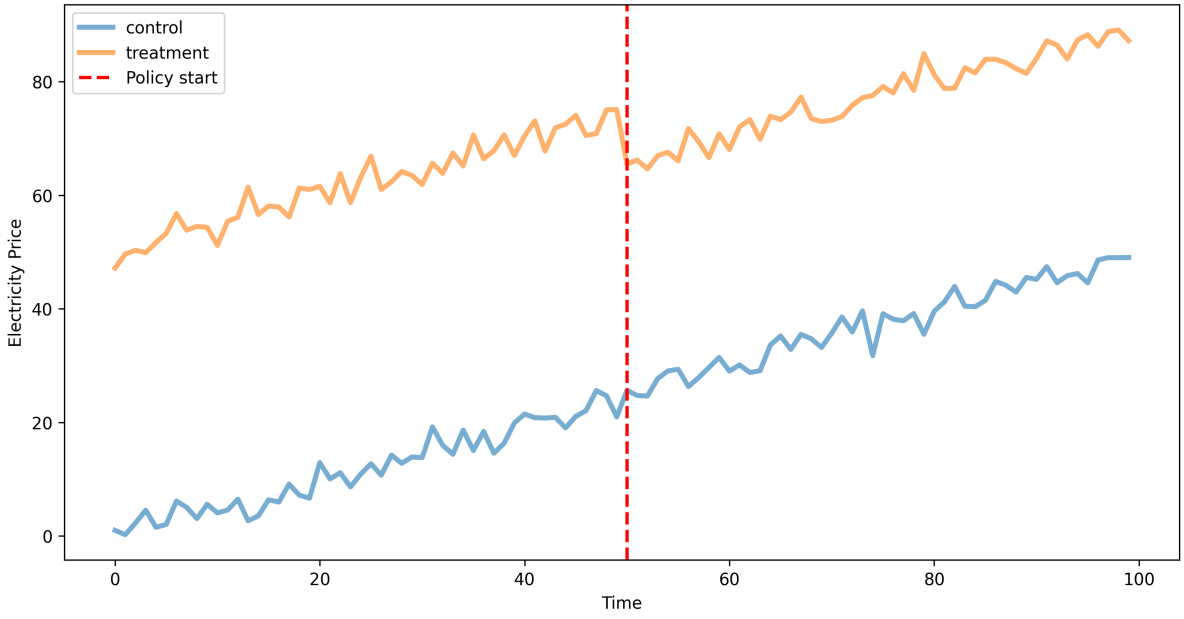
(continues on next page)

(continued from previous page)

```
196    96  86.232285  treatment     1.0
197    97  88.807450  treatment     1.0
198    98  89.116417  treatment     1.0
199    99  87.214059  treatment     1.0

[200 rows x 4 columns]
```

Let's now plot the time series corresponding to the treatment and control groups to visually explore potential effects.

```
import matplotlib.pyplot as plt

# Plot the data
plt.figure(figsize=(12, 6), dpi=300)
for key, grp in data.groupby('group'):
    plt.plot(grp['time'], grp['price'], label=key, lw=3, alpha=.6)
plt.axvline(policy_start, color='red', linestyle='--', label='Policy start', lw=2)
plt.xlabel('Time')
plt.ylabel('Electricity Price')
plt.legend()
plt.show()
```



As we can see, the prices in the treatment group appears to have slightly changed after the policy. Using the **parallel trends assumption**, we check for the difference in the prices, assuming the price in the treatment group would have followed the same trend of the control group.

Before fitting a DiD model, we need to create the dummy variables related to the group and the interaction between the group and the policy.

```
# Create a dummy variable for the group
data['group'] = np.where(data['group'] == 'treatment', 1, 0)

# Add an interaction term for DiD
data['interaction'] = data['group'] * data['period']
```

We now fit a simple OLS model on the data:

```
import statsmodels.api as sm

X = sm.add_constant(data[['group', 'period', 'interaction']])
y = data['price']
model = sm.OLS(y, X).fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.923
Model:                            OLS   Adj. R-squared:                  0.921
Method:                 Least Squares   F-statistic:                     778.6
Date:                Sun, 07 Jul 2024   Prob (F-statistic):          1.37e-108
Time:                        20:03:16   Log-Likelihood:                -679.31
No. Observations:                 200   AIC:                             1367.
Df Residuals:                     196   BIC:                             1380.
Df Model:                           3
Covariance Type:            nonrobust
===============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
const          11.7991      1.032     11.431      0.000       9.763      13.835
group          50.3724      1.460     34.507      0.000      47.493      53.251
period         25.4865      1.460     17.459      0.000      22.608      28.365
interaction   -10.2401      2.064     -4.960      0.000     -14.312      -6.169
==============================================================================
Omnibus:                       58.622   Durbin-Watson:                   0.309
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               10.369
Skew:                          -0.027   Prob(JB):                      0.00560
Kurtosis:                       1.886   Cond. No.                         6.85
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

We can see that the **interaction coefficients captures the causal effect of the policy**, accurately showing that the policy caused an increase of 10 units in the prices of the treatment group. This interaction term captures the differential effect of the policy on the treatment group relative to the control group, accounting for time trends common to both groups. A significant negative coefficient suggests that the policy led to a **reduction in prices for the treatment group compared to the control group**.

The **interaction term** effectively isolates the impact of the policy on the treatment group by controlling for:

- Any pre-existing differences between the treatment and control groups (coefficient “group”).
- Any changes over time that would affect both groups equally (coefficient “period”).

CHAPTER

TWENTY

INTERRUPTED TIME SERIES

Interrupted time series (ITS) analysis is econometrics approach that allows us to evaluate the impact of an intervention or policy change that occurs at a specific point in time. Unlike DiD, ITS **does not require a control group** as it assumes that the data-generating process would have continued in a similar way without the introduction of the new policy. ITS can be seen as a special case of a regression discontinuity design (RDD). In a typical RDD, the discontinuity is observed across different units based on a cutoff point. For example, the introduction of a special tax for plants over a certain size. ITS work in a similar way but instead of having the *discontinuity* based on a specific variable, the **discontinuity occurs over time**. Then, we can model the time series of interest using a **segmented regression** approach as in:

$$y = \beta_0 + \beta_1 \text{ time} + \beta_2 \text{ period} + \beta_3 \text{ time after policy} + \varepsilon \tag{20.1}$$

where the regression coefficients are used to capture:

- The baseline intercept (β_0).
- The pre-policy slope (β_1).
- The change in level at the introduction of the new policy (β_2). It should be noted that "period" is a dummy variable (0 or 1) that indicates whether the observation is from the period before the policy implementation (0) or after (1).
- The change in slope after the introduction of the new policy (β_3).

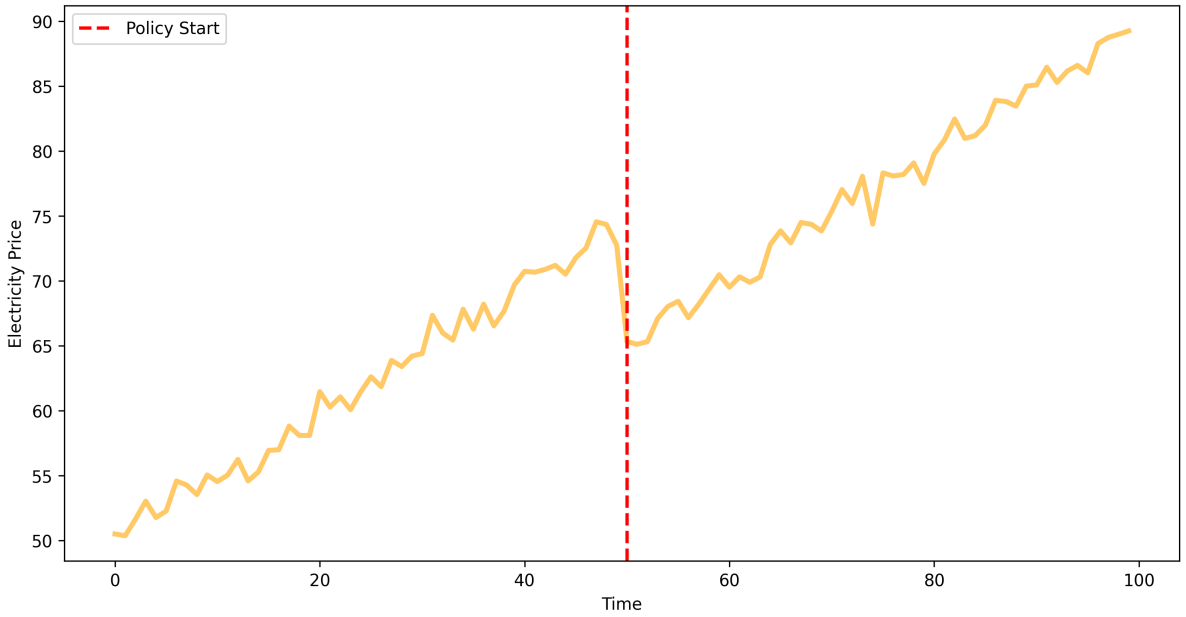
Let's now try to generate some data to try and apply this methodology in a practical **example**.

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

# Set random seed for reproducibility
np.random.seed(42)

# Generate time series data
n_periods = 100
time = np.arange(n_periods)
policy_start = 50  # Time when the policy starts

# Generate prices with a trend and some noise
prices = 50 + 0.5 * time + np.random.normal(scale=1, size=n_periods)

# Introduce a policy effect (e.g., a reduction in prices)
prices[policy_start:] += -10

# Create a DataFrame
data = pd.DataFrame({
    'time': time,
    'price': prices
```

(continues on next page)

(continued from previous page)

```
})

# Plot the data
plt.figure(figsize=(12, 6), dpi=300)
plt.plot(data['time'], data['price'], lw=3, alpha=.6, c='orange')
plt.axvline(policy_start, color='red', linestyle='--', label='Policy Start', lw=2)
plt.xlabel('Time')
plt.ylabel('Electricity Price')
plt.legend()
plt.show()
```



Here, we introduced a policy at time step 50, which causes a price reduction of 10 units. Let's now create the dummy variables and fit a segmented regression model to show if we are able to estimate the effect of the policy.

```
# Create the time after policy variable
data['time_after_policy'] = np.where(data['time'] >= policy_start, data['time'] -␣
↪policy_start, 0)
data['period'] = np.where(data['time'] >= policy_start, 1, 0)

import statsmodels.api as sm

# Define the independent variables
X = sm.add_constant(data[['time', 'period', 'time_after_policy']])
y = data['price']

# Fit the OLS model
model = sm.OLS(y, X).fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.993
Model:                            OLS   Adj. R-squared:                  0.993
```

(continues on next page)

(continued from previous page)

```
Method:                 Least Squares   F-statistic:                     4425.
Date:                Sun, 07 Jul 2024   Prob (F-statistic):           9.73e-103
Time:                        20:03:38   Log-Likelihood:                -129.67
No. Observations:                 100   AIC:                             267.3
Df Residuals:                      96   BIC:                             277.8
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.
↪975]
------------------------------------------------------------------------------
↪--
const         50.0644      0.252    198.924      0.000      49.565      50.
↪564
time           0.4882      0.009     55.153      0.000       0.471       0.
↪506
period        -9.3026      0.361    -25.742      0.000     -10.020      -8.
↪585
time_after_policy     0.0056      0.013      0.448      0.655      -0.019       0.
↪030
==============================================================================
Omnibus:                        0.634   Durbin-Watson:                   2.094
Prob(Omnibus):                  0.728   Jarque-Bera (JB):                0.440
Skew:                          -0.162   Prob(JB):                        0.802
Kurtosis:                       3.027   Cond. No.                         252.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

As we can see, the results show:

- The baseline price before the policy was 50 (β_0 = const).
- The prices were increasing by approximately 0.5 units per time period before the policy (β_1 = time).
- There was an immediate reduction of 9.3 units (fairly close to 10) in the price level after the policy (β_2 = period).
- The trend (slope) of prices did not significantly change after the policy (β_3 = time_after_policy).

20.1 Fitting an ITS on the residuals of a time series models

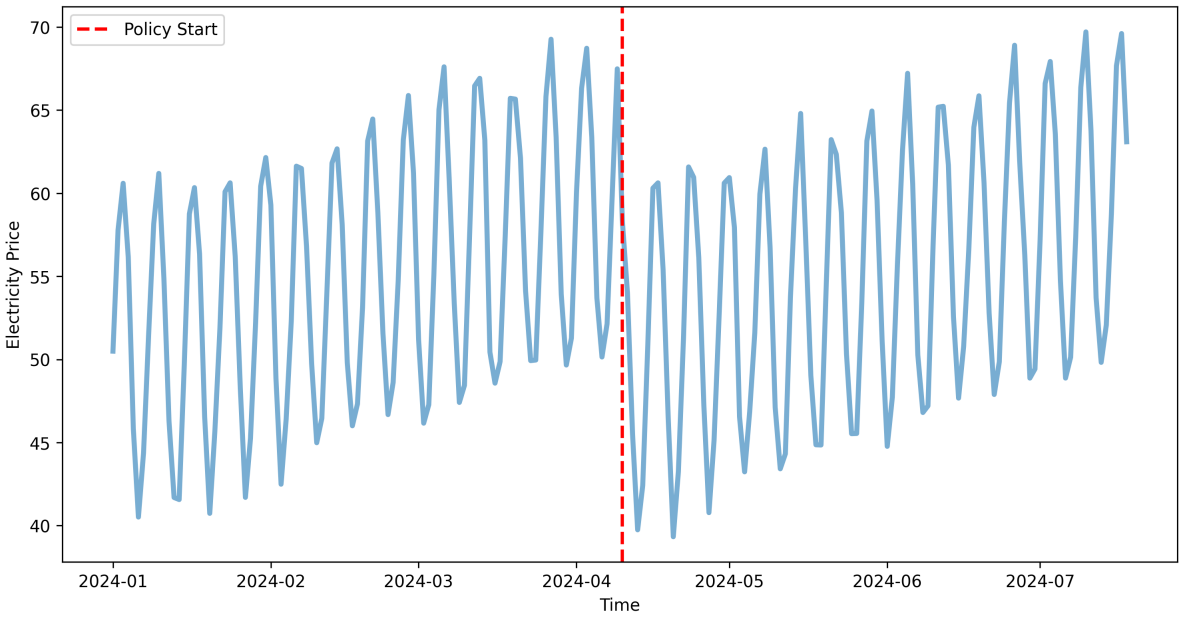
Many real-world data might be characterised by high autocorrelation or more complex dynamics. In that case, it might be useful to firts fit a model and then examine the behaviour of the residuals to estimate the effect of the policy. For example, using a time series model (e.g., SARIMA) followed by ITS analysis on the residuals might have several advantages, such as:

- Handling autocorrelation and seasonality: time series data often exhibit autocorrelation and seasonality, where current values are correlated with past values. Ignoring this aspect can lead to incorrect inferences and underestimated standard errors. By fitting a SARIMA model, we explicitly account for this, leading to more accurate residuals that reflect the underlying stochastic process.
- Isolating the intervention effect: by first modeling the time-dependent structure of the data (i.e., the autocorrelation), we can better isolate the effect of the intervention. The residuals from the SARIMA model represent the portion

of the time series that is not explained by its own past values, thus providing a clearer signal of any intervention effects.

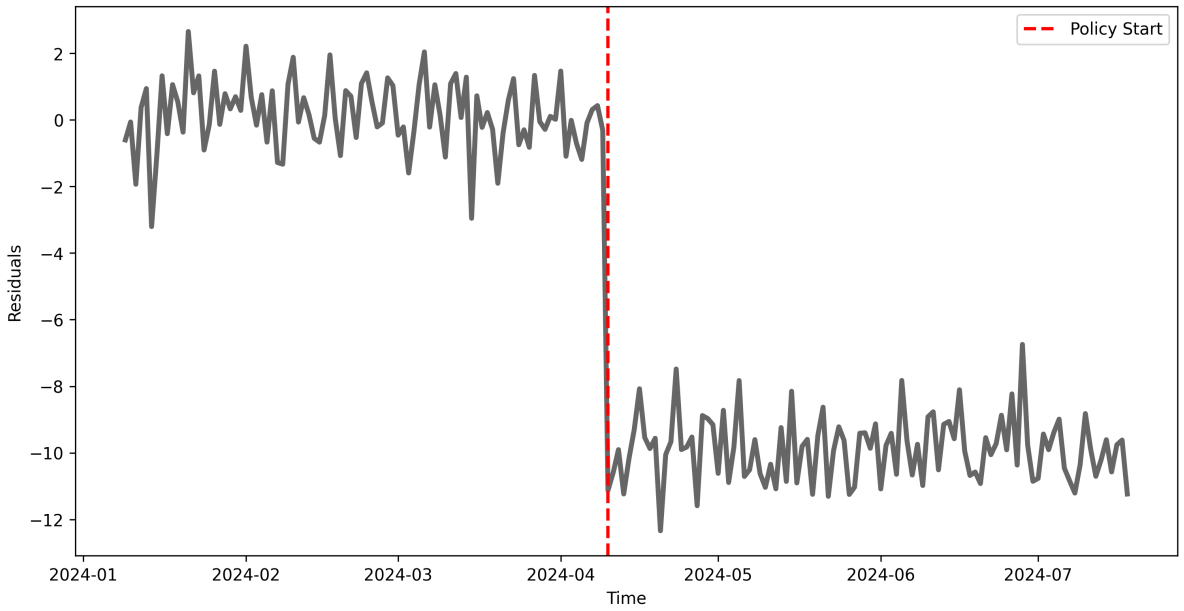
Let's try and generate some data, where we fit a SARIMA model on the pre-policy data, and use it to whiten the whole time series observed. Also in this case, we assume the **policy effect** is to reduce prices by 10 units.

```
# Set random seed for reproducibility
np.random.seed(42)

# Generate time series data with seasonality, trend, and some noise
n_periods = 200
time = pd.date_range(start='2024-01-01', periods=n_periods, freq='D')
seasonal_period = 7
policy_start = 100  # Time index when the policy starts

# Generate seasonal component
seasonal_effect = 10 * np.sin(2 * np.pi / seasonal_period * np.arange(n_periods))

# Generate trend component
trend_effect = 0.1 * np.arange(n_periods)

# Generate noise component
noise = np.random.normal(scale=1, size=n_periods)

# Combine components
prices = 50 + seasonal_effect + trend_effect + noise

# Introduce a policy effect (e.g., a reduction in prices)
prices[policy_start:] -= 10

# Create a DataFrame
its_data = pd.DataFrame({
    'time': time,
    'price': prices
})

# Set the index to the datetime
its_data.set_index('time', inplace=True)
its_data.index = pd.DatetimeIndex(its_data.index, freq='D')

# Plot the data
plt.figure(figsize=(12, 6), dpi=300)
plt.plot(its_data.index, its_data['price'], lw=3, alpha=.6)
plt.axvline(its_data.index[policy_start], color='red', linestyle='--', label='Policy␣
 ↪Start', lw=2)
plt.xlabel('Time')
plt.ylabel('Electricity Price')
plt.legend()
plt.show()
```



Since the policy has been introduced at the 100th observation, we can fint a SARIMA model on the first 100 observations, and then perform and ITS analysis on the reisudals.

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Fit a SARIMA model to the pre-intervention period
pre_policy_data = its_data.iloc[:policy_start]
sarima_model = SARIMAX(pre_policy_data['price'], order=(1, 1, 1), seasonal_order=(1,␣
↪1, 1, seasonal_period)).fit(disp=False)

# Get the residuals for the entire time series
its_data['residuals'] = its_data['price'] - sarima_model.predict(start=0, end=n_
↪periods-1, dynamic=False)

# Plot the data
plt.figure(figsize=(12, 6), dpi=300)
plt.plot(its_data.index[8:], its_data['residuals'][8:], lw=3, alpha=.6, c='k')
plt.axvline(its_data.index[policy_start], color='red', linestyle='--', label='Policy␣
↪Start', lw=2)
plt.xlabel('Time')
plt.ylabel('Residuals')
plt.legend()
plt.show()
```



As expected, given the change due to the new policy, the residuals before the policy (where we fitted the model) are quite low, while they are much higher after the policy was introduced.

We can now add the necessary dummy variables to perform the ITS analysis.

```
# Create the time after policy variable
its_data['time_after_policy'] = np.where(its_data.index >= its_data.index[policy_
 ↪start], np.arange(len(its_data)) - policy_start, 0)
its_data['policy'] = np.where(its_data.index >= its_data.index[policy_start], 1, 0)

its_data
```

```
                price  residuals  time_after_policy  policy
time
2024-01-01  50.496714  50.496714                   0       0
2024-01-02  57.780051   7.283374                   0       0
2024-01-03  60.596968   2.816929                   0       0
2024-01-04  56.161867  -4.435096                   0       0
2024-01-05  45.827009 -10.334864                   0       0
...               ...        ...                 ...     ...
2024-07-14  52.067003  -9.606227                  95       1
2024-07-15  58.716143 -10.573412                  96       1
2024-07-16  67.672040  -9.758926                  97       1
2024-07-17  69.607488  -9.616665                  98       1
2024-07-18  63.095867 -11.235913                  99       1

[200 rows x 4 columns]
```

Now, let's fit an segmented regression model, using OLS, as we did in the first part of the chapter.

```
# Define the independent variables for the ITS model
X = sm.add_constant(its_data[['time_after_policy', 'policy']])
y = its_data['residuals']

# Fit the OLS model on the residuals
```

(continues on next page)

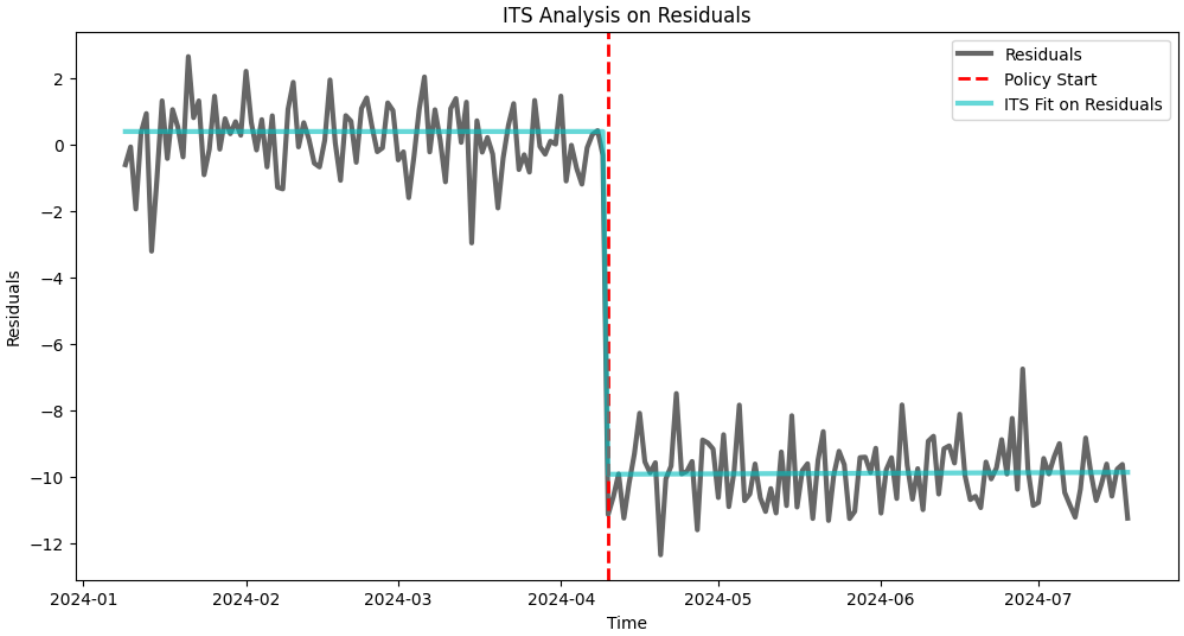
(continued from previous page)

```
its_model_on_residuals = sm.OLS(y, X).fit()
print(its_model_on_residuals.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:              residuals   R-squared:                       0.616
Model:                            OLS   Adj. R-squared:                  0.613
Method:                 Least Squares   F-statistic:                     158.3
Date:                Sun, 07 Jul 2024   Prob (F-statistic):           1.02e-41
Time:                        20:03:39   Log-Likelihood:                -563.84
No. Observations:                 200   AIC:                             1134.
Df Residuals:                     197   BIC:                             1144.
Df Model:                           2
Covariance Type:            nonrobust
=========================================================================================
                            coef    std err          t      P>|t|      [0.025      0.
↪975]
-----------------------------------------------------------------------------------------
↪--
const                     0.3999      0.409      0.979      0.329      -0.406       1.
↪206
time_after_policy         0.0006      0.014      0.041      0.967      -0.027       0.
↪029
policy                  -10.3131      0.908    -11.352      0.000     -12.105      -8.
↪522
==============================================================================
Omnibus:                      353.785   Durbin-Watson:                   0.997
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           114456.613
Skew:                           8.849   Prob(JB):                         0.00
Kurtosis:                     118.851   Cond. No.                         130.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

We can see how we were able to estimate the **effect of the policy** quite well.

```
# Plot the residuals and the ITS fit
plt.figure(figsize=(12, 6))
plt.plot(its_data.index[8:], its_data['residuals'][8:], label='Residuals', lw=3,␣
↪alpha=.6, c='k')
plt.axvline(its_data.index[policy_start], color='red', linestyle='--', label='Policy␣
↪Start', lw=2)
plt.plot(its_data.index[8:], its_model_on_residuals.predict(X)[8:], color='c', label=
↪'ITS Fit on Residuals', lw=3, alpha=.6)
plt.xlabel('Time')
plt.ylabel('Residuals')
plt.title('ITS Analysis on Residuals')
plt.legend()
plt.show()
```


ITS Analysis on Residuals
Residuals
Policy Start
ITS Fit on Residuals
Residuals
2
0
−2
−4
−6
−8
−10
−12
2024-01
2024-02
2024-03
2024-04
2024-05
2024-06
2024-07
Time

Part VI

IV. Interpretability

CHAPTER

TWENTYONE

OVERVIEW

Interpretability is crucial for understanding and explaining how models make predictions or decisions. This section introduces techniques that offer insights into the importance of different features, the effect of individual variables, and the overall behaviour of the model. These methods enhance transparency and trust in the models used in electricity markets.

Understanding and interpreting the inner workings of machine learning models is essential for several reasons:

- **Transparency**: clear insights into how a model makes decisions help build trust among stakeholders, including regulators, operators, and consumers.
- **Accountability**: interpretability allows for better scrutiny and ensures that the model's decisions can be explained and justified, which is particularly important in regulated industries like electricity markets.
- **Debugging and improvement**: by understanding which features influence the model's predictions, data scientists and engineers can identify and correct issues, improving the model's accuracy and robustness.

Despite providing some transparency, **these methods alone do not have causal guarantees** as they merely indicate how the predictions of the models are affected by changes in the covariates. Therefore, while they can show associations and effects within the model, they do not confirm causal relationships.

21.1 Content of Interpretability chapters

Chapter	Description
Partial Dependence Plots	How to visualize the relationship between a feature and the predicted outcome, while averaging out the effects of all other features.
Accumulated Local Effects	How to provide a more accurate and unbiased alternative to PDP by accounting for feature interactions.
Impulse Response Functions	How to assess the dynamic impact of a change in one variable on another over time in a time series context.
Shapley Values	How to obtain a measure of how each feature contributes to the predictions made by the model.

CHAPTER

TWENTYTWO

PARTIAL DEPENDENCE PLOTS

Partial Dependence Plots (PDPs) help us understand the relationship between a feature (or two features) and the predicted outcome of a model. The key idea is to see how changing the value of a feature affects the prediction, while **averaging out** the effects of all other features. A PDP can be very useful to unveil the nature of a relationship, for example by letting us see whether it is linear or not.

Consider a case where we fit a model to predict the wholesale electricity price using as input the temperature, the time of the day, and the day of the week. Using PDPs, we would like to explore the specific form of influence that each variable has on the predicted response. To explore that, let's now generate some data where the Price is a function of temperature, hour, and day. The effect we will generate, and that we expect to find from the PDPs are:

- **Temperature**: the typical banana shape we have seen in previous examples, where prices increase as we move away from mild temperatures. This means that the price will be higher when the temperature is either very low or very high, and lower when the temperature is moderate, forming a U-shaped curve.
- **Hour**: a sinusoidal pattern throughout the day, with two peaks and one trough. We expect higher prices in the morning (between 7 AM and 10 AM) and in the evening (between 6 PM and 10 PM) due to increased demand during these times. The prices will be lower in the afternoon (between 10 AM and 5 PM), reflecting decreased demand.
- **Day**: higher prices on Sundays compared to the rest of the week. This simulates the effect of increased electricity consumption on weekends when people are more likely to be at home.

```
import numpy as np
import pandas as pd

# Simulate data
np.random.seed(0)
n_samples = 1000
temperature = np.random.uniform(0, 40, n_samples)  # Temperature in Celsius
time_of_day = np.random.uniform(0, 24, n_samples)  # Time of day in hours
day_of_week = np.random.randint(0, 7, n_samples)  # Day of the week (0=Sunday,␣
↪6=Saturday)

price = (0.1 * (temperature - 20)**2 +  # Quadratic effect of temperature
         5 * np.sin((time_of_day - 7) * np.pi / 6) * (time_of_day >= 7) * (time_of_
↪day <= 10) +  # Smooth peak between 7 AM and 10 AM
         5 * np.sin((time_of_day - 18) * np.pi / 4) * (time_of_day >= 18) * (time_of_
↪day <= 22) +  # Smooth peak between 6 PM and 10 PM
         -5 * np.sin((time_of_day - 10) * np.pi / 7) * (time_of_day >= 10) * (time_of_
↪day <= 17) +  # Smooth trough between 10 AM and 5 PM
         7 * np.sin((time_of_day) * np.pi / 24) +  # Sinusoidal effect throughout the␣
↪day
         5 * (day_of_week == 0) +  # Higher prices on Sunday
         np.random.normal(0, 2, n_samples) * 2)  # Noise
```

(continues on next page)

(continued from previous page)

```
# Convert to DataFrame for convenience
data = pd.DataFrame({
    'Temperature': temperature,
    'Hour': time_of_day,
    'Day': day_of_week,
    'Price': price
})

# Display the first few rows
data.head()
```

```
   Temperature       Hour  Day     Price
0    21.952540  14.229126    3 -2.562450
1    28.607575   0.241529    3  7.237870
2    24.110535  11.419829    3  5.621266
3    21.795327  17.010489    5  6.220472
4    16.946192   1.055410    4 -2.435281
```

Using the **semi-parametric causal discovery methods** we discussed in the previous chapters, we might now be able to discover the true causal structure of the data-generating process. However, we will still not be able to see the **specific shape of the relationship** between a feature (or a set of features) and the predicted outcome. Now, we assume we already have some knowledge on the true causal structure, and want to use some approaches to further characterise the dependency patterns.

Imagine we fitted a model $\hat{f}$ that predicts the electricity price using the remaining three features as input variables. Let h represent the hour variable. The partial dependence function $\hat{f}(h)$ is defined as:

$$\hat{f}(h) = \mathbb{E}_{\mathbf{x}}[\hat{f}(h, \mathbf{x})] = \int \hat{f}(h, \mathbf{x}) dP(\mathbf{x}) \tag{22.1}$$

where:

- h is the feature of interest (e.g., the hour of the day). It can also be computed for a set of features instead of a single one.
- $\mathbf{x}$ is the vector of all other features in the dataset except h.
- $\hat{f}$ is the model, and $\hat{f}(h, \mathbf{x})$ is the prediction made by the model for given h and $\mathbf{x}$ values.
- $\mathbb{E}_{\mathbf{x}}$ denotes the expectation over the marginal distribution of $\mathbf{x}$.

The **partial dependence function** shows how the model's prediction changes when h changes, keeping the other features constant. Taking the expected value of the model's prediction over the distribution of $\mathbf{x}$ effectively averages out the effects of all other features. Indeed, the The integral indicates that we are integrating (or averaging) the model's predictions over the probability distribution of the features in $\mathbf{x}$. This integration helps in isolating the effect of h on the predicted outcome.

In practice, we estimate $\hat{f}(h)$ using the average prediction over the dataset:

$$\hat{f}(h) \approx \frac{1}{n} \sum_{i=1}^{n} \hat{f}(h, \mathbf{x}_i) \tag{22.2}$$

where n is the number of instances in the dataset, and $\mathbf{x}_i$ is the ith observation of the covariates. This sum basically calculates the average prediction of the model when h is fixed at a specific value, and $\mathbf{x}$ iterates through all the observed values in the dataset. Here is what we would do in practice:

1. **Fix the feature of interest**: pick a specific value of the feature h (e.g., Hour = 8 AM).

2. **Pair it with all other features**: for each data point in the dataset, replace the value of the feature h with the fixed value (e.g., set Hour to 8 AM for all instances), while keeping the other feature values $\mathbf{x}$ as they are.
3. **Make predictions**: use the fitted model to make predictions for all these modified data points.
4. **Average these predictions**: calculate the average of these predictions. **This average actually represents the partial dependence value $\hat{f}(h, \mathbf{x})$.

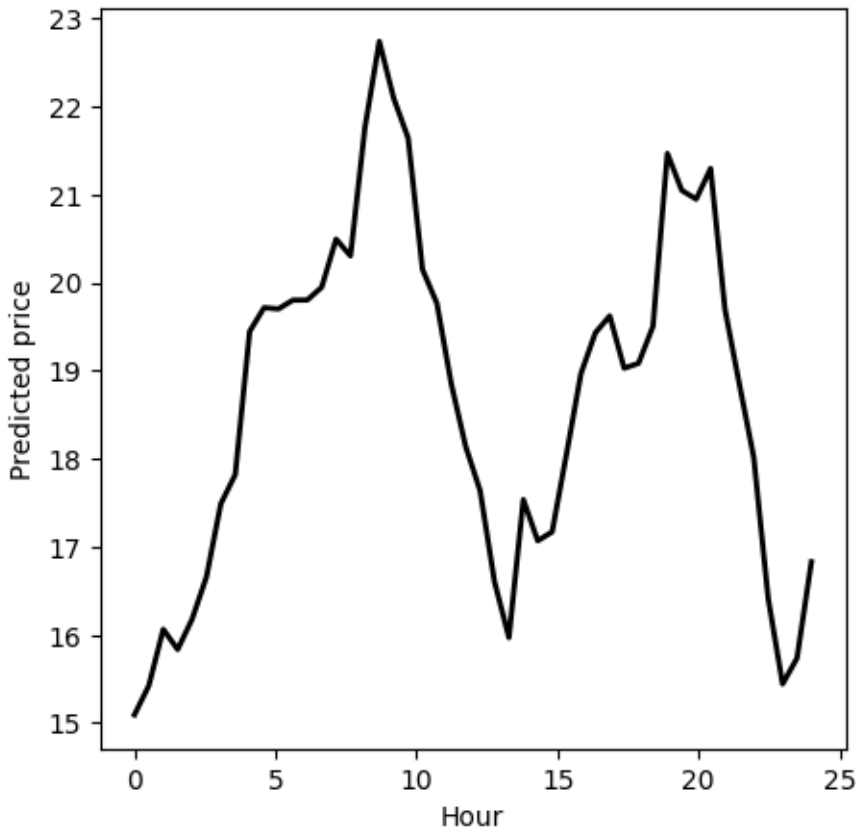
By iteratively fixing h at specific values and averaging over all the remaining features, we are actually trying to **isolate the influence** of changing hours, while keeping the other values constant.

To see this in practice, let's manually implement the PDP for the "Hour" feature. This will illustrate the steps involved in calculating and plotting a PDP. Here is a simple implementation of a function to estimate the **partial dependence function**:

```
def calculate_pdp(feature_name, feature_values, X, model):
    pdp_values = []
    for value in feature_values: # Step 1
        X_temp = X.copy()
        X_temp[feature_name] = value # Step 2
        predictions = model.predict(X_temp) # Step 3
        pdp_values.append(np.mean(predictions))  # Step 4
    return pdp_values
```

Let's now apply it to a random forest regression model fitted to the data we generated before:

```
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor

# Split data into features and target
x = data[['Temperature', 'Hour', 'Day']]
y = data['Price']

# Train a Random Forest Regressor
rf = RandomForestRegressor(n_estimators=10, random_state=0)
rf.fit(x, y)

# Calculate PDP for the "Hour" feature
hour_values = np.linspace(0, 24, 48)  # Range of hour values to evaluate
pdp_hour = calculate_pdp(feature_name='Hour', feature_values=hour_values, X=x,␣
 ↪model=rf)

# Plot the manually calculated PDP
plt.figure(figsize=(5, 5))
plt.plot(hour_values, pdp_hour, color='k', linewidth=2)
plt.xlabel('Hour')
plt.ylabel('Predicted price')
plt.show()
```



We can see how the PDP shows the pattern we expected, as the time of the day affects the price through the peak morning and evening hours. Now, we can apply the function we implemented to the remaining features too. To obtain more reliable estimates, we can **repeat the procedure many times**, and plot the mean dependence lines, along with some **Bootstrap confidence intervals**.

```
# Define the range of values for each feature
temperature_values = np.linspace(0, 35, 70)
hour_values = np.linspace(0, 24, 48)
day_values = np.arange(7)

# Calculate PDPs with bootstrapping
n_bootstraps = 100
pdp_temperature_bootstraps = []
pdp_hour_bootstraps = []
pdp_day_bootstraps = []

for _ in range(n_bootstraps):
    # Bootstrap sample
    indices = np.random.choice(range(n_samples), size=n_samples, replace=True)
    x_bootstrap = x.iloc[indices]
    y_bootstrap = y.iloc[indices]

    # Train model on bootstrap sample
    model_bootstrap = RandomForestRegressor(n_estimators=10, random_state=0)
    model_bootstrap.fit(x_bootstrap, y_bootstrap)

    # Calculate PDPs for each feature
```

(continues on next page)

(continued from previous page)

```
    pdp_temperature = calculate_pdp('Temperature', temperature_values, x, model_
↪bootstrap)
    pdp_hour = calculate_pdp('Hour', hour_values, x, model_bootstrap)
    pdp_day = calculate_pdp('Day', day_values, x, model_bootstrap)

    pdp_temperature_bootstraps.append(pdp_temperature)
    pdp_hour_bootstraps.append(pdp_hour)
    pdp_day_bootstraps.append(pdp_day)
```

We basically repeated the estimation procedure 100 times. Now, we can use a basic approach and simply take the empirical 2.5% and 97.5% percentiles to obtain a 95% confidence interval for the partial dependence functions.

```
pdp_temperature_bootstraps = np.array(pdp_temperature_bootstraps)
pdp_hour_bootstraps = np.array(pdp_hour_bootstraps)
pdp_day_bootstraps = np.array(pdp_day_bootstraps)

# Compute mean and confidence intervals
mean_pdp_temperature = np.mean(pdp_temperature_bootstraps, axis=0)
mean_pdp_hour = np.mean(pdp_hour_bootstraps, axis=0)
mean_pdp_day = np.mean(pdp_day_bootstraps, axis=0)
ci_temperature = np.percentile(pdp_temperature_bootstraps, [2.5, 97.5], axis=0)
ci_hour = np.percentile(pdp_hour_bootstraps, [2.5, 97.5], axis=0)
ci_day = np.percentile(pdp_day_bootstraps, [2.5, 97.5], axis=0)
```

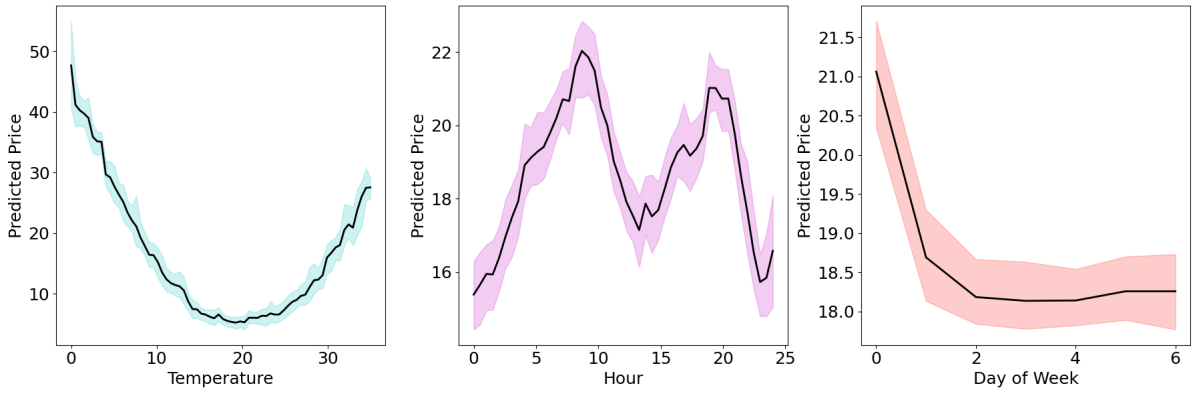
Let's now plot the result!

```
# Plot PDPs with confidence intervals
fig, ax = plt.subplots(1, 3, figsize=(18, 6))

fontsize = 18  # Set the font size for all labels

# Temperature
ax[0].plot(temperature_values, mean_pdp_temperature, label='Partial Dependence',␣
↪color='k', linewidth=2)
ax[0].fill_between(temperature_values, ci_temperature[0], ci_temperature[1], color='c
↪', alpha=0.2)
ax[0].set_xlabel('Temperature', fontsize=fontsize)
ax[0].set_ylabel('Predicted Price', fontsize=fontsize)
ax[0].tick_params(axis='both', which='major', labelsize=fontsize)

# Hour
ax[1].plot(hour_values, mean_pdp_hour, label='Partial Dependence', color='k',␣
↪linewidth=2)
ax[1].fill_between(hour_values, ci_hour[0], ci_hour[1], color='m', alpha=0.2)
ax[1].set_xlabel('Hour', fontsize=fontsize)
ax[1].set_ylabel('Predicted Price', fontsize=fontsize)
ax[1].tick_params(axis='both', which='major', labelsize=fontsize)

# Day
ax[2].plot(day_values, mean_pdp_day, label='Partial Dependence', color='k',␣
↪linewidth=2)
ax[2].fill_between(day_values, ci_day[0], ci_day[1], color='r', alpha=0.2)
ax[2].set_xlabel('Day of Week', fontsize=fontsize)
ax[2].set_ylabel('Predicted Price', fontsize=fontsize)
ax[2].tick_params(axis='both', which='major', labelsize=fontsize)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



Finally, we can also fix the level of **two variables**, and average over the remaining ones, to see the combined effect on the predicted response. In this case, we can see the results in the form of **contour plots**. Here is a simple extension of the previous function, to compute PDP for two variables:

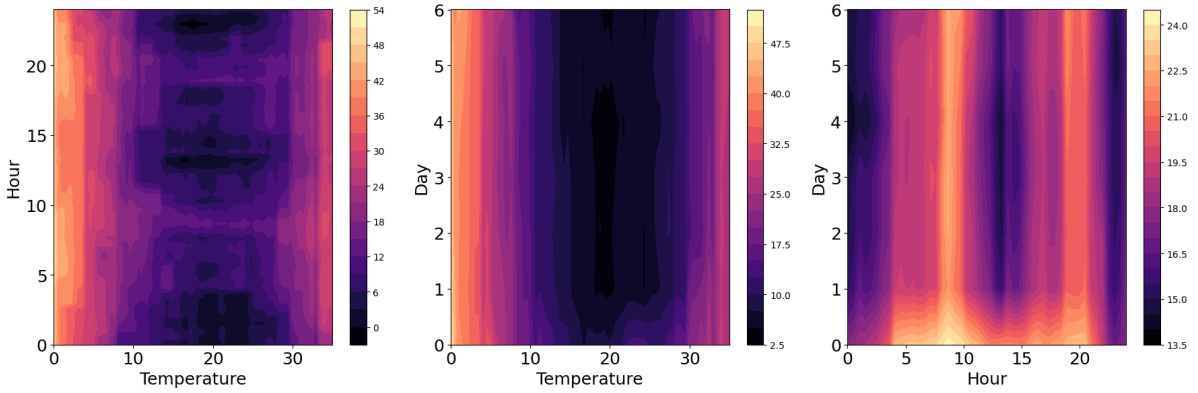
```
def calculate_pdp_two_features(feature_names, feature_values1, feature_values2, X,␣
↪model):
    pdp_values = np.zeros((len(feature_values1), len(feature_values2)))
    for i, value1 in enumerate(feature_values1):
        for j, value2 in enumerate(feature_values2):
            X_temp = X.copy()
            X_temp[feature_names[0]] = value1
            X_temp[feature_names[1]] = value2
            predictions = model.predict(X_temp)
            pdp_values[i, j] = np.mean(predictions)
    return pdp_values
```

```
# Calculate PDPs for the feature combinations
pdp_temperature_hour = calculate_pdp_two_features(['Temperature', 'Hour'],␣
↪temperature_values, hour_values, x, rf)
pdp_temperature_day = calculate_pdp_two_features(['Temperature', 'Day'], temperature_
↪values, day_values, x, rf)
pdp_hour_day = calculate_pdp_two_features(['Hour', 'Day'], hour_values, day_values, x,
↪ rf)

# Plot the PDPs as contour plots
fig, ax = plt.subplots(1, 3, figsize=(18, 6), dpi=100)

# Temperature and Hour
contour1 = ax[0].contourf(temperature_values, hour_values, pdp_temperature_hour.T,␣
↪cmap='magma', levels=20)
fig.colorbar(contour1, ax=ax[0])
ax[0].set_xlabel('Temperature', fontsize=fontsize)
ax[0].set_ylabel('Hour', fontsize=fontsize)
ax[0].tick_params(axis='both', which='major', labelsize=fontsize)

# Temperature and Day
contour2 = ax[1].contourf(temperature_values, day_values, pdp_temperature_day.T, cmap=
```

(continues on next page)

(continued from previous page)

```
 ↪'magma', levels=20)
fig.colorbar(contour2, ax=ax[1])
ax[1].set_xlabel('Temperature', fontsize=fontsize)
ax[1].set_ylabel('Day', fontsize=fontsize)
ax[1].tick_params(axis='both', which='major', labelsize=fontsize)

# Hour and Day
contour3 = ax[2].contourf(hour_values, day_values, pdp_hour_day.T, cmap='magma',␣
 ↪levels=20)
fig.colorbar(contour3, ax=ax[2])
ax[2].set_xlabel('Hour', fontsize=fontsize)
ax[2].set_ylabel('Day', fontsize=fontsize)
ax[2].tick_params(axis='both', which='major', labelsize=fontsize)

plt.tight_layout()
plt.show()
```



CHAPTER

TWENTYTHREE

ACCUMULATED LOCAL EFFECTS

Just like PDPs, accumulated local effects (ALE) plots are designed to investigate the impact of varying the levels of one feature on the predicted response. The main benefit of ALE plots is that they are deemed most effective in the presence of **correlated variables**.

Issues with PDPs: when features in a dataset are correlated, PDPs can be misleading. Since PDPs average predictions over the marginal distribution of the features, and do not consider the correlation patterns, they can sometimes result in **unrealistic data points**. For instance, if we predict electricity prices based on temperature, time of day, and day of the week, PDPs might consider unlikely combinations, such as high electricity prices at 3 AM on weekdays.

There are two potential solutions to this issue:

1. **Averaging over the conditional distribution**, instead of the marginal, thereby excluding the inclusion of unrealistic data points.
2. **Averaging over differences in predictions**, instead of averaging directly the predictions.

The second option is the one used by ALE plots, which compute **differences in predictions within small intervals**, ensuring the estimates are based on realistic data points. This approach makes ALE plots more reliable when features are correlated.

In practice, the ALE for the jth feature, x_j, is estimated by:

$$\hat{f}_{j,\text{ALE}}(x) = \sum_{k=1}^{k(x)} \frac{1}{n_j(k)} \sum_{i:x_j^{(i)} \in N_j(k)} \left(\hat{f}(z_{k,j}, x_{-j}^{(i)}) - \hat{f}(z_{k-1,j}, x_{-j}^{(i)}) \right) \tag{23.1}$$

where $N_j(k)$ is the kth interval for feature x_j, $z_{k,j}$ is the upper boundary of the kth interval, $x_{-j}^{(i)}$ are all the other features except x_j, for the ith data point, and $n_j(k)$ is the number of data points in the kth interval.

The previous equation represents the following steps:

1. **Divide the feature into intervals**: split the range of the variable of interest h into m intervals, often based on quantiles to ensure an equal number of data points in each interval.
2. **Compute local effects**: for each interval, compute the difference in model predictions when h is replaced by the interval's lower and upper bounds.
3. **Accumulate local effects**: sum up the local effects within each interval to obtain the accumulated effect.
4. **Center the ALE plot**: subtract the mean accumulated effect to center the plot around zero.

Thus, we are simply calculating the average difference in predictions when the feature of interest is changed from the lower to the upper boundary of the interval, and accumulating these differences across all intervals.

This process is expected to perform better than PDP for correlated features because it accounts for the dependencies between features, ensuring that the differences in predictions are computed based on realistic data points. By focusing on local changes within intervals, ALE plots provide a more accurate representation of feature effects in the presence of correlations.

Here is a simple manual implementation of ALE:

```
import numpy as np

def calculate_ale(feature_name, feature_values, X, model, num_intervals=10):
    interval_edges = np.linspace(min(feature_values), max(feature_values), num_
 ↪intervals + 1)
    ale_values = np.zeros(num_intervals)
    X_temp = X.copy()

    for i in range(num_intervals):
        lower = interval_edges[i]
        upper = interval_edges[i + 1]

        in_interval = (X[feature_name] >= lower) & (X[feature_name] < upper)
        if in_interval.sum() == 0:
            continue

        X_temp[feature_name] = lower
        preds_lower = model.predict(X_temp)

        X_temp[feature_name] = upper
        preds_upper = model.predict(X_temp)

        ale_values[i] = np.mean(preds_upper[in_interval] - preds_lower[in_interval])

    ale_accumulated = np.cumsum(ale_values)
    ale_accumulated -= np.mean(ale_accumulated)

    return interval_edges[:-1], ale_accumulated
```

Let's now generate some data to show how they work in practice. For simplicity, we will use the same example sued to show PDPs.

```
import pandas as pd

# Simulate data
np.random.seed(0)
n_samples = 1000
temperature = np.random.uniform(0, 40, n_samples)  # Temperature in Celsius
time_of_day = np.random.uniform(0, 24, n_samples)  # Time of day in hours
day_of_week = np.random.randint(0, 7, n_samples)  # Day of the week (0=Sunday,␣
 ↪6=Saturday)

# Simulate electricity price with desired patterns
price = (0.2 * (temperature - 20)**2 +  # Quadratic effect of temperature
         5 * np.sin((time_of_day - 7) * np.pi / 6) * (time_of_day >= 7) * (time_of_
 ↪day <= 10) +  # Peak between 7 AM and 10 AM
         5 * np.sin((time_of_day - 18) * np.pi / 4) * (time_of_day >= 18) * (time_of_
 ↪day <= 22) +  # Peak between 6 PM and 10 PM
         -5 * np.sin((time_of_day - 10) * np.pi / 7) * (time_of_day >= 10) * (time_of_
 ↪day <= 17) +  # Trough between 10 AM and 5 PM
         7 * np.sin((time_of_day) * np.pi / 24) +  # Sinusoidal effect throughout the␣
 ↪day
         2 * (1 - np.cos(day_of_week * np.pi / 7)) +  # Higher prices on Sunday with␣
 ↪smooth transition
         np.random.normal(0, 2, n_samples))  # Noise
```

(continues on next page)

(continued from previous page)

```
# Convert to DataFrame for convenience
data = pd.DataFrame({
    'Temperature': temperature,
    'Hour': time_of_day,
    'Day': day_of_week,
    'Price': price
})

# Display the first few rows
data.head()
```

```
   Temperature       Hour  Day      Price
0    21.952540  14.229126    3   1.830310
1    28.607575   0.241529    3  16.398082
2    24.110535  11.419829    3   8.902574
3    21.795327  17.010489    5   9.614504
4    16.946192   1.055410    4   3.108261
```

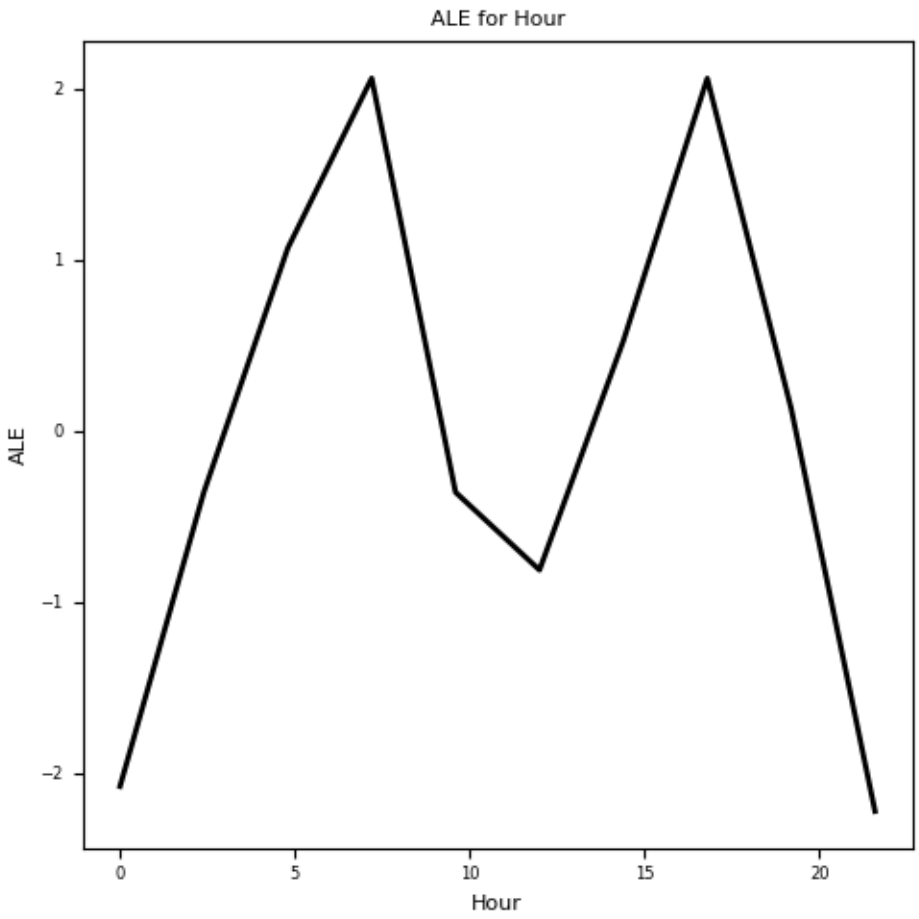
We can now fit a regression model (e.g., random forest), and compute the ALE:

```
from sklearn.ensemble import RandomForestRegressor
import matplotlib.pyplot as plt


# Split data into features and target
X = data[['Temperature', 'Hour', 'Day']]
y = data['Price']

# Train a Random Forest Regressor
model = RandomForestRegressor(n_estimators=10, random_state=0)
model.fit(X, y)

# Calculate ALE for "Hour" feature
hour_values = np.linspace(0, 24, 48)
intervals_hour, ale_hour = calculate_ale('Hour', hour_values, X, model)
```

We now used intervals of 30 minutes, which are typically referred to as **settlement periods (SPs)** in the electricity markets.

Let's plot the resulting ALE plot:

```
# Plot the ALE for "Hour" feature
plt.figure(figsize=(5, 5))
plt.plot(intervals_hour, ale_hour, color='k', linewidth=2)
plt.title('ALE for Hour', fontsize=8)
plt.xlabel('Hour', fontsize=8)
plt.ylabel('ALE', fontsize=8)
plt.xticks(fontsize=6)
plt.yticks(fontsize=6)
plt.tight_layout()
plt.show()
```



As we did for the PDPs, we can run the procedure multiple times for each feature to construct **Bootstrap confidence intervals** for the estimated ALE plots.

```
# Define the range of values for each feature
temperature_values = np.linspace(0, 35, 70)
hour_values = np.linspace(0, 24, 48)
day_values = np.arange(7)

# Calculate ALEs with bootstrapping
n_bootstraps = 100
ale_temperature_bootstraps = []
ale_hour_bootstraps = []
ale_day_bootstraps = []

for _ in range(n_bootstraps):
    # Bootstrap sample
    indices = np.random.choice(range(n_samples), size=n_samples, replace=True)
    X_bootstrap = X.iloc[indices]
    y_bootstrap = y.iloc[indices]

    # Train model on bootstrap sample
    model_bootstrap = RandomForestRegressor(n_estimators=10, random_state=0)
    model_bootstrap.fit(X_bootstrap, y_bootstrap)

    # Calculate ALEs for each feature
```

(continues on next page)

(continued from previous page)

```
    intervals_temperature, ale_temperature = calculate_ale('Temperature', temperature_
↪values, X, model_bootstrap)
    intervals_hour, ale_hour = calculate_ale('Hour', hour_values, X, model_bootstrap)
    intervals_day, ale_day = calculate_ale('Day', day_values, X, model_bootstrap)

    ale_temperature_bootstraps.append(ale_temperature)
    ale_hour_bootstraps.append(ale_hour)
    ale_day_bootstraps.append(ale_day)

# Convert results to numpy arrays for easier manipulation
ale_temperature_bootstraps = np.array(ale_temperature_bootstraps)
ale_hour_bootstraps = np.array(ale_hour_bootstraps)
ale_day_bootstraps = np.array(ale_day_bootstraps)

# Compute mean and confidence intervals
mean_ale_temperature = np.mean(ale_temperature_bootstraps, axis=0)
mean_ale_hour = np.mean(ale_hour_bootstraps, axis=0)
mean_ale_day = np.mean(ale_day_bootstraps, axis=0)
ci_temperature = np.percentile(ale_temperature_bootstraps, [2.5, 97.5], axis=0)
ci_hour = np.percentile(ale_hour_bootstraps, [2.5, 97.5], axis=0)
ci_day = np.percentile(ale_day_bootstraps, [2.5, 97.5], axis=0)
```

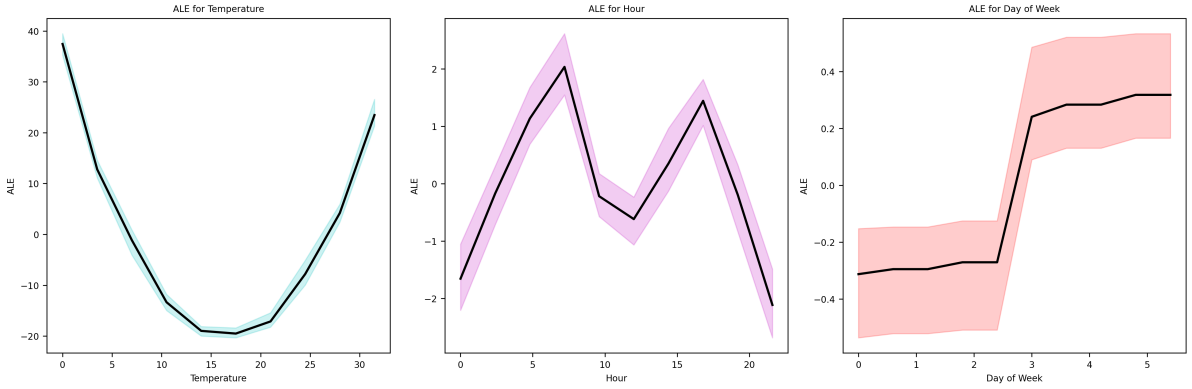
Let's plot the results

```
# Plot ALEs with confidence intervals
fig, ax = plt.subplots(1, 3, figsize=(15, 5), dpi=300)

fontsize = 8  # Set the font size for all labels

# Temperature
ax[0].plot(intervals_temperature, mean_ale_temperature, label='ALE', color='k',␣
↪linewidth=2)
ax[0].fill_between(intervals_temperature, ci_temperature[0], ci_temperature[1], color=
↪'c', alpha=0.2)
ax[0].set_title('ALE for Temperature', fontsize=fontsize)
ax[0].set_xlabel('Temperature', fontsize=fontsize)
ax[0].set_ylabel('ALE', fontsize=fontsize)
ax[0].tick_params(axis='both', which='major', labelsize=fontsize)

# Hour
ax[1].plot(intervals_hour, mean_ale_hour, label='ALE', color='k', linewidth=2)
ax[1].fill_between(intervals_hour, ci_hour[0], ci_hour[1], color='m', alpha=0.2)
ax[1].set_title('ALE for Hour', fontsize=fontsize)
ax[1].set_xlabel('Hour', fontsize=fontsize)
ax[1].set_ylabel('ALE', fontsize=fontsize)
ax[1].tick_params(axis='both', which='major', labelsize=fontsize)

# Day
ax[2].plot(intervals_day, mean_ale_day, label='ALE', color='k', linewidth=2)
ax[2].fill_between(intervals_day, ci_day[0], ci_day[1], color='r', alpha=0.2)
ax[2].set_title('ALE for Day of Week', fontsize=fontsize)
ax[2].set_xlabel('Day of Week', fontsize=fontsize)
ax[2].set_ylabel('ALE', fontsize=fontsize)
ax[2].tick_params(axis='both', which='major', labelsize=fontsize)

plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



As we can see, the effects of Hour and Temperature variables have been properly identified. To try and see the potential benefits ovet PDPs, we could generated correlated data and see if ALE plots offer improvements,

CHAPTER

TWENTYFOUR

IMPULSE RESPONSE FUNCTIONS

Conversely from PDPs and ALEs, impulse response functions (IRFs) are particularly suited for **time series**. They are a key tool in time series analysis, particularly in the context of **vector autoregressive (VAR) models**. They describe the reaction of the system (i.e., the time series) to a temporary shock in one of the variables. In the context of electricity markets, IRFs can help us understand how a shock in one factor, such as temperature or demand, impacts electricity prices over time.

In electricity markets, understanding the dynamic relationships between different factors is crucial. For example, a sudden increase in temperature might lead to higher electricity demand due to air conditioning, which in turn could increase electricity prices. IRFs allow us to quantify and visualize these dynamic responses, providing insights into the temporal effects of shocks on the market.

Consider a VAR(p) model with k variables. The order p determines the number of lagged observations included in the model, reflecting the extent to which past values influence the current value. The model is given by:

$$\mathbf{y}_t = \mathbf{A}_1\mathbf{y}_{t-1} + \mathbf{A}_2\mathbf{y}_{t-2} + \cdots + \mathbf{A}_p\mathbf{y}_{t-p} + \mathbf{u}_t \tag{24.1}$$

where $\mathbf{y}_t$ is a $k \times 1$ vector of time series variables at time t, $\mathbf{A}_i$ are the $k \times k$ coefficient matrices, and $\mathbf{u}_t$ is a $k \times 1$ vector of error terms (shocks).

An impulse response function measures the effect of a one-time shock to one of the variables in $\mathbf{u}_t$ on the current and future values of the variables in $\mathbf{y}_t$.

The estimation of the IRFs is divided into two main steps:

1. **Fit a VAR model**: estimate the coefficients $\mathbf{A}_i$ of the VAR model using the time series data.
2. **Impulse response calculation**: compute the IRFs by iterating the VAR model forward in time, starting from the shock.

In practice, measuring the effect of a shock means to:

1. **Generate a shock**: introduce a one-time shock to one of the variables in the error term vector $\mathbf{u}_t$.
2. **Propagate the shock**: use the estimated VAR model to propagate the shock through the system, calculating the response of each variable in $\mathbf{y}_t$ over time.
3. **Visualize the response**: plot the response of the variables to the shock over time, illustrating the dynamic effects of the shock on the system.

The response of each variable to the shock can be tracked over multiple time periods, providing a comprehensive view of the temporal dynamics within the system. This approach helps in understanding how an initial disturbance affects the system both immediately and in the future, offering valuable insights into the behavior and interactions of the variables in the model.

Let's clarify what a shock is in practice: In a VAR model, the error term (often referred to as "shock" or "innovation") represents unexpected changes or disturbances in the system. These are components of the time series that cannot be explained by the past values of the variables in the model. For instance, in an electricity market, the error term for

demand could represent unexpected changes in electricity demand due to sudden weather changes, unforeseen industrial activities, or other random events. When we introduce a shock to one of the error terms in the VAR model, we simulate an unexpected change in one of the variables (the one corresponding to the same index of the shock). The IRF then shows us how this shock propagates through the system over time.

Here's how it works step-by-step:

1. **Initial shock**: suppose we introduce a shock to the error term for electricity demand. This means we simulate an unexpected increase or decrease in demand at time t.
2. **Immediate effect**: this shock directly impacts the demand variable at time t. Since the demand variable is part of the VAR model, this immediate change is reflected in the model's equation for demand.
3. **Propagation through the system**: the change in demand affects future values of demand (due to the autoregressive nature of the model) and can also affect other variables in the system, such as price. This is because the future values of all variables in a VAR model depend on past values of all variables. For example, an increase in demand could lead to an increase in electricity prices due to higher demand pressures. This effect is captured in the VAR model coefficients.
4. **Dynamic responses**: by iterating the VAR model forward in time, the IRF shows how the initial shock to demand affects both demand and price (and potentially other variables) over multiple future time periods.

Practical example: consider a VAR(2) model with two variables, demand D and price P. The model equations might look like this:

$$\begin{aligned} D_t &= a_{10} + a_{11}D_{t-1} + a_{12}P_{t-1} + a_{13}D_{t-2} + a_{14}P_{t-2} + u_{Dt} \\ P_t &= b_{10} + b_{11}D_{t-1} + b_{12}P_{t-1} + b_{13}D_{t-2} + b_{14}P_{t-2} + u_{Pt} \end{aligned} \tag{24.2}$$

where D_t and P_t are the demand and price at time t, a_{ij} and b_{ij} are the model coefficients, u_{Dt} and u_{Pt} are the error terms (shocks) for demand and price at time t.

Step-by-step simulation of a shock:

1. **Introduce a shock**: suppose we introduce a positive shock to u_{Dt} at time t, representing an unexpected increase in demand.
2. **Immediate Effect**: this shock increases D_t, as u_{Dt} is part of the equation for D_t.
3. **Model Iteration**: at time $t+1$, the increased demand D_t affects both D_{t+1} and P_{t+1} through the VAR model equations. The model will show how the increase in D_t (demand) influences P_{t+1} (price) and D_{t+1} (future demand).
4. **Propagation**: this process continues iteratively, showing how the initial shock propagates through time and affects the system.

In summary, the noise or error terms in a VAR model allow us to introduce and **simulate unexpected changes in the variables**. By analyzing how these shocks affect the system over time using IRFs, we can understand the dynamic interactions between variables. This approach provides valuable insights into the **temporal dependencies** and relationships within the system, which are crucial for effective modeling and forecasting in electricity markets.

Let's now consider a practical example in Python, by generating some time-dependent data related to temperature, demand, and price.

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

# Simulate data
np.random.seed(0)
n_samples = 100
time_index = pd.date_range(start='1/1/2020', periods=n_samples, freq='D')
temperature = np.sin(np.linspace(0, 3 * np.pi, n_samples)) + np.random.normal(0, 0.5,␣
```

(continues on next page)

(continued from previous page)

```
↪n_samples)
demand = np.sin(np.linspace(0, 2 * np.pi, n_samples) + np.pi/4) + np.random.normal(0,␣
↪0.5, n_samples)
price = 0.5 * temperature + 0.3 * demand + np.random.normal(0, 0.2, n_samples)

# Create a DataFrame
data = pd.DataFrame({'Temperature': temperature, 'Demand': demand, 'Price': price},␣
↪index=time_index)

# Display the first few rows
data.head()
```

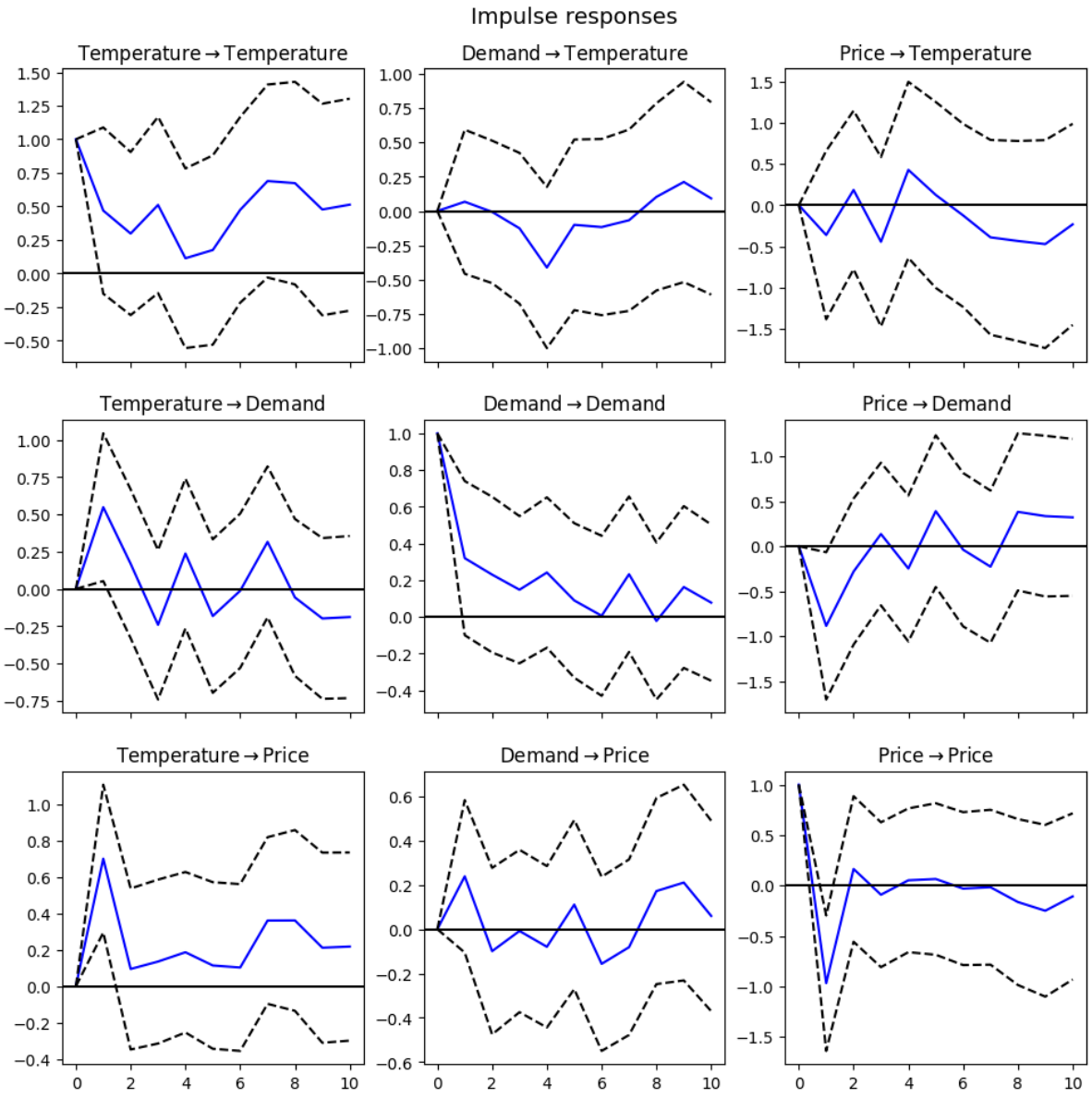
	Temperature	Demand	Price
2020-01-01	0.882026	1.648682	0.861781
2020-01-02	0.295135	0.076651	0.122687
2020-01-03	0.678620	0.155690	0.605949
2020-01-04	1.402179	1.312848	1.225997
2020-01-05	1.305441	0.275470	0.863388

We now proceed with **step 1 of the analysis**, which means to fit a VAR model to the data.

```
from statsmodels.tsa.api import VAR

# Fit a VAR model
model = VAR(data)
results = model.fit(maxlags=15, ic='aic')

# Print summary of the model
# results.summary()
```

We can now move to **step 2**, by introducing shocks in each variable and observing the temporal dependencies and connections among them.

```
# Compute IRFs
irf = results.irf(10)  # Compute IRFs for 10 periods ahead

# Plot IRFs
irf.plot(orth=False)
plt.show()
```



Each subplot represents the response of one variable to a one-time shock in another variable (or itself). The blue line indicates the estimated impulse response, while the dashed lines represent the confidence intervals. Generally, a shock to a variable has a significant immediate effect on that variable, which then diminishes over time. Temperature shocks tend to increase demand and price initially, aligning with the expectation that higher temperatures drive up electricity demand and prices. Demand shocks significantly affect both demand and price, reflecting the typical market behavior where increased demand leads to higher prices. Price shocks also impact demand and price, though the effects stabilize over time. These IRFs provide insights into the dynamic interactions in the electricity market, showing how shocks propagate through the system.

CHAPTER

TWENTYFIVE

SHAPLEY ADDITIVE EXPLANATIONS

25.1 Shapley values

Shapley additive explanations (SHAP) is a method to explain individual predictions of machine learning models by attributing the output of a model to its input features. It is based on cooperative game theory and provides a way to fairly distribute the "payout" (prediction) among the "players" (features).

In cooperative game theory, the Shapley value is used to distribute the total gains to players based on their contributions. For a set of features N and a prediction function f, the Shapley value ϕ_i for feature i is defined as:

$$\phi_i(f) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [f(S \cup \{i\}) - f(S)] \tag{25.1}$$

where:

- S is a subset of features not including i.
- $f(S)$ is the prediction function evaluated with features in S present and others absent.
- $|S|$ is the number of features in subset S.
- $|N|$ is the total number of features.

This formula calculates the **contribution of a specific feature to the model's prediction** by averaging how much the prediction changes when the feature is added to all possible combinations of other features, with each combination weighted to ensure fairness.

A simple example to show how it works: power plants as features

Let's make an example to understand how this formula works in practice. Imagine a scenario where different power plants contribute to the total electricity generation. We want to fairly distribute the revenue generated from selling electricity among these power plants based on their contribution to the total electricity generation.

Let's say we have power plants A, B, and C. The total revenue from selling electricity is influenced by the contribution of each power plant. Then, we have that

- N is the set of all power plants, $N = \{A, B, C\}$.
- S is a subset of power plants.
- $f(S)$ is the revenue generated by the subset S of power plants.
- $\phi_i(f)$ is the Shapley value for power plant i, representing its fair share of the total revenue.

Step-by-step procedure:

1. **Identify all subsets without plant** A: the possible subsets S without A are: {}, {B}, {C}, and {B, C}.

2. **Marginal contribution of adding plant** A: for each subset S, calculate the additional revenue generated by adding plant A to S.
3. **Calculate marginal contributions**: the impact of adding a specific plant (or feature) to a subset of other plants (or features) in terms of the change in the power production (or model's prediction). Essentially, they quantify how much the outcome changes when a particular plant is added to different groups of other plants. We will compute one marginsal contribution for each subset:
    - For {}: $f(\{A\}) - f(\{\})$
    - For {B}: $f(\{A, B\}) - f(\{B\})$
    - For {C}: $f(\{A, C\}) - f(\{C\})$
    - For {B, C}: $f(\{A, B, C\}) - f(\{B, C\})$
4. **Weight Each Contribution**: the weight for each marginal contribution is

$$\frac{|S|!(|N| - |S| - 1)!}{|N|!} \tag{25.2}$$

    It ensures that the contribution of each feature is averaged fairly across all possible combinations of features:
    - $|S|!$ is the factorial of the size of the subset S, and represents the number of ways to arrange the features in subset S.
    - $(|N| - |S| - 1)!$ is the factorial of the number of features not in subset S or the feature i being evaluated, and represents the number of ways to arrange the remaining features excluding the feature i.
    - $|N|!$ is the factorial of the total number of features, and represents the total number of ways to arrange all the features.

Let's compute the Shapley value for plant A:

$$\phi_A(f) = \sum_{S \subseteq N \setminus \{A\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [f(S \cup \{A\}) - f(S)] \tag{25.3}$$

Where:

- $|N| = 3$ (total number of power plants)
- $|S|$ is the size of subset S

Breaking it down for each subset S:

1. **For** $S = \{\}$:
    - $|S| = 0$
    - Marginal contribution: $f(\{A\}) - f(\{\})$
    - Weight: $\frac{0!(3-0-1)!}{3!} = \frac{1 \cdot 2}{6} = \frac{1}{3}$
2. **For** $S = \{B\}$:
    - $|S| = 1$
    - Marginal contribution: $f(\{A, B\}) - f(\{B\})$
    - Weight: $\frac{1!(3-1-1)!}{3!} = \frac{1 \cdot 1}{6} = \frac{1}{6}$
3. **For** $S = \{C\}$:
    - $|S| = 1$
    - Marginal contribution: $f(\{A, C\}) - f(\{C\})$

- Weight: $\frac{1!(3-1-1)!}{3!} = \frac{1 \cdot 1}{6} = \frac{1}{6}$

4. **For** $S = \{B, C\}$:
   - $|S| = 2$
   - Marginal contribution: $f(\{A, B, C\}) - f(\{B, C\})$
   - Weight: $\frac{2!(3-2-1)!}{3!} = \frac{2 \cdot 1}{6} = \frac{1}{3}$

Once we have the weights for the possible subsets (or coalitions), we can **sum up the contributions**:

$$\phi_A(f) = \frac{1}{3}\left[f(\{A\}) - f(\{\})\right] + \frac{1}{6}\left[f(\{A, B\}) - f(\{B\})\right] + \frac{1}{6}\left[f(\{A, C\}) - f(\{C\})\right] + \frac{1}{3}\left[f(\{A, B, C\}) - f(\{B, C\})\right] \tag{25.4}$$

The Shapley value $\phi_A(f)$ represents the fair share of revenue that power plant A should receive based on its contribution to the total revenue generated, considering all possible combinations of power plants. By applying this process, we can similarly calculate the Shapley values for power plants B and C, ensuring that each plant receives a fair share of the revenue based on its contribution to the overall electricity generation.

25.2 Connection between Shapley values and SHAP

Shapley values come from cooperative game theory and provide a way to fairly distribute the total gain (e.g., revenue) among players (e.g., power plants). The Shapley value for each player (feature) is calculated based on the average marginal contribution of that player across all possible subsets of players. Instead, SHAP adapts Shapley values to explain the output of machine learning models. It provides a way to understand how each feature contributes to an individual prediction. SHAP uses the same principles as Shapley values but applies them in the context of model predictions. We just have to consider the prediction of a machine learning model as the "total gain" in a cooperative game, and the features of the model are the "players" contributing to the prediction.

25.3 Practical SHAP implementation

1. **Model training**: train your machine learning model on your data.
2. **SHAP explainer**: use a SHAP explainer to compute SHAP values for the trained model. SHAP explainers are designed to efficiently compute the Shapley values for machine learning models. The SHAP library provides different types of explainers such as KernelExplainer, TreeExplainer, and DeepExplainer, depending on the model type.
3. **Compute SHAP values**: for a given instance x, compute the SHAP values for all features. These values indicate the contribution of each feature to the prediction.

Let's now **generate some data** to apply SHAP, as we did for PDPs and ALEs.

```
import numpy as np
import pandas as pd


import matplotlib.pyplot as plt

# Simulate data
np.random.seed(0)
n_samples = 1000
temperature = np.random.uniform(0, 40, n_samples)  # Temperature in Celsius
time_of_day = np.random.uniform(0, 24, n_samples)  # Time of day in hours
```

(continues on next page)

(continued from previous page)

```
day_of_week = np.random.randint(0, 7, n_samples)  # Day of the week (0=Sunday,␣
↪6=Saturday)

price = (0.1 * (temperature - 20)**2 +
         5 * np.sin((time_of_day - 7) * np.pi / 6) * (time_of_day >= 7) * (time_of_
↪day <= 10) +
         5 * np.sin((time_of_day - 18) * np.pi / 4) * (time_of_day >= 18) * (time_of_
↪day <= 22) +
         -5 * np.sin((time_of_day - 10) * np.pi / 7) * (time_of_day >= 10) * (time_of_
↪day <= 17) +
         7 * np.sin((time_of_day) * np.pi / 24) +
         5 * (day_of_week == 0) +
         np.random.normal(0, 2, n_samples) * 2)

data = pd.DataFrame({
    'Temperature': temperature,
    'Hour': time_of_day,
    'Day': day_of_week,
    'Price': price
})

data.head()
```

	Temperature	Hour	Day	Price
0	21.952540	14.229126	3	-2.562450
1	28.607575	0.241529	3	7.237870
2	24.110535	11.419829	3	5.621266
3	21.795327	17.010489	5	6.220472
4	16.946192	1.055410	4	-2.435281

We can then train a random forest model on the data we just generated and then apply SHAP to explain its predictions.

```
from sklearn.ensemble import RandomForestRegressor

# Split data into features and target
X = data[['Temperature', 'Hour', 'Day']]
y = data['Price']

# Train a Random Forest Regressor
rf = RandomForestRegressor(n_estimators=100, random_state=0)
rf.fit(X, y)
```

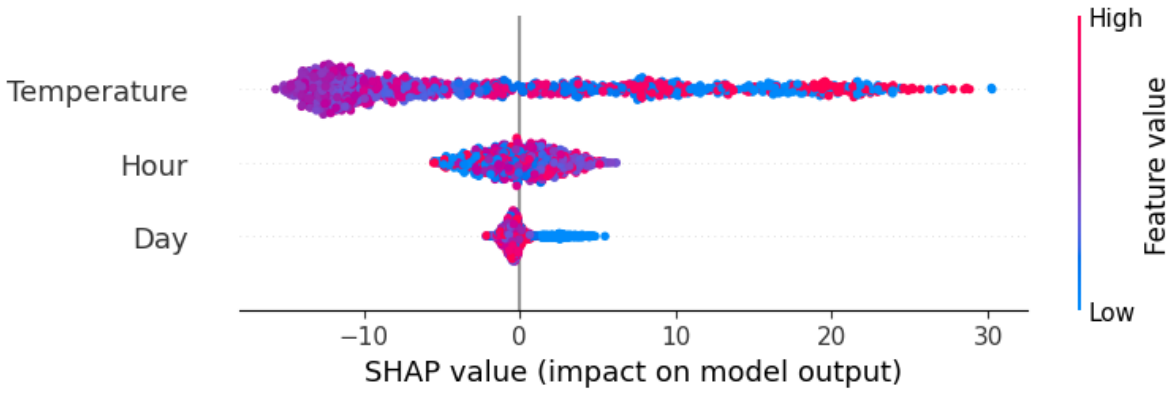
```
RandomForestRegressor(random_state=0)
```

We can implement SHAP for the random forest using the `TreeExplainer` function from the `shap` library:

```
import shap

# Use SHAP to explain the model predictions
explainer = shap.TreeExplainer(rf)
shap_values = explainer.shap_values(X)
```

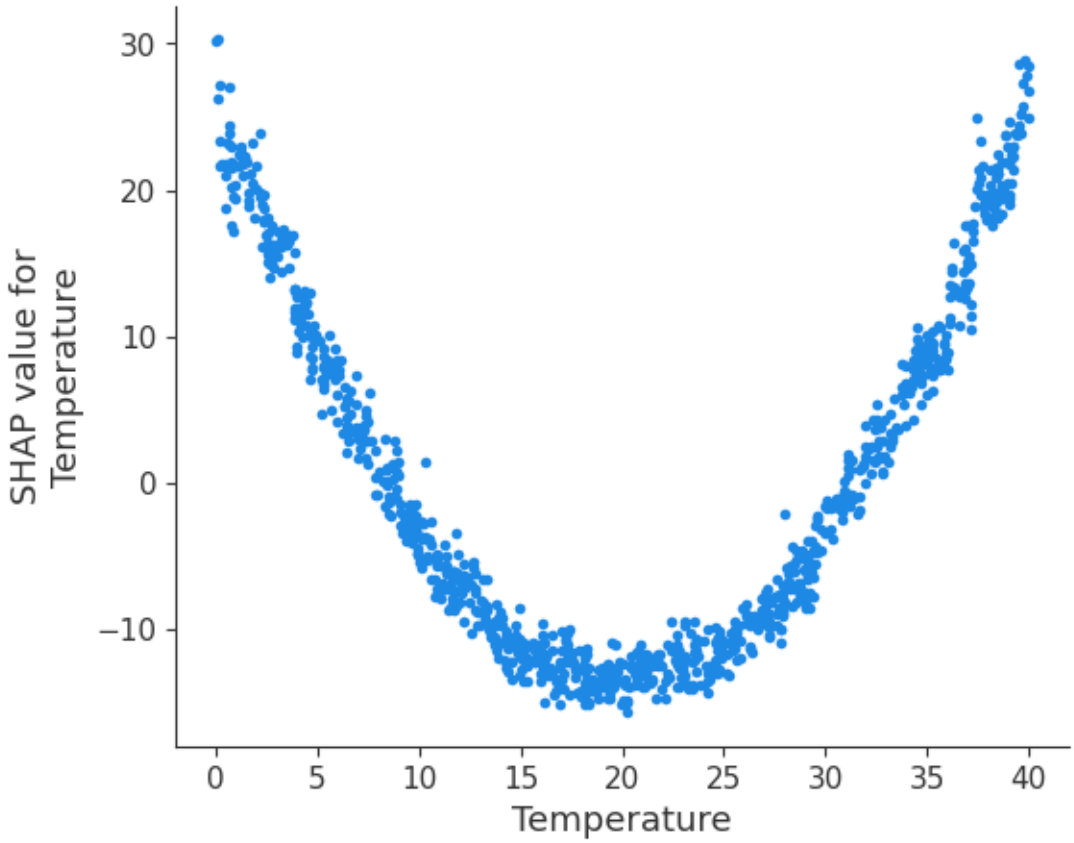
Then, we can visualize the results using the **summary plot**, which shows the distribution of Shapley values for all features across the dataset.

```
# Summary plot for all instances
shap.summary_plot(shap_values, X, plot_type="dot")
```



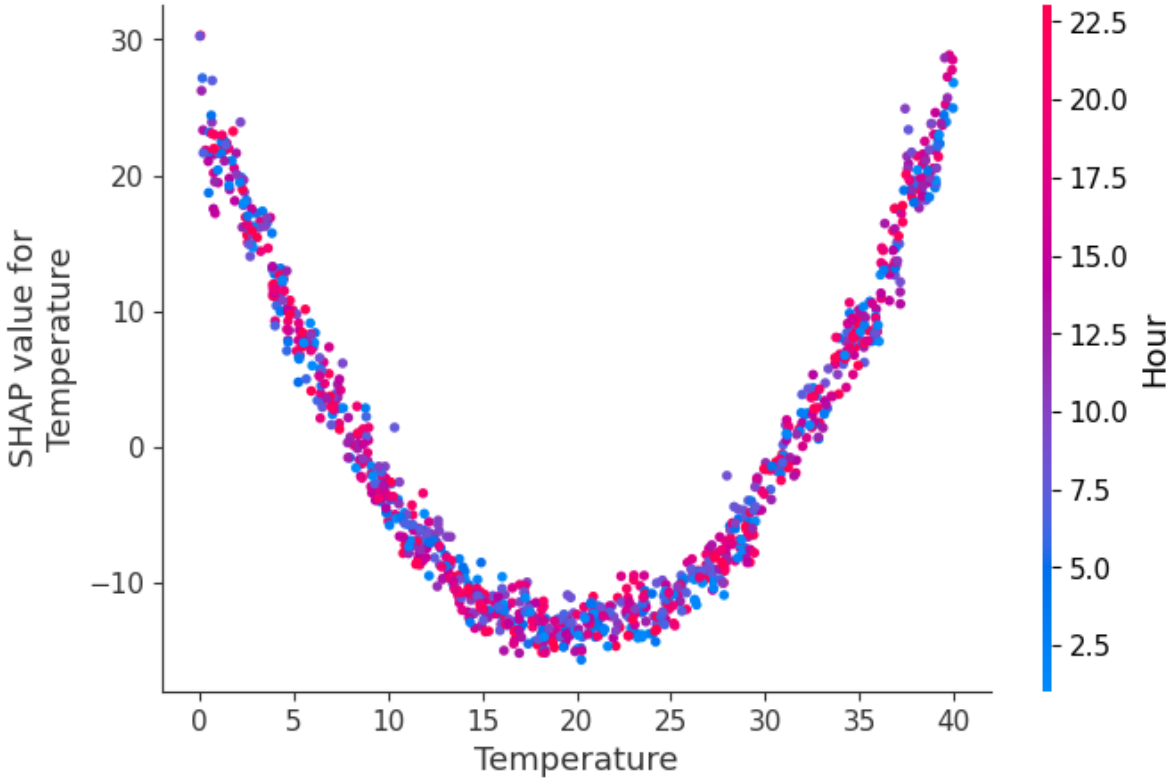
The **dependence plot** highlights how a specific feature (e.g., Temperature) influences the prediction.

```
# Dependence plot for a specific feature without interaction feature
shap.dependence_plot('Temperature', shap_values, X, interaction_index=None)
```



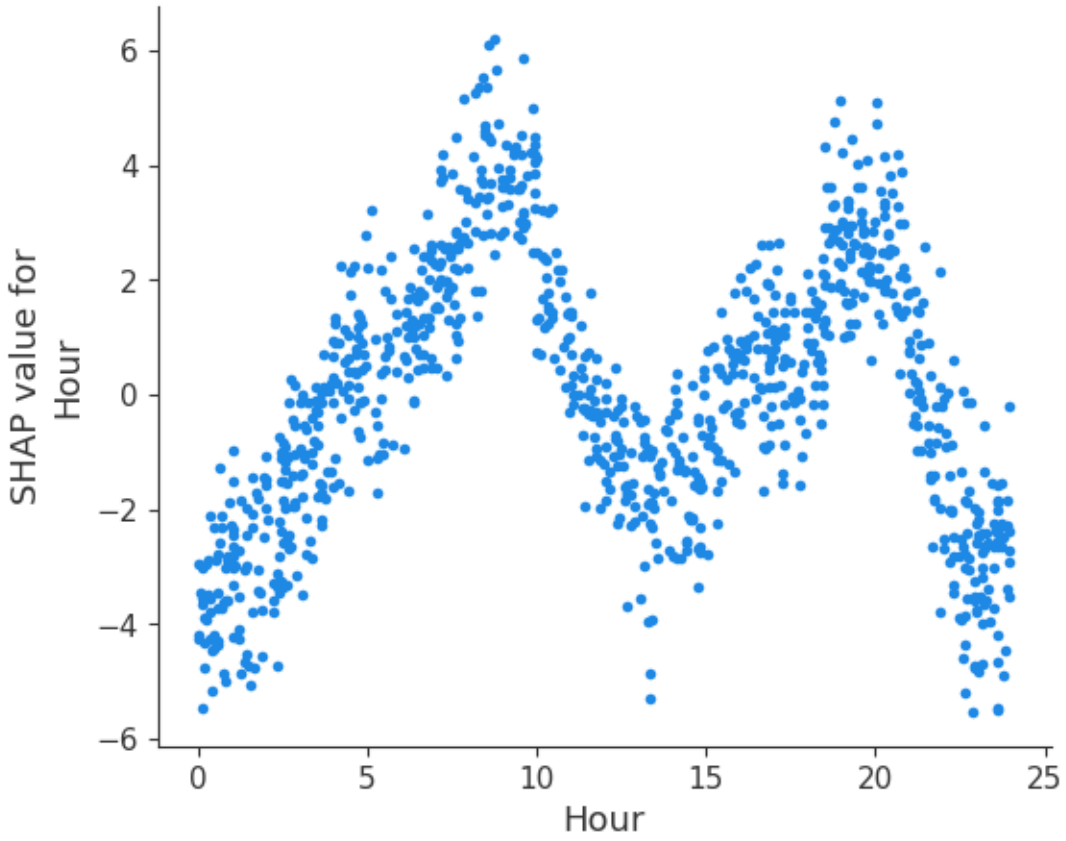
We can also see the same plot including the interaction between Temperature and Hour:

```
# Dependence plot for a specific feature with interaction
shap.dependence_plot('Temperature', shap_values, X, interaction_index='Hour')
```

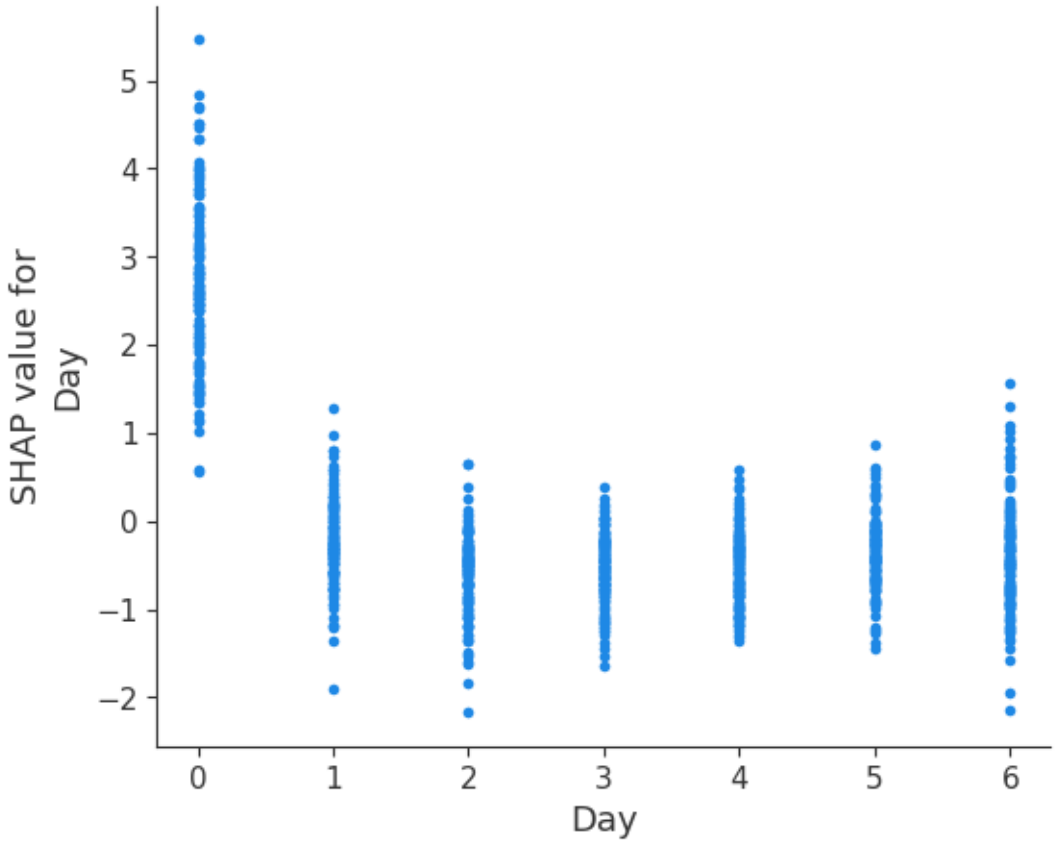


Then, we can also plot the dependence plots for Hour and Day, to see if SHAP is able to unveil the daily and weekly profiles.

```
# Dependence plot for a specific feature
shap.dependence_plot('Hour', shap_values, X, interaction_index=None)
```




```
# Dependence plot for a specific feature
shap.dependence_plot('Day', shap_values, X, interaction_index=None)
```



Shapley values offer a robust method for interpreting machine learning models by providing a fair and comprehensive attribution of feature contributions to individual predictions. In the context of electricity markets, this can help stakeholders understand the impact of various factors on electricity prices and make more informed decisions.

Part VII

V. Experiments and Data Collection

CHAPTER

TWENTYSIX

OVERVIEW

Collecting relevant data is always the first step for a good statistical analysis. In this context, a designed experiment refers to the structured and methodical approach to planning, conducting, analysing, and interpreting controlled experiments. This part covers techniques for setting up experiments to ensure that the results are valid, reliable, and can be used to draw causal conclusions. Effective experimental design is key to deriving actionable insights and making evidence-based decisions for several reasons:

- **Validity**: ensures that the experiments measure what they are intended to measure, providing accurate results.
- **Reliability**: guarantees that the results are consistent and replicable across different trials and settings.
- **Causality**: facilitates the identification of causal relationships by controlling for confounding variables and ensuring that the observed effects are due to the treatments applied.
- **Efficiency**: optimizes the use of resources, minimizing the cost and time required to conduct experiments while maximizing the information gained.

Poor experimental design can lead to several issues:

- **Bias**: without proper control and randomization, experiments can produce biased results, leading to incorrect conclusions.
- **Confounding**: failing to account for confounding variables can obscure the true relationship between the treatment and the outcome.
- **Lack of generalizability**: experiments that are not well-designed may produce results that cannot be generalized to other settings, populations, or time periods.

26.1 Content of Data Collection chapters

Chapter	Description
Design of Experiments	Techniques for planning and conducting controlled experiments to draw causal conclusions.
Active Learning	Methods for sequentially selecting data points to be labeled in a way that improves model performance.
A/B Testing	Practical applications of A/B testing to compare two versions of a variable to determine which performs better.
Multi-Armed Bandits	Approaches to balance exploration and exploitation in experimental settings to optimize decision-making.

CHAPTER

TWENTYSEVEN

A/B TESTING

A/B testing is one of the most commonly employed experimental framework. It is a powerful method for comparing two versions of a variable to determine which one performs better in terms of a given metric. This technique is widely used in various fields, including marketing, product development, and website optimization, to make data-driven decisions. In this tutorial, we'll explore the basics of A/B testing, illustrate a simple example, and delve into more advanced topics, such as network effects and complex statistical considerations.

27.1 A simple example

Consider a scenario where we want to compare the effectiveness of two different marketing campaigns aimed at encouraging customers to use more energy during off-peak hours. We'll define two groups:

- **Group A (control group)**: receives the standard marketing message.
- **Group B (treatment group)**: receives a new, enhanced marketing message.

The goal is to determine which marketing message results in higher usage of energy during off-peak hours.

Step-by-step procedure:

1. **Random assignment**: randomly assign customers to either Group A or Group B to ensure that each group is representative of the overall population.
2. **Implementation**: send the standard marketing message to Group A, and the enhanced marketing message to Group B.
3. **Measurement**: after a predetermined period, measure the amount of energy used by customers in both groups during off-peak hours.
4. **Analysis**: compare the average energy usage between the two groups using statistical tests, such as a **t-test**, to determine if the difference is statistically significant.

```
import numpy as np
import pandas as pd
from scipy import stats

# Simulate data
np.random.seed(42)
group_a = np.random.normal(loc=50, scale=10, size=100)  # Control group
group_b = np.random.normal(loc=55, scale=10, size=100)  # Treatment group

# Perform t-test
t_stat, p_value = stats.ttest_ind(group_b, group_a)
```

(continues on next page)

(continued from previous page)

```
print(f"T-statistic: {t_stat:.3f}")
print(f"P-value: {p_value:.3f}")

# Interpretation
if p_value < 0.05:
    print("The enhanced marketing message significantly increases energy usage during␣
 ↪off-peak hours.")
else:
    print("There is no significant difference between the two marketing messages.")
```

```
T-statistic: 4.755
P-value: 0.000
The enhanced marketing message significantly increases energy usage during off-
 ↪peak hours.
```

27.2 Statistical considerations

A/B testing involves several statistical concepts that needs to be taken into account to ensure valid and reliable results. These includes:

1. **Power analysis**: to determine the sample size needed to detect a significant effect.
2. **Multiple testing**: to adjust for the increased likelihood of Type I errors when conducting multiple comparisons (e.g., using Bonferroni correction).
3. **Heterogeneous treatment effects**: to explore how treatment effects vary across different subgroups.
4. **Network effects**: to eestimate treatment effects in networks where there is interference between users or consumers.

27.2.1 Power analysis

Here we demonstrate how to determine the minimum number of participants needed in each group of an A/B test to ensure reliable results. This process, called power analysis, helps us calculate the sample size required to detect a meaningful difference between two groups with a high probability. Power analysis is a technique used to figure out the smallest sample size that can reliably detect an effect in an experiment. It balances the need to find real differences with the desire to avoid wasting resources on excessively large samples. Essentially, it helps ensure that our A/B test is both effective and efficient.

Key parameters:

- **Effect size**: this measures the expected difference between the groups. It is a standardized value that reflects how large an effect we expect the treatment to have. A common way to select the effect size is to use previous research or pilot studies to estimate the likely impact. In our example, we chose an effect size of 0.5, indicating a moderate difference that we want to detect. The effect size is a measure of the magnitude of the difference between two groups. It is not just the raw difference between the means of the two groups, but rather a standardised measure that accounts for the variability within the data. This allows for a more meaningful comparison across different contexts and studies.
- **Significance level (alpha)**: this is the threshold for determining statistical significance, often set at 0.05. It represents the probability of rejecting the null hypothesis (concluding there is an effect) when it is actually true (a false positive). We selected an alpha of 0.05, which is standard in many fields, meaning we are willing to accept a 5% chance of a false positive.
- **Power**: this represents the probability of correctly rejecting the null hypothesis when it is false, i.e., detecting a true effect. A common target for power is 0.8, or 80%, meaning we want an 80% chance of detecting the effect if it

truly exists. This balance ensures that we have a high likelihood of finding true effects without requiring excessively large sample sizes.

```
from statsmodels.stats.power import TTestIndPower

# Parameters
effect_size = 0.5
alpha = 0.05
power = 0.8

# Calculate required sample size
analysis = TTestIndPower()
sample_size = analysis.solve_power(effect_size=effect_size, alpha=alpha, power=power,␣
↪alternative='two-sided')
print(f"Required sample size per group: {int(sample_size)}")
```

```
Required sample size per group: 63
```

After running the power analysis with these parameters, the output tells us the required sample size per group. For instance, if the result is 63, it means we need at least 63 participants in both the treatment and control groups. This ensures that our A/B test has enough power to detect a moderate effect with an 80% chance, while keeping the probability of a false positive at 5%. This helps in making sure our experiment is well-designed and our conclusions are reliable.

27.2.2 Multiple testing

When conducting multiple tests, the probability of making at least one type I error (false positive) increases. To address this, we use the Bonferroni correction, which adjusts the significance level to control the family-wise error rate (FWER). The adjusted significance level is given by:

$$\alpha_{\text{adj}} = \frac{\alpha}{m} \tag{27.1}$$

where α is the original significance level, and m is the number of tests.

Let's consider an example where a utility company is testing five different marketing campaigns to see if they significantly increase the adoption of a new energy-saving device.

Scenario:

- The company runs five separate A/B tests, one for each marketing campaign.
- The null hypothesis for each test is that the campaign has no effect on adoption rates.
- The significance level for the tests is initially set at 0.05.

Step-by-step procedure:

1. **Set parameters**: original significance level ((\alpha)) and number of tests.
2. **Simulate p-values**: generate p-values from the five tests. In a realistic scenario, these p-values would come from actual data analysis.
3. **Apply Bonferroni correction**: adjust the significance level using the Bonferroni method, and determine which hypotheses to reject based on the corrected p-values.

```
from statsmodels.stats.multitest import multipletests

# Parameters
alpha = 0.05  # Original significance level
```

(continues on next page)

(continued from previous page)

```
num_tests = 5  # Number of tests

# Adjusted significance level using Bonferroni correction
alpha_adj = alpha / num_tests
print(f"Adjusted significance level: {alpha_adj}")

# Simulate p-values from 5 tests
# Realistically, these p-values would be the result of statistical tests on actual␣
↪data
np.random.seed(42)
p_values = np.array([0.02, 0.04, 0.20, 0.01, 0.03])  # Simulated p-values for␣
↪illustration
print(f"Original p-values: {p_values}")

# Apply Bonferroni correction
reject, pvals_corrected, _, _ = multipletests(p_values, alpha=alpha, method=
↪'bonferroni')
print(f"Corrected p-values: {pvals_corrected}")
print(f"Reject null hypothesis: {reject}")

# Interpretation of results
for i, (pval, corr_pval, rej) in enumerate(zip(p_values, pvals_corrected, reject)):
    print(f"Test {i + 1}: Original p-value = {pval}, Corrected p-value = {corr_pval},␣
↪Reject null hypothesis = {rej}")
```

```
Adjusted significance level: 0.01
Original p-values: [0.02 0.04 0.2  0.01 0.03]
Corrected p-values: [0.1  0.2  1.   0.05 0.15]
Reject null hypothesis: [False False False  True False]
Test 1: Original p-value = 0.02, Corrected p-value = 0.1, Reject null hypothesis =␣
↪False
Test 2: Original p-value = 0.04, Corrected p-value = 0.2, Reject null hypothesis =␣
↪False
Test 3: Original p-value = 0.2, Corrected p-value = 1.0, Reject null hypothesis =␣
↪False
Test 4: Original p-value = 0.01, Corrected p-value = 0.05, Reject null hypothesis␣
↪= True
Test 5: Original p-value = 0.03, Corrected p-value = 0.15, Reject null hypothesis␣
↪= False
```

This correction ensures that we control the family-wise error rate, reducing the risk of false positives when conducting multiple tests. Here we used fictional p-values: `[0.02, 0.04, 0.20, 0.01, 0.03]`. In practice, these would be obtained from statistical tests on actual experimental data.

27.2.3 Heterogeneous treatment effects

Heterogeneous treatment effects occur when the effect of a treatment varies across different subgroups of the population. Identifying these differences is crucial for understanding how different groups respond to the treatment and for making targeted decisions.

Suppose an electricity company is testing a new marketing campaign to encourage customers to switch to a green energy plan. The company wants to understand if the campaign's effectiveness varies by customer age. Let's simulate this case with a simple example, where:

- We generate data for 1000 customers with ages between 18 and 70.

- The `treatment` variable indicates whether the customer received the marketing campaign (1) or not (0).
- The `response` variable represents whether the customer switched to the green energy plan, influenced by their age, the treatment, and the interaction between age and treatment.

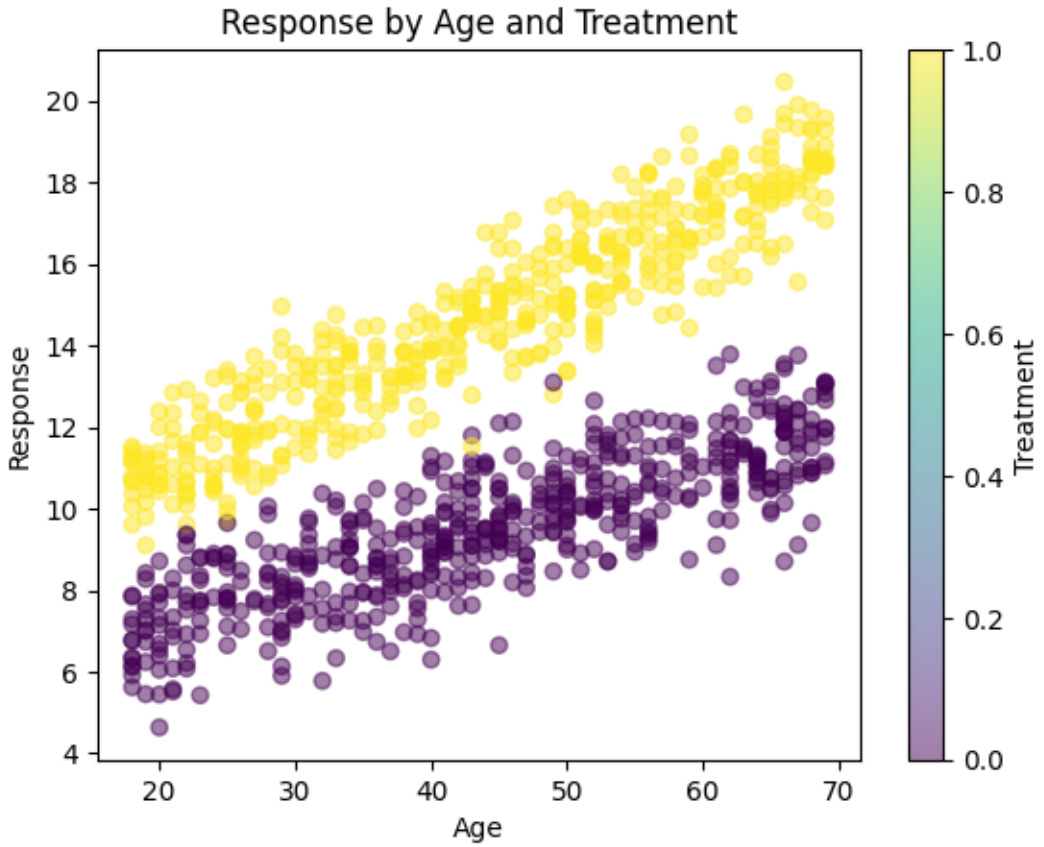
```
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Simulate data
np.random.seed(42)
n = 1000
age = np.random.randint(18, 70, size=n)
treatment = np.random.choice([0, 1], size=n)
age_treatment = age * treatment
response = 5 + 3 * treatment + 0.1 * age + 0.05 * age_treatment + np.random.
 ↪normal(size=n)

# Create DataFrame
data = pd.DataFrame({'age': age, 'treatment': treatment, 'response': response, 'age_
 ↪treatment': age_treatment})

# Fit the model
X = sm.add_constant(data[['treatment', 'age', 'age_treatment']])
model = sm.OLS(data['response'], X).fit()
print(model.summary())

# Plot interaction
plt.scatter(data['age'], data['response'], c=data['treatment'], cmap='viridis',␣
 ↪alpha=0.5)
plt.xlabel('Age')
plt.ylabel('Response')
plt.title('Response by Age and Treatment')
plt.colorbar(label='Treatment')
plt.show()
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               response   R-squared:                       0.912
Model:                            OLS   Adj. R-squared:                  0.912
Method:                 Least Squares   F-statistic:                     3446.
Date:                Sun, 07 Jul 2024   Prob (F-statistic):               0.00
Time:                        20:02:47   Log-Likelihood:                -1409.1
No. Observations:                1000   AIC:                             2826.
Df Residuals:                     996   BIC:                             2846.
Df Model:                           3
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
const            5.2533      0.135     38.770      0.000       4.987       5.519
treatment        2.5726      0.194     13.260      0.000       2.192       2.953
age              0.0967      0.003     33.110      0.000       0.091       0.102
age_treatment    0.0584      0.004     13.948      0.000       0.050       0.067
==============================================================================
Omnibus:                        1.105   Durbin-Watson:                   2.009
Prob(Omnibus):                  0.576   Jarque-Bera (JB):                1.138
Skew:                          -0.080   Prob(JB):                        0.566
```

(continues on next page)

(continued from previous page)

```
Kurtosis:                       2.959   Cond. No.                         369.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```



In the results:

- The coefficient for `treatment` indicates the average effect of the marketing campaign.
- The coefficient for `age` shows the effect of age on the response.
- The coefficient for `age_treatment` reveals how the treatment effect changes with age. A significant coefficient suggests that the treatment effect varies across different age groups.

This example demonstrates how to estimate heterogeneous treatment effects in an A/B testing context. By including interaction terms in the regression model, we can identify how the treatment effect varies across different subgroups, providing valuable insights for tailoring strategies to specific segments. This approach helps maximize the effectiveness of interventions by understanding and leveraging the heterogeneity in treatment effects.

27.2.4 Network effects

Traditional A/B testing methods, relying on the stable unit treatment value assumption (SUTVA), assume that each individual's response to a treatment is independent of others. However, in many real-world scenarios, especially in social networks and interconnected systems, this assumption does not hold. Users' behaviors and outcomes can be influenced by their peers, leading to network effects. Network A/B testing is an advanced technique that takes into account the influence of social connections and interactions when evaluating the effectiveness of treatments or interventions [GXBH15].

Relevance of network A/B testing:

1. **Capturing spillover effects**: in social networks, the effect of a treatment on one individual can spill over to their connected peers. For instance, in marketing campaigns, a customer's decision to adopt a new product might influence their friends to do the same.
2. **Accurate estimation of treatment effects**: by considering network effects, we can obtain more accurate estimates of the Average Treatment Effect (ATE). Ignoring these effects can lead to biased results and incorrect conclusions.
3. **Optimizing interventions**: understanding how treatments propagate through networks can help in designing more effective interventions. For example, targeting influential nodes in a network can amplify the overall impact of a campaign.

In network A/B testing, we need to account for two main factors:

- **Direct treatment effect**: The impact of the treatment on the individual who receives it.
- **Indirect treatment effect**: The impact of the treatment on individuals connected to those who receive it.

To estimate these effects, we use a model that includes both the treatment status of the individual and the treatment status of their neighbors.

Consider a scenario where an electricity company wants to test a new marketing campaign to encourage customers to switch to a green energy plan. The company knows that customers are likely to influence each other's decisions through social interactions. Therefore, they want to account for these network effects in their A/B testing. We will simulate a network of customers, randomly assign them to treatment or control groups, and then estimate the treatment effects considering the influence of their connected peers.

Step-by-step procedure:

Generating the network: we use an Erdős-Rényi model to create a random network. In this model, each pair of nodes (customers) has a fixed probability of being connected.

Simulating treatment assignment: we randomly assign each node (customer) to either the treatment group (Z=1) or the control group (Z=0) with equal probability. This represents whether a customer receives the new marketing campaign or not.

Simulating the response variable: The response variable Y represents the outcome we are interested in measuring, such as whether a customer switched to the green energy plan. The outcome for each customer is influenced by their own treatment status Z and the treatment statuses of their connected neighbors, which we account for using the adjacency matrix A.

The formula used for simulating the response is:

$$Y = \beta_0 + \beta_1 Z + \beta_2 (A \cdot Z) + \epsilon \tag{27.2}$$

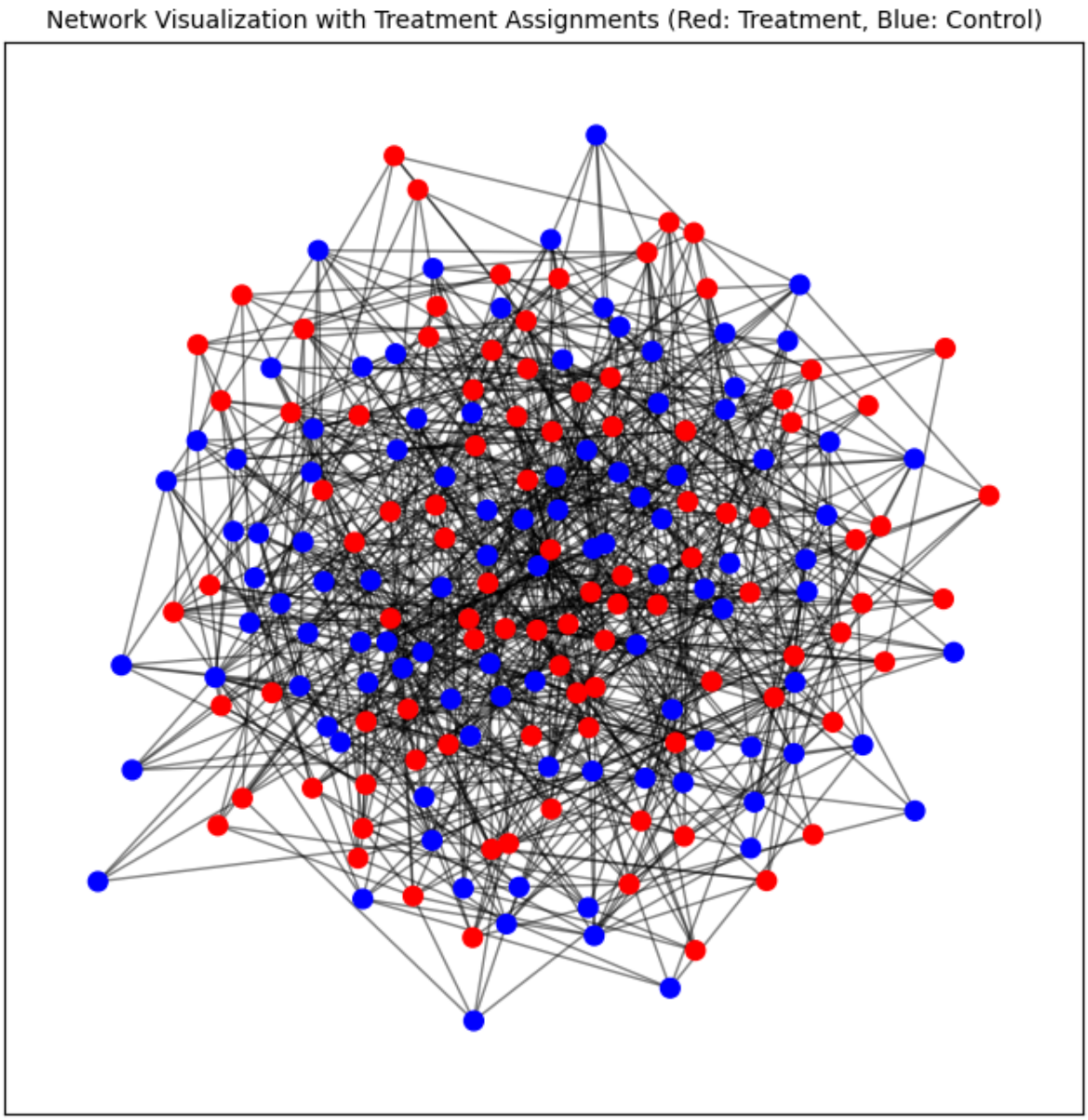
where:

- β_0 is the intercept.
- β_1 is the direct effect of the treatment.
- β_2 captures the network effects (i.e., how the treatment status of neighbors influences the outcome).
- ϵ is random noise.

```
import networkx as nx

# Generate a random graph
n_samples = 200
G = nx.erdos_renyi_graph(n=n_samples, p=0.05, seed=42)
A = nx.to_numpy_array(G)

# Simulate data
np.random.seed(42)
Z = np.random.choice([0, 1], size=n_samples, p=[0.5, 0.5])  # Random assignment to␣
↪treatment or control
Y = 2 + 1.5 * Z + 0.8 * np.dot(A, Z) + np.random.normal(size=n_samples)  # Network␣
↪effects
```

The response Y is simulated by combining the direct effect of the treatment (Z) and the indirect network effect ($A \cdot Z$). The term $\beta_2(A \cdot Z)$ captures the influence of the treatment status of neighboring nodes on the outcome.

```
# Plotting the network
plt.figure(figsize=(8, 8), dpi=100)
pos = nx.spring_layout(G, seed=42)
node_color = ['r' if z == 1 else 'b' for z in Z]
nx.draw_networkx_nodes(G, pos, node_color=node_color, node_size=60, alpha=1)
nx.draw_networkx_edges(G, pos, alpha=0.5)
plt.title("Network Visualization with Treatment Assignments (Red: Treatment, Blue:␣
↪Control)", fontsize=10)
plt.show()
```



We now prepare the aata for fitting a linear regression model. In particular, we create the feature matrix X, which includes:

- The treatment status Z of each customer.
- The sum of the treatment statuses of their neighbors, calculated as $A \cdot Z$.

Then, we can fit a linear regression model to estimate the coefficients β_1 and β_2, which correspond to the direct treatment effect and the network effect, respectively.

```
# Prepare the data for regression
X = np.column_stack([Z, np.dot(A, Z)])
X = sm.add_constant(X)  # Add intercept

# Fit the model using statsmodels
model = sm.OLS(Y, X).fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.746
Model:                            OLS   Adj. R-squared:                  0.744
Method:                 Least Squares   F-statistic:                     290.0
Date:                Sun, 07 Jul 2024   Prob (F-statistic):           1.98e-59
Time:                        20:02:48   Log-Likelihood:                -276.44
No. Observations:                 200   AIC:                             558.9
Df Residuals:                     197   BIC:                             568.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.1994      0.191     11.502      0.000       1.822       2.577
x1             1.5484      0.138     11.249      0.000       1.277       1.820
x2             0.7663      0.035     21.954      0.000       0.697       0.835
==============================================================================
Omnibus:                        9.767   Durbin-Watson:                   2.140
Prob(Omnibus):                  0.008   Jarque-Bera (JB):               13.614
Skew:                           0.315   Prob(JB):                      0.00111
Kurtosis:                       4.112   Cond. No.                         15.3
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

The model summary provides the estimates for β_1 (direct treatment effect) and β_2 (network effect). The estimated Average Treatment Effect (ATE) considering network effects is extracted from the coefficient β_1.

```
# Calculate ATE considering network effects
ATE = model.params[1]  # Coefficient for treatment
print(f"ATE considering network effects: {ATE:.3f}")
```

```
ATE considering network effects: 1.548
```

By simulating the response variable to include both direct and network effects, and then using regression analysis to estimate these effects, we can accurately measure the influence of treatments in interconnected systems. This approach helps account for the spillover effects that are common in social networks and other interconnected environments.

CHAPTER

TWENTYEIGHT

MULTI-ARMED BANDITS

Multi-armed bandit (MAB) problems are a class of sequential decision-making problems that model the trade-off between exploration and exploitation. The name comes from the metaphor of a gambler facing multiple slot machines (one-armed bandits), each with a different probability of payout. The gambler's objective is to maximize their total reward over a series of pulls.

The MAB problem can be formalized as follows:

- **Arms**: let K be the number of arms (choices or actions) available.
- **Rewards**: each arm $k \in \{1, 2, ..., K\}$ provides a reward $r_k(t)$ at time t.
- **Objective**: the objective is to maximize the cumulative reward over T rounds, $\sum_{t=1}^{T} r_k(t)$.

Imagine you are a decision-maker in an electricity market, tasked with optimizing various strategies such as dynamic pricing, demand response programs, or marketing campaigns. Each strategy can be thought of as a slot machine (arm), each with an unknown probability of success (reward). You have a limited budget and need to decide how to allocate resources across these strategies to maximize your overall reward. The dilemma is that the more you explore different strategies to learn their effectiveness, the less you have left to exploit the best-performing strategy. This trade-off between exploration (trying different options to gather information) and exploitation (using known information to maximize reward) lies at the heart of the multi-armed bandit (MAB) problem.

Practical applications of MAB problems in electricity markets might include:

1. **Dynamic pricing**: suppose you need to determine the optimal pricing strategy for electricity during peak and off-peak hours. You could implement different pricing models (arms) and observe consumer reactions (rewards). Using MAB, you can dynamically adjust the pricing strategies based on observed data to maximize revenue without running prolonged inefficient experiments.
2. **Demand response programmes**: consider various incentive schemes to encourage consumers to reduce usage during peak times. Each scheme can be tested (explored) initially, and based on which ones yield the highest reductions in usage (exploitation), more resources can be allocated to the most effective programs.
3. **Marketing campaigns**: you may have multiple marketing strategies to promote energy-efficient appliances. Initially, you allocate equal resources to all strategies to see which performs best. As data comes in, you shift more resources to the campaigns that show higher engagement and conversion rates, optimizing your overall marketing budget.

28.1 MAB vs. A/B testing

Traditional methods for testing different options, such as A/B testing, involve splitting resources equally across different strategies (pure exploration), but this can be inefficient and costly, as it doesn't adapt to the observed performance of the strategies. To this extent, the key **limitations of A/B testing** are:

- **Resource inefficiency**: equally distributing resources among all strategies can waste time and opportunities on less-performing options.
- **Costly**: every test interaction involves costs related to market operations, consumer interactions, and potential financial impacts.
- **Non-personalized**: A/B testing typically identifies a winner for the majority, which may not be optimal for all segments of the market.

On the other side, the **advantages of MAB approaches** include:

- **Dynamic allocation**: MAB algorithms initially explore all options but gradually allocate more resources to the best-performing strategies, improving overall efficiency.
- **Higher success rates**: by continuously adapting to performance data, MAB approaches can increase the overall success rate of the strategies implemented.
- **Contextual personalization**: advanced MAB variants like contextual bandits tailor strategies to different market segments, enhancing personalization and engagement.

28.2 Key concepts and high-level example

The two key conepts in MAB problems are:

1. **Exploration vs. exploitation**: where **exploration** means trying out different arms to gather more information about their rewards, and **exploitation** refers to choosing the arm that is currently believed to provide the highest reward.
2. **Regret**: the difference between the reward obtained by the optimal arm and the reward obtained by the algorithm. The goal is to minimize regret over time.

Let's break down the MAB problem step by step, focusing on the concepts of exploration, reward, and how they are computed.

Step-by-step illustration

Setup

Imagine you have three slot machines (arms) in a casino, each with an unknown probability of payout (reward). Initially, you do not know which machine is the best, so you need to **explore** by trying out each machine. Over time, as you gather more information, you start to get an idea about the different machines and can **exploit** the available information by choosing the machine that seems to give the highest reward based on your observations.

True reward probabilities:

- Slot Machine 1: 0.3
- Slot Machine 2: 0.5
- Slot Machine 3: 0.7

These probabilities are unknown to you. Your goal is to find out which machine has the highest probability of giving a payout by trying them out.

Step 1: initial exploration

To start, you need to try each machine a few times to get an idea of their payouts.

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

# True reward probabilities
true_means = [0.3, 0.5, 0.7]

# Number of times to pull each machine initially
initial_pulls = 10

# Simulate initial exploration
np.random.seed(42)
initial_rewards = []

for i in range(len(true_means)):
    rewards = []
    for _ in range(initial_pulls):
        reward = 1 if np.random.rand() < true_means[i] else 0
        rewards.append(reward)
    initial_rewards.append(rewards)

# Print initial rewards
for i, rewards in enumerate(initial_rewards):
    print(f"Rewards for Machine {i + 1}: {rewards}")
```

```
Rewards for Machine 1: [0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
Rewards for Machine 2: [1, 0, 0, 1, 1, 1, 1, 0, 1, 1]
Rewards for Machine 3: [1, 1, 1, 1, 1, 0, 1, 1, 1, 1]
```

Step 2: compute average rewards

After the initial exploration, compute the average reward for each machine to get an estimate of their payout probabilities.

```
# Calculate average rewards
average_rewards = [np.mean(rewards) for rewards in initial_rewards]

# Print average rewards
for i, avg in enumerate(average_rewards):
    print(f"Average reward for Machine {i + 1}: {avg:.2f}")
```

```
Average reward for Machine 1: 0.30
Average reward for Machine 2: 0.70
Average reward for Machine 3: 0.90
```

The mean of the rewards obtained from the initial pulls for each machine. This gives an estimate of the payout probability for each machine.

Step 3: exploitation

Based on the average rewards, you start choosing the machine that seems to give the highest reward more often.

```
# Number of additional rounds
additional_rounds = 70

# Continue simulation
```

(continues on next page)

(continued from previous page)

```
total_rewards = initial_rewards.copy()
cumulative_rewards = np.zeros(initial_pulls * 3 + additional_rounds)
cumulative_rewards[:initial_pulls * 3] = np.array([sum(rewards) for rewards in␣
↪initial_rewards]).repeat(initial_pulls)

for t in range(additional_rounds):
    # Choose the machine with the highest average reward
    best_machine = np.argmax(average_rewards)

    # Simulate pulling the best machine
    reward = 1 if np.random.rand() < true_means[best_machine] else 0
    total_rewards[best_machine].append(reward)

    # Update average rewards
    average_rewards[best_machine] = np.mean(total_rewards[best_machine])

    # Update cumulative rewards
    cumulative_rewards[initial_pulls * 3 + t] = cumulative_rewards[initial_pulls * 3␣
↪+ t - 1] + reward

# Print final average rewards
for i, avg in enumerate(average_rewards):
    print(f"Final average reward for Machine {i + 1}: {avg:.2f}")
```

```
Final average reward for Machine 1: 0.30
Final average reward for Machine 2: 0.64
Final average reward for Machine 3: 0.71
```

At each step, we define as **best machine** the machine with the highest average reward so far, and we continue pulling the best machine and update the average rewards based on the new observations.

Step $: regret calculation

Regret is a measure of how much worse our algorithm performs compared to if we had always chosen the best possible machine (the one with the highest true mean reward). If we knew in advance which machine had the highest probability of giving us a prize, we would always choose that machine. However, since we do not know this in advance, we have to try out all the machines to gather information. While we are trying out all the machines (exploring), we might choose the less optimal machines sometimes, which gives us less reward compared to the best machine. Regret is the difference between the total reward we would have gotten by always choosing the best machine and the total reward we actually got by following our algorithm.

We have:

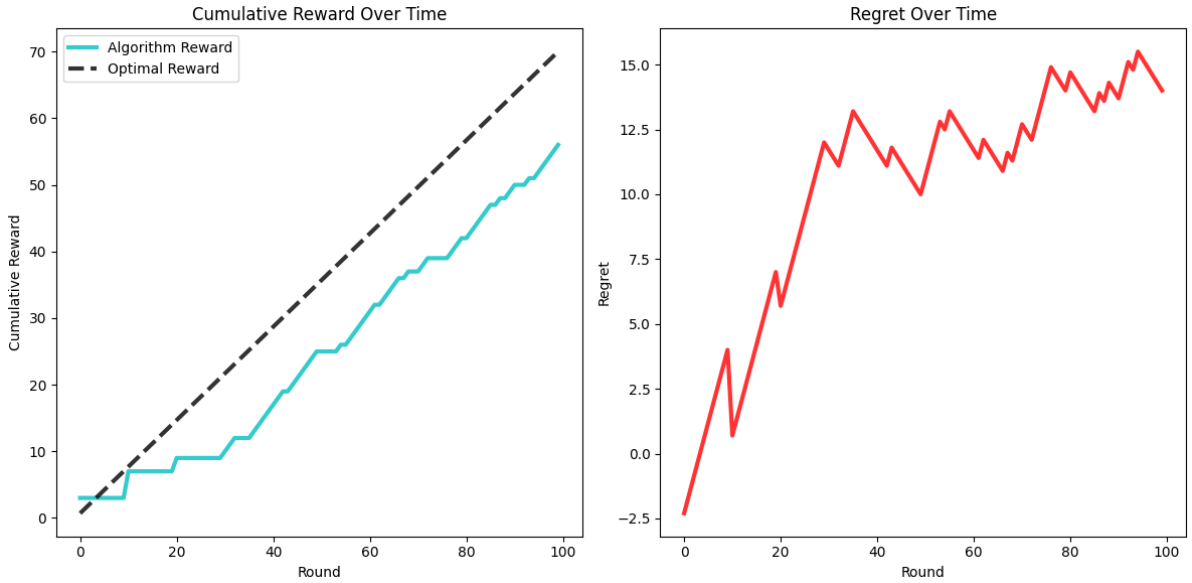
- Optimal reward: this is the reward we would have gotten if we had always chosen the best machine. It is calculated as the cumulative sum of the highest true mean reward over all rounds.
- Algorithm reward: this is the reward we actually got by following our algorithm, which includes exploration and exploitation steps.
- Regret: the difference between the optimal reward and the algorithm reward,

```
# True means and optimal reward
optimal_mean = max(true_means)
optimal_cumulative_rewards = np.cumsum([optimal_mean] * (initial_pulls * 3 +␣
↪additional_rounds))
```

(continues on next page)

(continued from previous page)

```
# Calculate regret
regret = optimal_cumulative_rewards - cumulative_rewards

# Plot cumulative rewards and regret
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(cumulative_rewards, label='Algorithm Reward', lw=3, c='c', alpha=.8)
plt.plot(optimal_cumulative_rewards, label='Optimal Reward', linestyle='--', lw=3, c=
↪'k', alpha=.8)
plt.xlabel('Round')
plt.ylabel('Cumulative Reward')
plt.title('Cumulative Reward Over Time')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(regret, label='Regret', color='r', lw=3, alpha=.8)
plt.xlabel('Round')
plt.ylabel('Regret')
plt.title('Regret Over Time')

plt.tight_layout()
plt.show()
```



How to read these graphs:

1. **Cumulative reward**:
    - This graph shows the total reward accumulated over time by the algorithm.
    - The x-axis represents the number of rounds (time), and the y-axis represents the cumulative reward.
    - Initially, during the exploration phase, the rewards might increase slowly because the algorithm is trying out different arms to gather information.
    - As the algorithm gathers more information and starts to exploit the best-performing arm, the slope of the cumulative reward graph should increase, indicating a faster accumulation of rewards.

- The optimal cumulative reward line represents the reward we would have accumulated if we had always chosen the best arm from the beginning. This line has a constant, steep slope, indicating the highest possible reward accumulation rate.

2. **Regret**:
    - This graph shows the difference between the optimal cumulative reward and the cumulative reward obtained by the algorithm.
    - The x-axis represents the number of rounds (time), and the y-axis represents the regret.
    - At the beginning, the regret might increase rapidly because the algorithm is exploring and might be choosing suboptimal arms, leading to lower rewards compared to the optimal arm.
    - As the algorithm starts to exploit the best-performing arm more frequently, the rate at which regret increases should slow down. Ideally, the regret graph will start to flatten out, indicating that the algorithm is performing close to optimally.
    - The ideal scenario is to have a regret graph that flattens out as early as possible, showing that the algorithm quickly learned to choose the best arm.

28.3 Key algorithms for MAB problems

In this section, we will explain three popular algorithms for solving the multi-armed bandit (MAB) problem: epsilon-greedy, upper confidence bound (UCB), and Thompson sampling. We will also compare their performance against a random strategy over multiple simulation runs.

28.3.1 Epsilon-greedy

The Epsilon-Greedy algorithm balances exploration and exploitation by choosing a random arm with a small probability (ϵ) and the best-known arm with a large probability (1 - ϵ). Think of ϵ as your curiosity factor. A higher ϵ means you are more curious and willing to try different options even if you know some options perform well. A lower ϵ means you are more inclined to stick with what you know works best.

Mechanism:

- **Exploration**: with a small probability (ϵ), you choose a random arm to try out new options.
- **Exploitation**: with a large probability (1 - ϵ), you choose the arm that has given you the highest average reward so far.

Formulation:

- **Estimated mean reward**: for each arm k, you keep track of the average reward it has given you until this time t, $\hat{\mu}_k(t)$.
- **Updating the Estimate**: every time you pull an arm and get a reward, you update the average reward for that arm using the formula:

$$\hat{\mu}_k(t+1) = \hat{\mu}_k(t) + \frac{r_k(t) - \hat{\mu}_k(t)}{n_k(t)} \tag{28.1}$$

where $\hat{\mu}_k(t)$ is the current average reward for arm k at time t, $r_k(t)$ is the reward you get from arm k at time t, and $n_k(t)$ is the number of times you have pulled arm k up to time t.

Intuitively, when you receive a new reward for an arm, you need to update your estimate of the arm's average reward. The goal is to make sure that this estimate becomes more accurate as you gather more data. To update the estimate of the average reward, we need to balance:

- New information: the reward you just observed provides new information about the arm's performance.
- Past experience the average reward you have observed so far reflects your past experience with that arm.

The update formula balances these two by adjusting the current average reward ($\hat{\mu}_k(t)$) using the new reward ($r_k(t)$). The difference between the new reward and the current average reward is adjusted by $\frac{1}{n_k(t)}$. This fraction gets smaller as $n_k(t)$, the number of times the arm has been pulled, increases. The key reasons for this is that:

- Diminishing impact: the fraction $\frac{1}{n_k(t)}$ ensures that the impact of new rewards diminishes over time. This means that early rewards have a bigger influence on your estimate, helping you quickly form an initial understanding of the arm's performance.
- As you pull the arm more times, you gather more information, and your estimate becomes more stable. New rewards should have less impact on the estimate because you already have a lot of information. Later rewards refine your estimate more subtly, preventing large swings in the estimate and ensuring that the estimate converges to the true average reward as you gather more data.

Simple example to illustrate how it works

- **First Pull**:
  - Suppose you pull an arm for the first time and get a reward of 10.
  - Your initial estimate $\hat{\mu}_k(1)$ is 10 because it's the only data point you have.
- **Second Pull**:
  - You pull the same arm again and get a reward of 8.
  - You update your estimate:
$$\hat{\mu}_k(2) = 10 + \frac{8 - 10}{2} = 9 \tag{28.2}$$
  - The new reward adjusts the estimate significantly because you only have two data points.
- **Hundredth Pull**:
  - After pulling the arm 99 times, suppose your current estimate is 9.5.
  - You pull the arm one more time and get a reward of 9.
  - You update your estimate:
$$\hat{\mu}_k(100) = 9.5 + \frac{9 - 9.5}{100} = 9.495 \tag{28.3}$$
  - The new reward barely changes the estimate because you have a lot of data, making the estimate more stable.

By using this adjustment, you ensure that your estimate of the average reward for each arm becomes more accurate and stable over time, leading to better decision-making in the epsilon-greedy algorithm.

Implementation:

```
def epsilon_greedy(arms, epsilon=0.1, n_rounds=1000):
    n_arms = len(arms)
    rewards = np.zeros(n_arms)
    counts = np.zeros(n_arms)
    total_rewards = []

    for t in range(n_rounds):
        if np.random.rand() < epsilon:
            # Exploration: choose a random arm
            arm = np.random.choice(n_arms)
        else:
```

(continues on next page)

(continued from previous page)

```
            # Exploitation: choose the best arm so far
            arm = np.argmax(rewards / (counts + 1e-5))

        # Simulate pulling the arm
        reward = np.random.rand() < arms[arm]
        rewards[arm] += reward
        counts[arm] += 1
        total_rewards.append(reward)

    return np.cumsum(total_rewards)
```

28.3.2 Upper confidence bound (UCB)

This algorithm selects arms based on the upper confidence bounds (UCBs) of their estimated rewards. The UCB is a combination of the estimated reward and a term that accounts for the uncertainty in the estimate. The key idea is to decide which options to try out by balancing between choosing the option that seems the best based on what you know so far and exploring less-tried options to discover their potential. It does this by considering both the estimated reward of each option and the uncertainty around that estimate.

Mechanism:

- At each time step t, select the arm k that maximizes the upper confidence bound $UCB_k(t)$.

Formulation:

- The upper confidence bound for arm k at time t is given by:

$$UCB_k(t) = \hat{\mu}_k(t) + c\sqrt{\frac{\ln t}{n_k(t)}} \tag{28.4}$$

which is composed of two terms:

- **Estimated reward**: the term $\hat{\mu}_k(t)$ represents our current best guess of the mean reward for arm k based on the rewards we have observed so far. This is our current estimate of the average reward for arm k. It is updated every time we pull the arm and observe a reward, using the same updating mechanism as in the epsilon-greedy algorithm.
- **Uncertainty term**: The term $c\sqrt{\frac{\ln t}{n_k(t)}}$ represents the uncertainty or confidence interval around the estimated reward. It ensures that arms with fewer pulls (higher uncertainty) are given a higher chance of being selected. Here, $n_k(t)$ is the number of times arm k has been selected up to time t, and c is a confidence parameter that controls the degree of exploration. Let's break down the components of this term to better understand how it works:
  - $\ln t$: The natural logarithm of the current time step t. As time progresses, this term grows, but at a decreasing rate.
  - $\frac{1}{n_k(t)}$: the reciprocal of the number of times arm k has been pulled. This term decreases as we pull the arm more often.
  - Square root: the square root ensures that the uncertainty term grows more slowly as the number of pulls increases.
  - Confidence parameter c: this parameter controls how much weight we give to the uncertainty term. A higher c value means more exploration.

Why This Works:

- **Early stages**: at the beginning of the process, $n_k(t)$ is small for all arms, making the uncertainty term large. This encourages exploration of all arms to gather initial information.

- **Later stages**: as $n_k(t)$ increases for an arm, the uncertainty term decreases. This means that the algorithm will increasingly favor arms with higher estimated rewards, but will still occasionally explore other arms to ensure they are not overlooked. Over time, the algorithm balances between exploiting machines with high estimated rewards and exploring machines with higher uncertainty to ensure it does not miss out on potentially better options.

Implementation:

```
def ucb(arms, c=2, n_rounds=1000):
    n_arms = len(arms)
    rewards = np.zeros(n_arms)
    counts = np.zeros(n_arms)
    total_rewards = []

    for t in range(1, n_rounds + 1):
        ucb_values = rewards / (counts + 1e-5) + c * np.sqrt(np.log(t) / (counts + 1e-
↪5))
        arm = np.argmax(ucb_values)

        # Simulate pulling the arm
        reward = np.random.rand() < arms[arm]
        rewards[arm] += reward
        counts[arm] += 1
        total_rewards.append(reward)

    return np.cumsum(total_rewards)
```

28.3.3 Thompson sampling

Thompson sampling balances between trying out different options (exploration) and sticking with the best-known option (exploitation) by using probability distributions to model the uncertainty of each option's rewards.

Mechanism:

1. **Maintain a probability distribution**: for each option (or arm), we keep track of a probability distribution that represents our belief about the likelihood of getting a reward from that option. In Thompson sampling, we use the Beta distribution for this purpose.
2. **Sample a value**: from these distributions, we randomly sample a value for each option. These sampled values represent our current guess about the expected reward from each option. This step introduces randomness, ensuring that we occasionally try out arms that have fewer successes but might have potential.
3. **Choose the best option**: we select the option with the highest sampled value. This means that we are more likely to choose options that have shown higher rewards in the past, but we still occasionally try out other options to gather more information about them.

Formulation:

1. **Beta distribution**: the reward probability of each arm is assumed to follow a Beta distribution. The Beta distribution is a family of continuous probability distributions defined on the interval `[0, 1]`, parameterized by two positive shape parameters, α and β. The distribution represents the posterior probability of the success rate of a Bernoulli trial (a trial with two outcomes, success or failure). The first parameter, α_k represents the number of successes (rewards), while β_k represents the number of failures (no rewards) for arm k. For example, if we have an arm (option) with $\alpha = 5$ and $\beta = 3$, the Beta distribution would give us a distribution of possible success probabilities for this arm, based on the 5 successes and 3 failures observed so far. When we get new data, we can easily update the Beta distribution with simple calculations. This is because the Beta distribution is a "conjugate prior" for the Bernoulli distribution, which means their mathematical properties align perfectly for easy updates.

2. **Updating the distribution**: when we pull an arm k and observe a reward $r_k(t)$, we update the parameters of the Beta distribution as follows:

   - If we get a reward ($r_k(t) = 1$), we increase α_k by 1, indicating one more success:

   $$\alpha_k = \alpha_k + r_k(t) \tag{28.5}$$

   - If we do not get a reward ($r_k(t) = 0$), we increase β_k by 1, indicating one more failure:

   $$\beta_k = \beta_k + 1 - r_k(t) \tag{28.6}$$

In practice: Initially, both α and β start at zero since we have not observed any rewards or failures yet. However, the Beta distribution with parameters $\alpha = 0$ and $\beta = 0$ is undefined because the Beta distribution requires positive parameters. So, we add 1 to each of these two values:

- **Initial prior**: by starting with $\alpha = 1$ and $\beta = 1$, we assume a weak prior belief that each arm has an equal probability of success and failure. This is often referred to as a "non-informative" or "uniform" prior.
- **Practicality**: it ensures that the Beta distribution is always defined, allowing us to sample from it even before any data is observed.
- **Conjugate prior**: When we update the Beta distribution with observed data, adding 1 ensures that our initial prior belief is combined with the observed data correctly.

Suppose we have an arm that has been pulled 10 times, resulting in 7 successes and 3 failures:

- Successes: 7
- Failures: 3

For Thompson sampling:

- The α parameter (successes) is $7 + 1 = 8$
- The β parameter (failures) is $3 + 1 = 4$

This gives us a Beta distribution with parameters $\alpha = 8$ and $\beta = 4$, from which we can sample to estimate the probability of success for this arm.

Implementation:

```
# Thompson Sampling algorithm
def thompson_sampling(arms, n_rounds=1000):
    n_arms = len(arms)
    successes = np.zeros(n_arms)
    failures = np.zeros(n_arms)
    total_rewards = []

    for _ in range(n_rounds):
        sampled_probs = [np.random.beta(successes[i] + 1, failures[i] + 1) for i in␣
↪range(n_arms)]
        arm = np.argmax(sampled_probs)

        # Simulate pulling the arm
        reward = np.random.rand() < arms[arm]
        if reward:
            successes[arm] += 1
        else:
            failures[arm] += 1
        total_rewards.append(reward)

    return np.cumsum(total_rewards)
```

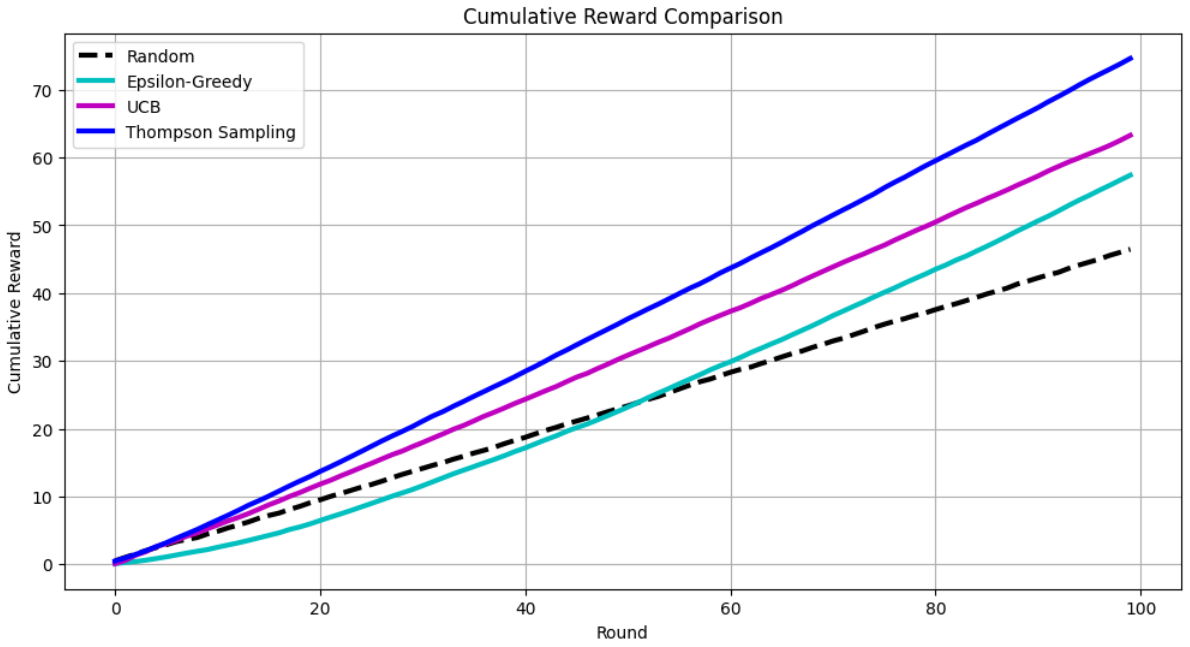
28.3.4 Comparing the three algorithms with random sampling

Now, we can run a comprehensive example where we compare the performance of epsilon-greedy, UCB, and Thompson sampling against a random strategy.

```
# Define the true reward probabilities for each arm
true_means = [0.1, 0.5, 0.8]
n_rounds = 100
n_simulations = 100

# Simulation to collect rewards
def simulate(algorithm, arms, n_simulations=100, n_rounds=1000, **kwargs):
    rewards = np.zeros((n_simulations, n_rounds))
    for i in range(n_simulations):
        rewards[i, :] = algorithm(arms, n_rounds=n_rounds, **kwargs)
    return rewards

# Run simulations
rewards_random = simulate(epsilon_greedy, true_means, n_simulations, n_rounds,␣
↪epsilon=1.0)  # Pure exploration (random)
rewards_epsilon_greedy = simulate(epsilon_greedy, true_means, n_simulations, n_rounds,
↪ epsilon=0.1)
rewards_ucb = simulate(ucb, true_means, n_simulations, n_rounds, c=2)
rewards_thompson = simulate(thompson_sampling, true_means, n_simulations, n_rounds)
```

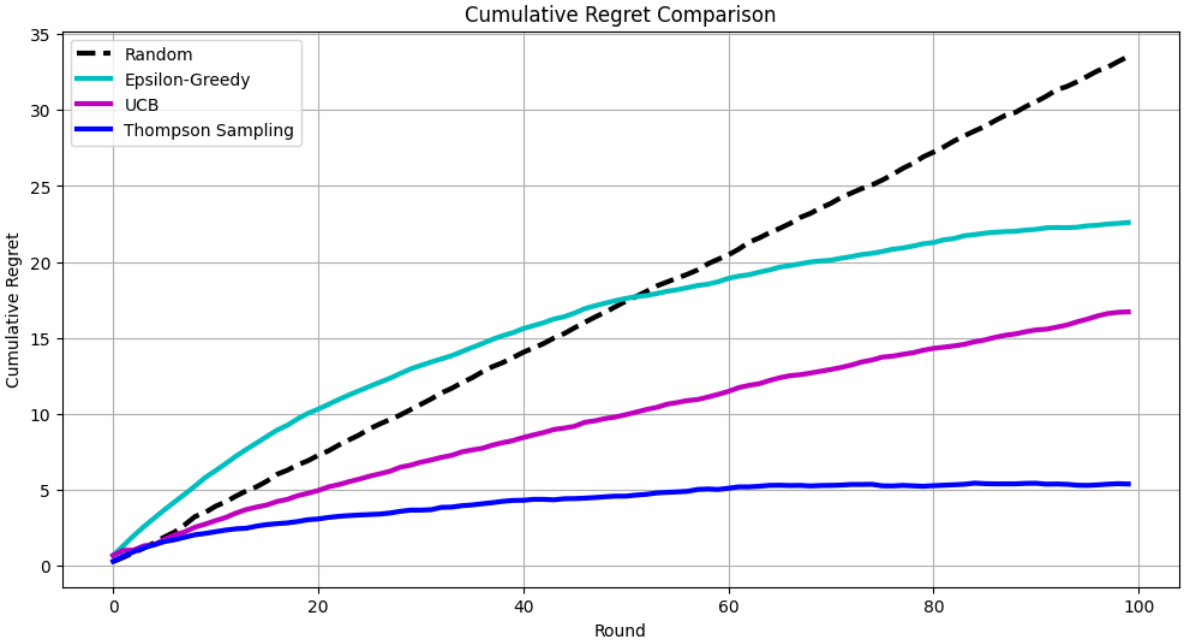
We can the compare them in terms of **cumulative reward**, showing how the total reward accumulates over time for each algorithm.

```
# Calculate mean cumulative rewards
mean_rewards_random = np.mean(rewards_random, axis=0)
mean_rewards_epsilon_greedy = np.mean(rewards_epsilon_greedy, axis=0)
mean_rewards_ucb = np.mean(rewards_ucb, axis=0)
mean_rewards_thompson = np.mean(rewards_thompson, axis=0)

# Plot cumulative rewards
plt.figure(figsize=(12, 6))
plt.plot(mean_rewards_random, label='Random', linestyle='--', lw=3, c='k')
plt.plot(mean_rewards_epsilon_greedy, label='Epsilon-Greedy', lw=3, c='c')
plt.plot(mean_rewards_ucb, label='UCB', lw=3, c='m')
plt.plot(mean_rewards_thompson, label='Thompson Sampling', lw=3, c='b')
plt.xlabel('Round')
plt.ylabel('Cumulative Reward')
plt.title('Cumulative Reward Comparison')
plt.legend()
plt.grid(True)
plt.show()
```



and here in terms of **regret**, showing the difference between the reward of the optimal machine and the reward obtained by each algorithm.

```
# Calculate regret
optimal_mean = max(true_means)
regret_random = optimal_mean * np.arange(1, n_rounds + 1) - mean_rewards_random
regret_epsilon_greedy = optimal_mean * np.arange(1, n_rounds + 1) - mean_rewards_
↪epsilon_greedy
regret_ucb = optimal_mean * np.arange(1, n_rounds + 1) - mean_rewards_ucb
regret_thompson = optimal_mean * np.arange(1, n_rounds + 1) - mean_rewards_thompson

# Plot regret
plt.figure(figsize=(12, 6))
plt.plot(regret_random, label='Random', linestyle='--', lw=3, c='k')
plt.plot(regret_epsilon_greedy, label='Epsilon-Greedy', lw=3, c='c')
plt.plot(regret_ucb, label='UCB', lw=3, c='m')
plt.plot(regret_thompson, label='Thompson Sampling', lw=3, c='b')
plt.xlabel('Round')
plt.ylabel('Cumulative Regret')
plt.title('Cumulative Regret Comparison')
plt.legend()
plt.grid(True)
plt.show()
```



Through these plots, we can visually compare the efficiency and performance of different algorithms in balancing exploration and exploitation, highlighting the advantages of more sophisticated MAB strategies over a random approach.

CHAPTER

TWENTYNINE

DESIGN OF EXPERIMENTS

Design of experiments (DoE) is a systematic approach to planning, conducting, analysing, and interpreting controlled experiments [Mon17]. It aims to ensure that the results are valid, reliable, and can be used to draw causal conclusions. DoE is essential in various fields, including engineering, medicine, agriculture, and computer science.

The **key concepts** of DoE include:

- **Randomisation**: Randomly assigning subjects to different treatment groups to eliminate bias.
- **Replication**: Repeating the experiment to estimate variability and ensure reliability.
- **Blocking**: Grouping similar experimental units and randomising within these blocks to reduce variability.

A good mantra for dealing with the presence of confounders is **block what you can, randomise what you cannot**. What do we mean by this? In experimental design, controlling for confounders is crucial to ensure that the results are valid and reliable. Confounders are variables that can affect both the independent variable and the dependent variable, potentially leading to biased or incorrect conclusions about the relationship between these variables. The mantra "block what you can, randomise what you cannot" provides a practical approach to managing confounding variables.

1. **Block what you can**: blocking is a technique used to control for known sources of variability that are not of primary interest but could influence the outcome of the experiment. By grouping similar experimental units into blocks, we can account for these sources of variability and reduce their impact on the results. For example, in an experiment to test the efficiency of different battery storage systems, we might block by the type of battery used. Each block would contain all types of batteries, ensuring that the variation due to the battery type is controlled for and does not confound the results. Blocking helps isolate the effect of the primary factors being studied by accounting for the variability due to other known factors.
2. **Randomise what you cannot**: randomisation is used to control for unknown or uncontrollable sources of variability. By randomly assigning experimental units to different treatment groups, we ensure that these sources of variability are evenly distributed across all groups, reducing the risk of bias. For example, if we are conducting an experiment to test different charging rates and temperatures on battery efficiency, and we cannot control the environmental conditions precisely, we can randomly assign the charging rates and temperatures to different batteries. This way, any unknown or uncontrollable factors (like slight variations in ambient temperature) are equally likely to affect all treatment groups, minimising their impact on the results. Randomisation helps ensure that the treatment groups are comparable and that the observed effects are due to the treatments themselves rather than other extraneous factors.

29.1 Some important designs

In most cases, DoE is used to analyse the relationship between the input and output factors of a process. Various tests are performed to see the effect of varying levels of input factors on the response. For example, in the context of electricity markets, we might be interested in designing an experiment to understand the effect of charge rate and temperature on the efficiency of a battery storage system. By systematically varying these factors, researchers can identify optimal operating conditions and interactions between variables. The critical aspect of DoE is the way the factor levels are varied throughout the experiment.

29.1.1 Factorial Designs

One of the most common types of design is the factorial design, where in each complete trial or replicate of the experiment all possible combinations of the levels of the factors are investigated. A 2^K **factorial design** is a special case of factorial design where each of the K factors is studied at two levels, often referred to as low (-1) and high (1). This design is highly efficient for exploring the main effects and interactions among factors. In the context of a battery storage system, suppose we are interested in the effects of two factors: charge rate and temperature. Each factor can be set at two levels: low (-1) and high (1). This is the data that would be collected using a 2^2 factorial design, without replications:

Run	Charge Rate (C)	Temperature (°C)	Efficiency (%)
1	-1	-1	85
2	-1	1	80
3	1	-1	90
4	1	1	75

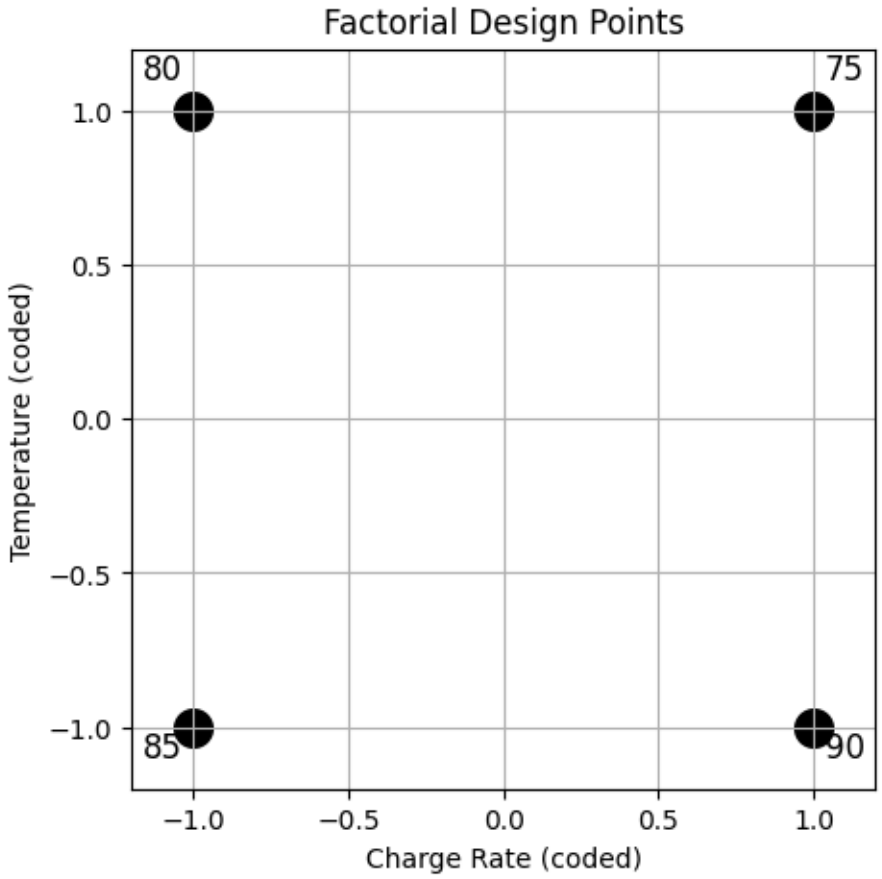
The figure below shows a factorial experiment where both charge rate and temperature can be set at two levels. The black dots represent the points where the tests are performed, namely where the response is measured.

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

# Data for factorial design
data = pd.DataFrame({
    'Charge_Rate': [-1, -1, 1, 1],
    'Temperature': [-1, 1, -1, 1],
    'Efficiency': [85, 80, 90, 75]
})

# Generate grid for plotting
x1 = np.array([-1, 1])
x2 = np.array([-1, 1])
X1, X2 = np.meshgrid(x1, x2)

# Plotting the factorial design points
plt.figure(figsize=(5, 5))
plt.scatter(data['Charge_Rate'], data['Temperature'], color='black', s=200)
plt.xlabel('Charge Rate (coded)')
plt.ylabel('Temperature (coded)')
plt.title('Factorial Design Points')
plt.grid(True)

# Annotating the points with their efficiency values
```

(continues on next page)

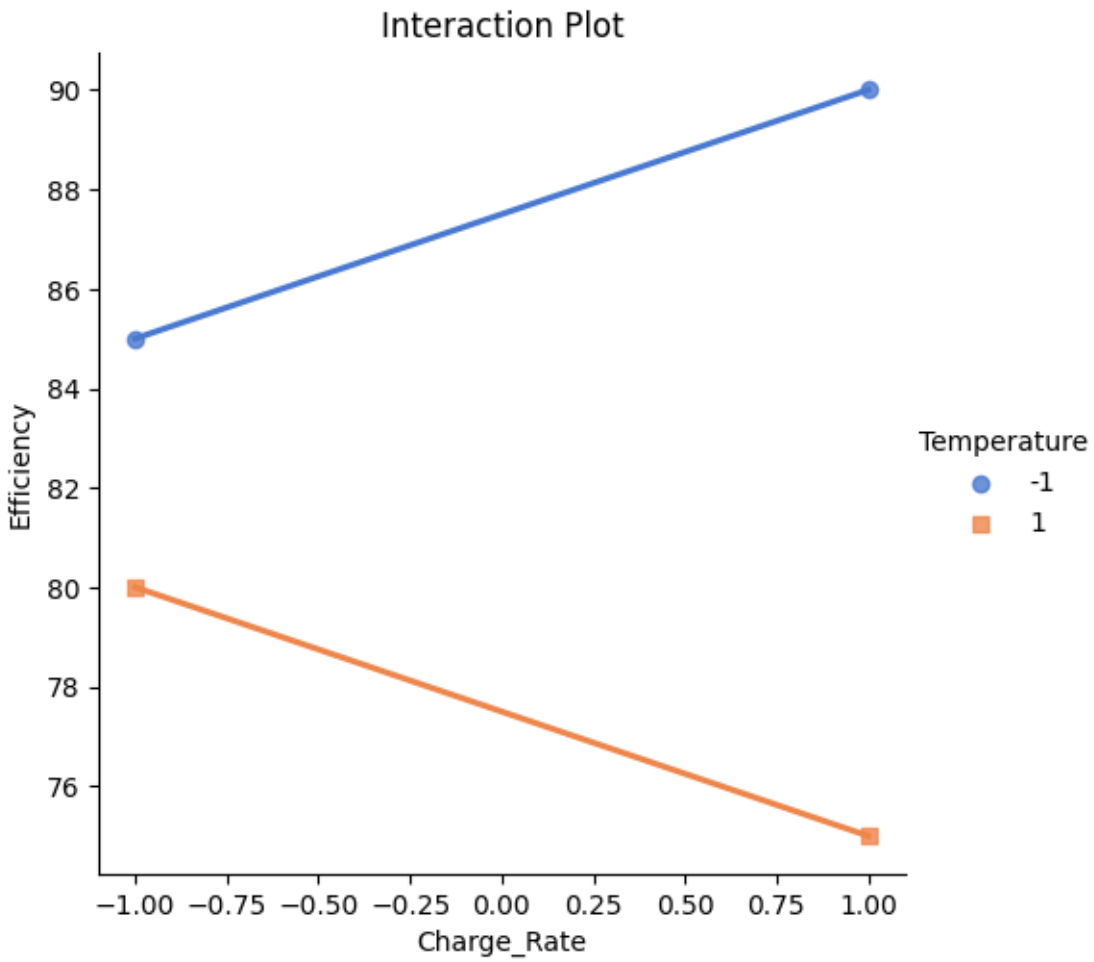
(continued from previous page)

```
for i, txt in enumerate(data['Efficiency']):
    plt.annotate(txt, (data['Charge_Rate'][i]*1.1, data['Temperature'][i]*1.1),␣
 ↪fontsize=12, color='k', ha='center')
plt.ylim(-1.2, 1.2)
plt.xlim(-1.2, 1.2)
plt.show()
```



The effect of a factor is defined as the change in response produced by a change in the level of the factor. Assuming linearity in the factor effects within the design space, an ordinary least squares (OLS) regression model is usually fit on the experimental data.

```
from statsmodels.formula.api import ols
import seaborn as sns

# Fit the model
model = ols('Efficiency ~ Charge_Rate * Temperature', data=data).fit()
# print(model.summary())

# Plotting the interaction
sns.lmplot(x='Charge_Rate', y='Efficiency', hue='Temperature', data=data, markers=['o
 ↪', 's'], ci=None, palette='muted', height=5, aspect=1)
plt.title('Interaction Plot')
plt.show()
```



The interaction plot visualises the relationship between the charge rate and the efficiency of the battery storage system, with temperature as the moderating variable. Each line in the plot represents the effect of charge rate on efficiency at a specific temperature level, indicated by different colors and markers. The x-axis shows the coded values for charge rate (with -1 representing a low charge rate and 1 representing a high charge rate), while the y-axis displays the battery efficiency in percentage.

From this plot, readers can observe how the efficiency changes with varying charge rates and how this relationship is influenced by temperature.

- If the lines for different temperatures are **parallel**, it indicates that there is no interaction between charge rate and temperature—each factor independently affects efficiency.
- If the lines are **not parallel**, this suggests an interaction effect, meaning the impact of charge rate on efficiency depends on the temperature level. For instance, one might notice that increasing the charge rate significantly boosts efficiency at a lower temperature but not at a higher temperature. This insight is crucial for optimising the battery storage system, as it highlights the importance of considering both factors together rather than in isolation.

Overall, the interaction plot provides a clear and intuitive way to understand complex relationships between multiple factors, helping researchers and practitioners to make informed decisions about optimising the system's performance.

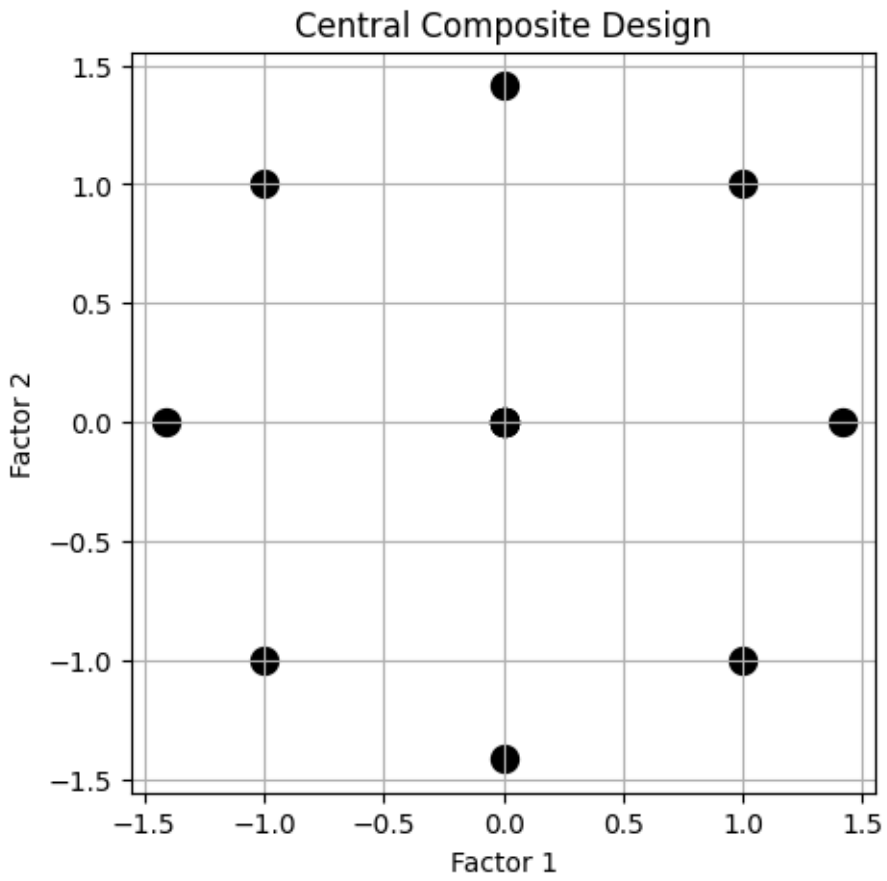
29.1.2 Central composite designs

The central composite design (CCD) is an extension of factorial design used for building a second-order (quadratic) model without needing to perform a complete three-level factorial experiment. It includes:

- **Factorial points**: these are points from a standard 2^k factorial design, where k is the number of factors. The factorial points allow for the estimation of the main effects and interaction effects.
- **Centre points**: these are the midpoints of all factor levels and are replicated to provide an estimate of the experimental error. Centre points help detect curvature in the response surface.
- **Axial points**: these points are added at a distance α from the centre along each axis of the factor space. Axial points enable the estimation of the quadratic terms, providing the necessary information to fit a second-order model.

The CCD is highly efficient for exploring the quadratic response surface and identifying the optimal settings of the input factors. By combining these points, CCDs provide a balanced and comprehensive design for understanding the effects of factors and their interactions. Here you can see an example of how to obtain a CCD by adding centre points and axial points to a regular 2^k factorial design.

```
from pyDOE2 import ccdesign

# Create a CCD design
ccd = ccdesign(2, center=(4, 4))

# Plot the CCD design
plt.figure(figsize=(5, 5))
plt.scatter(ccd[:, 0], ccd[:, 1], c='k', linewidths=5)
plt.xlabel('Factor 1')
plt.ylabel('Factor 2')
plt.title('Central Composite Design')
plt.grid(True)
plt.show()
```



Let's now consider an example where we want to use a CCD to build a model for optimising the output of a solar power plant based on two controllable variables: cleaning frequency (x_1) and tilt angle (x_2). Using a CCD and measuring the value of the response (y) at the design locations, we would collect this dataset:

Run	Cleaning Frequency (times/month)	Tilt Angle (degrees)	Output (kWh)
1	-1	-1	500
2	-1	1	520
3	1	-1	540
4	1	1	560
5	0	0	550
6	0	0	570
7	-1.414	0	530
8	1.414	0	580
9	0	-1.414	540
10	0	1.414	580

```
from statsmodels.tools import add_constant
from mpl_toolkits.mplot3d import Axes3D

# Data for CCD
data = pd.DataFrame({
    'Cleaning_Frequency': [-1, -1, 1, 1, 0, 0, -1.414, 1.414, 0, 0],
    'Tilt_Angle': [-1, 1, -1, 1, 0, 0, 0, 0, -1.414, 1.414],
```

(continues on next page)

(continued from previous page)

```
    'Output': [500, 520, 540, 560, 550, 570, 530, 580, 540, 580]
})

# Define the design matrix for the quadratic model
X = np.column_stack((data['Cleaning_Frequency'], data['Tilt_Angle'],
                     data['Cleaning_Frequency']**2, data['Tilt_Angle']**2,
                     data['Cleaning_Frequency']*data['Tilt_Angle']))

# Add constant (intercept)
X = add_constant(X)

# Fit the model
model = OLS(data['Output'], X).fit()

# Prepare data for 3D surface plot
x1 = np.linspace(-1.5, 1.5, 30)
x2 = np.linspace(-1.5, 1.5, 30)
X1, X2 = np.meshgrid(x1, x2)
X1_flat = X1.flatten()
X2_flat = X2.flatten()

# Create design matrix for predictions
X_pred = np.column_stack((np.ones_like(X1_flat), X1_flat, X2_flat,
                          X1_flat**2, X2_flat**2, X1_flat*X2_flat))

# Predict the response
Y_pred = model.predict(X_pred).reshape(X1.shape)

# Plot the response surface
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X1, X2, Y_pred, cmap='viridis', edgecolor='none')
ax.set_xlabel('Cleaning Frequency (times/month)')
ax.set_ylabel('Tilt Angle (degrees)')
ax.set_zlabel('Output (kWh)')
plt.title('Response Surface using CCD')
plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 20
     17 X = add_constant(X)
     19 # Fit the model
---> 20 model = OLS(data['Output'], X).fit()
     22 # Prepare data for 3D surface plot
     23 x1 = np.linspace(-1.5, 1.5, 30)

NameError: name 'OLS' is not defined
```

We can see:

- **Quadratic effects**: the CCD allows for the estimation of both linear and quadratic effects, providing a more accurate representation of the response surface.
- **Optimisation**: by examining the response surface plot, we can identify the optimal settings for cleaning frequency and tilt angle to maximise the solar power plant's output.

- **Balanced design**: the inclusion of centre and axial points ensures that the design is balanced and provides sufficient information to detect curvature in the response surface.

This is a simple example of response surface methodology (RSM) [MMAC16], which is a collection of statistical and mathematical techniques useful for developing, improving, and optimizing processes . It uses quantitative data from appropriate experiments to determine regression models and identify the optimal conditions. The goal is to optimize this response by finding the best settings of the input variables.

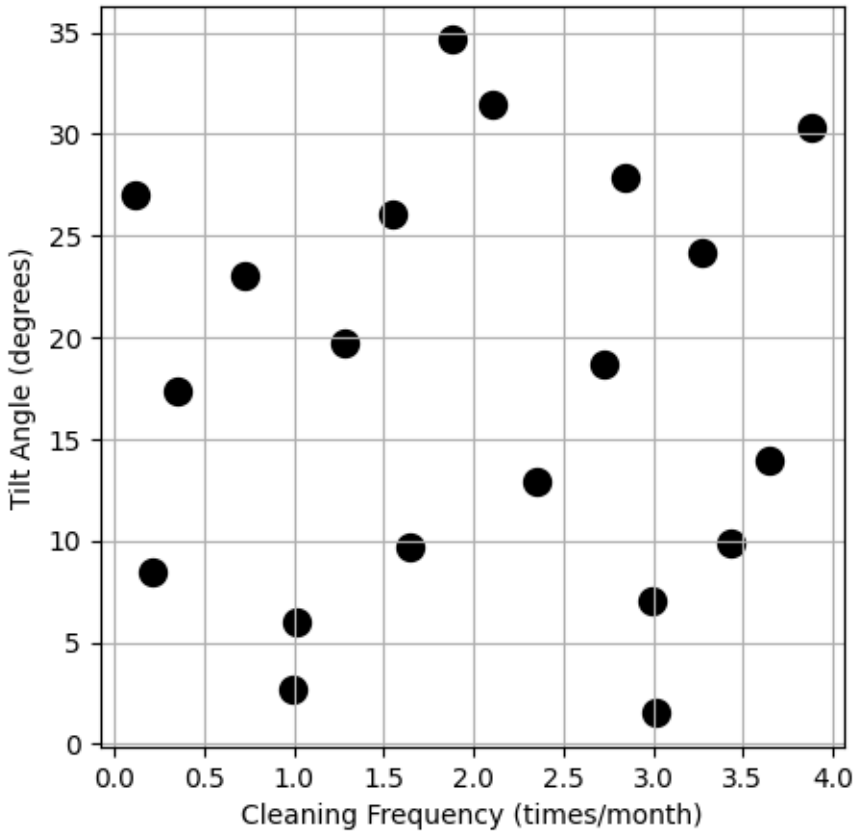
29.1.3 Space-filling designs

Space-filling designs aim to cover the experimental space uniformly. This is particularly important for **computer experiments** [SWNW03] where the objective is to explore the entire input space efficiently. Unlike physical experiments where replication helps estimate variability and improve reliability, computer experiments often use deterministic simulators, making replication less useful. Instead, the focus is on spreading out the points as much as possible to gain comprehensive insights across the entire input space.

Sobol sequences

One example of designs for computer experiments are Sobol sequences. Sobol sequences are a type of low-discrepancy sequence used to generate space-filling designs. They provide a quasi-random sequence that covers the input space uniformly. Sobol sequences are particularly useful in high-dimensional spaces and are known for their good uniformity properties, making them ideal for numerical integration and simulation tasks.

```
from scipy.stats import qmc

# Define the number of points and dimensions (2 in this case: Cleaning Frequency and␣
 ↪Tilt Angle)
n_points = 20
dimensions = 2

# Generate Sobol sequence
sobol = qmc.Sobol(d=dimensions, scramble=True)
points = sobol.random(n=n_points)

# Scale points to the desired range (e.g., Cleaning Frequency: 0-4 times/month, Tilt␣
 ↪Angle: 0-35 degrees)
cleaning_frequency = points[:, 0] * 4  # Scale to 0-4
tilt_angle = points[:, 1] * 35         # Scale to 0-35

# Plot the space-filling design
plt.figure(figsize=(5, 5))
plt.scatter(cleaning_frequency, tilt_angle, c='k', linewidths=5)
plt.xlabel('Cleaning Frequency (times/month)')
plt.ylabel('Tilt Angle (degrees)')
plt.grid(True)
plt.show()
```



Latin hypercubes

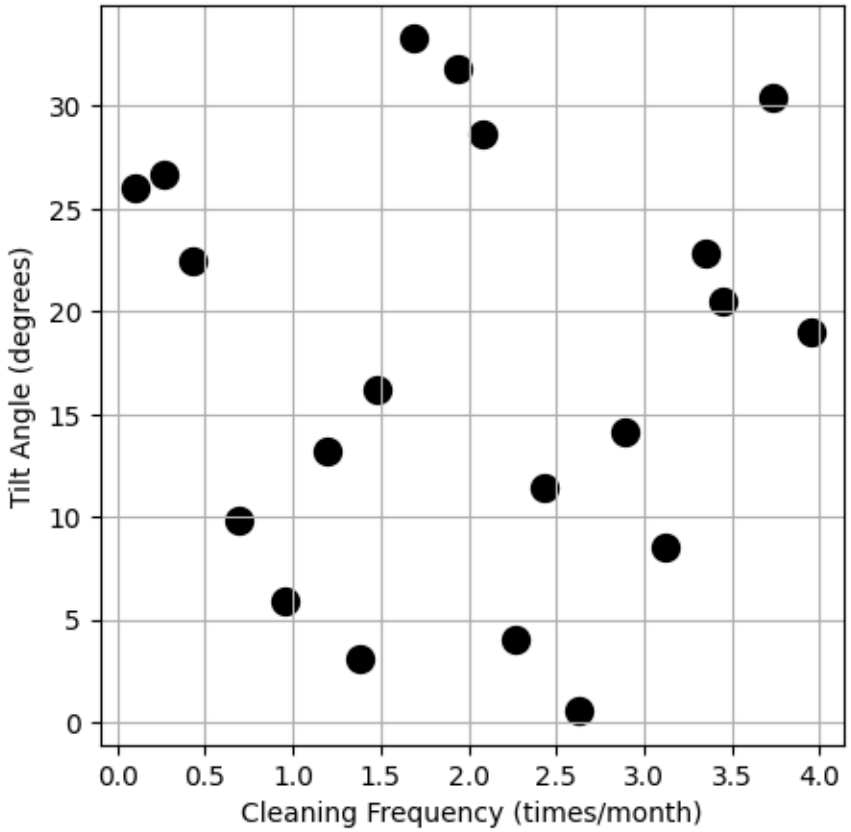
Latin Hypercube Sampling (LHS) is another method used to create space-filling designs. It divides the range of each input variable into equal intervals and ensures that each interval is sampled exactly once. This approach ensures a more uniform coverage of the input space compared to simple random sampling. LHS is particularly effective in reducing variance and ensuring that the sample points are well-distributed across the entire range of each input variable.

```
from scipy.stats import qmc

# Define the number of points and dimensions (2 in this case: Cleaning Frequency and␣
 ↪Tilt Angle)
n_points = 20
dimensions = 2

# Generate Latin Hypercube Sampling points
lhs = qmc.LatinHypercube(d=dimensions)
points = lhs.random(n=n_points)

# Scale points to the desired range (e.g., Cleaning Frequency: 0-4 times/month, Tilt␣
 ↪Angle: 0-35 degrees)
cleaning_frequency = points[:, 0] * 4  # Scale to 0-4
tilt_angle = points[:, 1] * 35         # Scale to 0-35

# Plot the space-filling design
plt.figure(figsize=(5, 5))
plt.scatter(cleaning_frequency, tilt_angle, c='k', linewidths=5)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Cleaning Frequency (times/month)')
plt.ylabel('Tilt Angle (degrees)')
plt.grid(True)
plt.show()
```



Space-filling designs are essential for computer experiments where the goal is to explore the entire input space efficiently. By ensuring uniform coverage, these designs provide comprehensive insights into the system's behaviour, improving the accuracy and robustness of predictive models. Both Sobol sequences and Latin Hypercube Sampling are effective methods for generating space-filling designs, each with its unique advantages in providing uniform and well-distributed sample points across the input space.

29.2 Design optimality

Design optimality involves selecting an experimental design that provides the most information about the parameters of interest with the least experimental effort. Optimal designs are tailored to achieve specific statistical goals, ensuring that experiments are both efficient and informative. Here are some common criteria used to achieve design optimality:

- **D-optimality**: This criterion maximizes the determinant of the information matrix. A D-optimal design provides the most precise parameter estimates by ensuring that the volume of the confidence ellipsoid for the estimated parameters is minimized. In essence, it spreads the experimental points in such a way that they collectively capture as much information as possible about the parameters.
- **A-optimality**: This criterion minimizes the trace of the inverse information matrix. By focusing on reducing the average variance of the parameter estimates, A-optimal designs aim to make the overall estimation process more

efficient. This means that the parameters can be estimated with lower average uncertainty.

- **G-optimality**: This criterion minimizes the maximum prediction variance within the design space. G-optimal designs are concerned with ensuring that the model's predictions are as accurate as possible across the entire experimental region. This approach is particularly useful when the goal is to have a model that performs well uniformly across the design space.

CHAPTER

THIRTY

ACTIVE LEARNING

Active learning is a machine learning technique that aims to select the most informative data points for training a model. In the context of causal inference, active learning can be particularly useful for efficiently estimating parameters when data collection is limited or expensive. By selectively choosing the most informative samples, we can improve the estimation accuracy of causal effects with fewer data points.

Active learning involves iteratively selecting the most informative data points to be labeled and added to the training set. The main scenarios of active learning include:

1. **Pool-based active learning**: starting with a large pool of unlabeled data, the algorithm selects the most informative samples to label.
2. **Stream-based active learning**: samples arrive in a stream, and the algorithm decides whether to label each incoming sample.
3. **Query synthesis**: the algorithm generates new samples to query an oracle for their labels.

Consider the case where we want to estimate the effect of a variable using a linear model of the kind:

$$y = X\beta + \epsilon \tag{30.1}$$

where:

- y is the response variable.
- X is the design matrix containing the input variables.
- β is the vector of coefficients.
- ϵ is the error term, assumed to be normally distributed with mean zero and variance σ^2.

The goal is to estimate the coefficients β using the observed data. The variance of the coefficients β can be computed as:

$$\text{Var}(\beta) = \sigma^2 (X^T X)^{-1} \tag{30.2}$$

where σ^2 is the variance of the residuals, and $(X^T X)^{-1}$ is the inverse of the information matrix.

In the context of linear models, **prediction variance** is a measure of the uncertainty associated with the predictions made by the model. In active learning, we can use the prediction variance to identify the most informative data points. The idea is to select the points that, when added to the training set, will most reduce the uncertainty in the model's predictions. This is connected with the concept of **D-optimality** discussed in the previous chapter. Indeed, using active learning to select the most informative data points based on prediction variance is analogous to D-optimality. By choosing points that maximize the reduction in prediction variance, we effectively seek to create a design that is close to D-optimal.

Let's create a simple function to compute the prediction variance for a linear model

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Function to compute variance of coefficients
def compute_variance(X, sigma_squared):
    XTX_inv = np.linalg.inv(X.T @ X)
    return sigma_squared * XTX_inv
```

Next, we define the function that performs one iteration of active learning. Now, we assume we are in the **pool-based** setting of active learning.

```
# Function to perform active learning iteration
def active_learning_iteration(X_initial, y_initial, X_pool, y_pool):
    # Train linear regression model
    model = LinearRegression().fit(X_initial, y_initial)
    y_pred_initial = model.predict(X_initial)
    residuals = y_initial - y_pred_initial
    sigma_squared = np.var(residuals)
    variance = compute_variance(X_initial, sigma_squared)

    # Select the most informative data points (highest prediction variance)
    prediction_variances = sigma_squared * np.sum((X_pool @ np.linalg.inv(X_initial.T␣
 ↪@ X_initial)) * X_pool, axis=1)
    most_informative_idx = np.argmax(prediction_variances)
    X_most_informative = X_pool[most_informative_idx]
    y_most_informative = y_pool[most_informative_idx]

    # Add to the labeled dataset
    X_initial = np.vstack((X_initial, X_most_informative))
    y_initial = np.append(y_initial, y_most_informative)

    # Remove from the pool
    X_pool = np.delete(X_pool, most_informative_idx, axis=0)
    y_pool = np.delete(y_pool, most_informative_idx, axis=0)

    return X_initial, y_initial, X_pool, y_pool, np.mean(np.diag(variance))
```

What do we do, iteratively, in the active learning routine:

- **Training the model**: we fit a linear regression model to the initial labeled dataset.
- **Computing residuals and variance**: the residuals (differences between actual and predicted values) are computed to estimate the variance σ^2 of the residuals.
- **Selecting the most Informative data point**: the prediction variances are computed for the pool of unlabeled data points. The point with the highest prediction variance is selected as the most informative.
- **Updating the labeled dataset**: the selected data point is added to the labeled dataset, and it is removed from the pool.

For comparison, we also define a random sampling approach, where instead of selecting points with high prediction variance, we just sample data at random from the pool.

```
# Function to perform random sampling iteration
def random_sampling_iteration(X_initial, y_initial, X_pool, y_pool):
    # Train linear regression model
    model = LinearRegression().fit(X_initial, y_initial)
```

(continues on next page)

(continued from previous page)

```
    y_pred_initial = model.predict(X_initial)
    residuals = y_initial - y_pred_initial
    sigma_squared = np.var(residuals)
    variance = compute_variance(X_initial, sigma_squared)

    # Randomly select a data point
    random_idx = np.random.choice(range(X_pool.shape[0]))
    X_random = X_pool[random_idx]
    y_random = y_pool[random_idx]

    # Add to the labeled dataset
    X_initial = np.vstack((X_initial, X_random))
    y_initial = np.append(y_initial, y_random)

    # Remove from the pool
    X_pool = np.delete(X_pool, random_idx, axis=0)
    y_pool = np.delete(y_pool, random_idx, axis=0)

    return X_initial, y_initial, X_pool, y_pool, np.mean(np.diag(variance))
```

Now, we define the function to run the entire active learning simulation, comparing active learning with random sampling over multiple iterations and runs.

```
# Function to perform the full active learning simulation
def run_active_learning_simulation_with_data_regeneration(n_runs=10, n_iterations=20):
    variances_active_runs = []
    variances_random_runs = []

    for run in range(n_runs):  # Running multiple simulations with different seeds
        # Generate synthetic data
        np.random.seed(run)
        X = np.random.rand(1000, 2)
        true_coefficients = np.array([1.5, -2.0])
        y = X @ true_coefficients + np.random.randn(1000) * 0.5

        # Initial small labeled dataset
        initial_indices = np.random.choice(range(1000), size=10, replace=False)
        X_initial, y_initial = X[initial_indices], y[initial_indices]

        # Remaining pool of unlabeled data
        remaining_indices = np.setdiff1d(range(1000), initial_indices)
        X_pool, y_pool = X[remaining_indices], y[remaining_indices]

        # Initialize data for random sampling
        X_initial_random, y_initial_random = X_initial.copy(), y_initial.copy()
        X_pool_random, y_pool_random = X_pool.copy(), y_pool.copy()

        variances_active = []
        variances_random = []

        for iteration in range(n_iterations):
            # Active learning iteration
            X_initial, y_initial, X_pool, y_pool, avg_variance_active = active_
↪learning_iteration(X_initial, y_initial, X_pool, y_pool)
            variances_active.append(avg_variance_active)
```

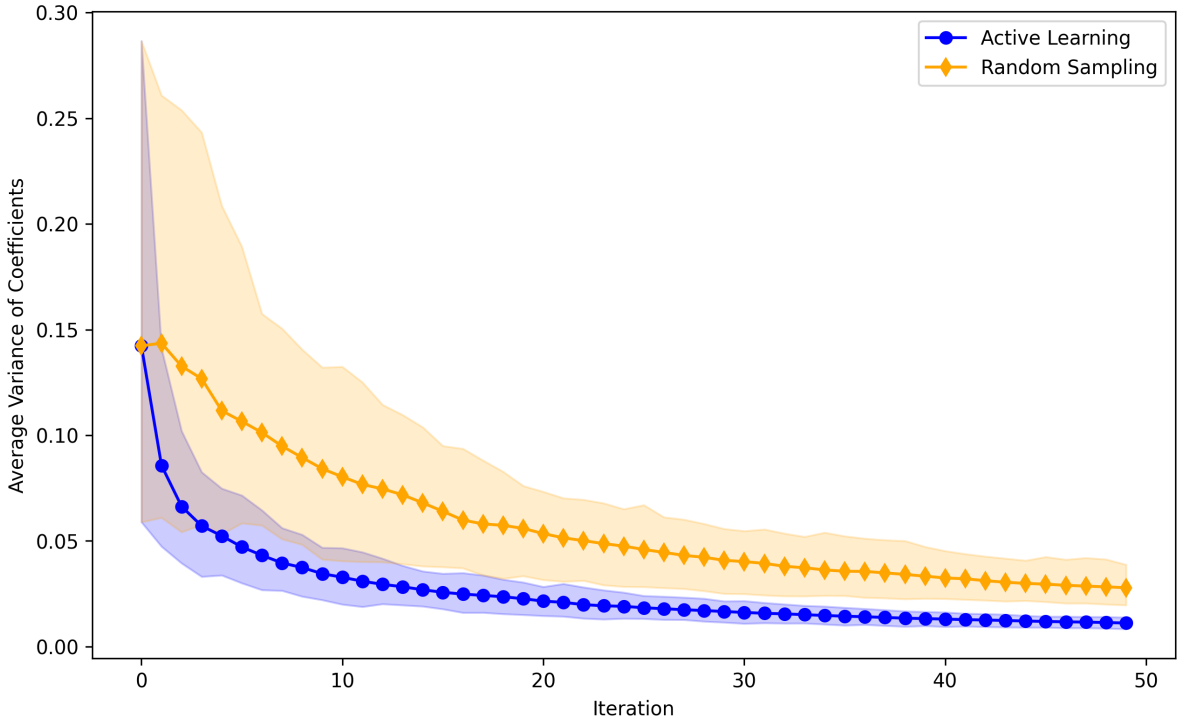
(continues on next page)

(continued from previous page)

```
            # Random sampling iteration
            X_initial_random, y_initial_random, X_pool_random, y_pool_random, avg_
 ↪variance_random = random_sampling_iteration(X_initial_random, y_initial_random, X_
 ↪pool_random, y_pool_random)
            variances_random.append(avg_variance_random)

        variances_active_runs.append(variances_active)
        variances_random_runs.append(variances_random)

    return np.array(variances_active_runs), np.array(variances_random_runs)
```

Finally, we run the simulation and plot the learning curves for active learning and random sampling.

```
# Run the simulation
variances_active_runs, variances_random_runs = run_active_learning_simulation_with_
 ↪data_regeneration(n_runs=50, n_iterations=50)

# Compute means and confidence intervals
mean_variances_active = np.mean(variances_active_runs, axis=0)
mean_variances_random = np.mean(variances_random_runs, axis=0)

ci_variances_active_5 = np.percentile(variances_active_runs, 10, axis=0)
ci_variances_active_95 = np.percentile(variances_active_runs, 90, axis=0)
ci_variances_random_5 = np.percentile(variances_random_runs,10, axis=0)
ci_variances_random_95 = np.percentile(variances_random_runs, 90, axis=0)

# Plot the learning curves with shaded regions for confidence intervals
n_iterations = 50
plt.figure(figsize=(10, 6), dpi=300)
plt.plot(mean_variances_active, label='Active Learning', marker='o', color='blue')
plt.fill_between(range(n_iterations), ci_variances_active_5, ci_variances_active_95,␣
 ↪color='blue', alpha=0.2)
plt.plot(mean_variances_random, label='Random Sampling', marker='d', color='orange')
plt.fill_between(range(n_iterations), ci_variances_random_5, ci_variances_random_95,␣
 ↪color='orange', alpha=0.2)
plt.xlabel('Iteration')
plt.ylabel('Average Variance of Coefficients')
plt.legend()
plt.show()
```



We can see how active learning is a powerful technique for efficiently estimating parameters in causal inference, especially when data collection is limited or expensive. By selectively choosing the most informative samples, active learning can significantly improve the estimation accuracy with fewer data points compared to random sampling. This tutorial demonstrated the application of active learning in a linear regression context, highlighting its advantages over random sampling.

Part VIII

Other

CHAPTER

THIRTYONE

REFERENCES

BIBLIOGRAPHY

[BIM20] Derek W Bunn, John N Inekwe, and David MacGeehan. Analysis of the fundamental predictability of prices in the british balancing market. *IEEE Transactions on Power Systems*, 36(2):1309–1316, 2020.

[CCD+18] Victor Chernozhukov, Denis Chetverikov, Mert Demirer, Esther Duflo, Christian Hansen, Whitney Newey, and James Robins. Double/debiased machine learning for treatment and structural parameters. *The Econometrics Journal*, 21(1):C1–C68, 2018.

[GXBH15] Huan Gui, Ya Xu, Anmol Bhasin, and Jiawei Han. Network a/b testing: from sampling to estimation. In *Proceedings of the 24th International Conference on World Wide Web*, 399–409. 2015.

[HyvarinenSH08] Aapo Hyvärinen, Shohei Shimizu, and Patrik O Hoyer. Causal modelling combining instantaneous and lagged effects: an identifiable model based on non-gaussianity. In *Proceedings of the 25th international conference on Machine learning*, 424–431. 2008.

[HyvarinenZSH10] Aapo Hyvärinen, Kun Zhang, Shohei Shimizu, and Patrik O Hoyer. Estimation of a structural vector autoregression model using non-gaussianity. *Journal of Machine Learning Research*, 2010.

[IIZ+23] Takashi Ikeuchi, Mayumi Ide, Yan Zeng, Takashi Nicholas Maeda, and Shohei Shimizu. Python package for causal discovery based on lingam. *Journal of Machine Learning Research*, 24(14):1–8, 2023.

[JonssonPM10] Tryggvi Jónsson, Pierre Pinson, and Henrik Madsen. On the market impact of wind energy forecasts. *Energy Economics*, 32(2):313–320, 2010.

[Mon17] Douglas C Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

[MMAC16] Raymond H Myers, Douglas C Montgomery, and Christine M Anderson-Cook. *Response surface methodology: process and product optimization using designed experiments*. John Wiley & Sons, 2016.

[PMJScholkopf14] Jonas Peters, Joris M Mooij, Dominik Janzing, and Bernhard Schölkopf. Causal discovery with continuous additive noise models. *Journal of Machine Learning Research*, 2014.

[RR83] Paul R Rosenbaum and Donald B Rubin. The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1):41–55, 1983.

[RNK+19] Jakob Runge, Peer Nowack, Marlene Kretschmer, Seth Flaxman, and Dino Sejdinovic. Detecting and quantifying causal associations in large nonlinear time series datasets. *Science advances*, 5(11):eaau4996, 2019.

[SWNW03] Thomas J Santner, Brian J Williams, William I Notz, and Brain J Williams. *The design and analysis of computer experiments*. Volume 1. Springer, 2003.

[SHHyvarinen+06] Shohei Shimizu, Patrik O Hoyer, Aapo Hyvärinen, Antti Kerminen, and Michael Jordan. A linear non-gaussian acyclic model for causal discovery. *Journal of Machine Learning Research*, 2006.

[Spi01] Peter Spirtes. An anytime algorithm for causal inference. In *International Workshop on Artificial Intelligence and Statistics*, 278–285. PMLR, 2001.

[SGS01] Peter Spirtes, Clark Glymour, and Richard Scheines. *Causation, prediction, and search*. MIT press, 2001.