Contents

3 *Data Collection and Processing*

Data collection is a critical step in the prescriptive analytics process. It involves gathering, cleaning, and organizing data to create a dataset that can be used for analysis and modeling. In this chapter, we will discuss the importance of data collection in prescriptive analytics, the challenges associated with data collection, and best practices for collecting and processing data for prescriptive analytics projects.

Recall again that in the previous chapter, we use a data-driven approach to build a model that solves a specific problem. The model that we build is based on a dataset that we collect and process so that it is ready to load into the model. The dataset was formatted into a tabular format that consists of rows and columns that represent products and their attributes, respectively. See Table 3.1 for an example.

	A	**B**	**C**	**D**
1	product	revenue	cost	production_capacity
2	1	50	30	200
3	2	70	40	150
4	3	90	50	100

data | params

Table 3.1: The Simple Production Planning Problem Dataset (Exercise 2.4), available at `https://docs.google.com/spreadsheets/d/e/2PACX-1vQzIHhg3mZq4eGXraXQZl07kduWMhwrnUqqs_gPT6qH_V1SWI3crMZMllxG6MX1sz3QJCFBjMt9tftr/pub?output=xlsx`).

Note, however, that the values in the dataset can be a summary of the actual raw data. Revenue and cost, for example, can be calculated from the actual sales data aggregated over a period of time at particular store locations. Production capacity can be estimated based on the historical production data, which might vary overtime due to various factors such as machine breakdowns, maintenance schedules, and other operational issues. In this chapter, we will discuss the process of collecting and processing data to create a dataset that can be used for analysis and modeling. It includes gathering data from various sources, cleaning the data to ensure accuracy and reliability, transforming the data into a suitable format, and formatting the data for analysis and modeling.

3.1 *Data Gathering*

The first step in the data collection process is gathering data from various sources. Data can be collected from internal sources, such as databases, spreadsheets, and other data repositories, as well as external sources, such as APIs, web scraping, and third-party data providers. The data collected should be relevant to the problem being addressed and should be of high quality to ensure accurate and reliable results.

3.1.1 *Databases*

Databases are a common source of data for prescriptive analytics projects. Databases can contain a wide range of information, including customer data, product data, sales data, and other relevant information. Databases can be relational databases, such as MySQL, PostgreSQL, and Oracle, or NoSQL databases, such as MongoDB, Cassandra, and Redis. Databases can be accessed using SQL queries or APIs, depending on the type of database being used.

A common challenge when working with databases is extracting and processing data from multiple tables or databases. This can be a time-consuming process, especially if the data is spread across multiple databases or tables. To address this challenge, it is important to have a clear understanding of the data schema and relationships between tables, as well as the ability to write complex SQL queries to extract and process the data.

Exercise 3.1. Download the Sales Analytics SQL dataset and import it into an online SQL editor like SQLiteOnline.com. Write a SQL query that outputs the `product_id`, `unit_price`, `cost`, and `production_capacity` for the top 3 products with the highest unit price.

Solution: The SQL query in figure 3.1 retrieves the product_id, unit_price, cost from the products table and the production_capacity from the productions table. The query joins the products and productions tables on the product_id column and selects the relevant columns. The results are ordered by the unit_price in descending order and limited to the top 3 products. Downloading this query result to a CSV or Excel file can be done using the SQL editor's export feature. Figure 3.2 shows the query result.

```sql
SELECT
    p.product_id AS id,
    p.unit_price AS price,
    p.cost,
    pr.production_capacity AS cap
FROM
    products p
JOIN
    productions pr ON id = pr.id
ORDER BY
    price DESC
LIMIT 3;
```

Figure 3.1: SQL query to retrieve the top 3 products with the highest unit price.

id	price	cost	cap
779	1498	16.28	552
818	1494	43.68	783
855	1491	55.48	500

Figure 3.2: Query result for the top 3 products with the highest unit price.

Note a few basic operations we perform with SQL queries in the solution to exercise 3.1[1]:

- **Data Selection**: The query selects key fields, including product details, unit price, cost, and production capacity.
- **Joins and Relationships**: The JOIN clause links the products and productions tables on the product_id column to combine data from both tables.
- **Sorting**: The ORDER BY clause sorts the results by unit price in descending order. Sorting in ascending order can be done by changing DESC to ASC (which is the default).
- **Limiting Results**: The LIMIT clause restricts the output to the top 3 products with the highest unit price.
- **Aliases**: The AS keyword is used to create aliases for columns, making the output more readable.

[1] The Sales Analytics SQL dataset can be downloaded from https://drive.google.com/file/d/1AfotiR977uDVjjUbNsK_qRYnQNQVVM5d/view?usp=sharing.

Exercise 3.2. Let's do a more complex query to analyze sales performance using the Sales Analytics SQL dataset. Write a SQL query that retrieves and calculates the following information from the dataset:

- product_id, product_name, unit_price, cost from the products table
- production_capacity from the productions table
- customer_id and customer_name from the sales table
- Calculate the profit_per_unit as the difference between the unit_price and cost from the products table
- Calculate the total_quantity_sold as the sum of the quantity_sold from the sales table
- Calculate the total_revenue as the sum of the quantity_sold multiplied by the unit_price from the sales table
- Calculate the total_profit as the sum of the quantity_sold multiplied by the difference between the unit_price and cost from the sales table
- Group the data by product_id, customer_id, product_name, unit_price, cost, production_capacity, and customer_name
- Order the results by total_revenue in descending order

Note that you might need to use JOIN and LEFT JOIN clauses to combine data from multiple tables.

Solution: The SQL query in figure 3.3 retrieves and calculates the specified information from the products, productions, sales, and customers tables. It illustrates how data from multiple related tables can be combined to produce meaningful insights. The query uses SQL functions like SUM and MAX to calculate total units sold, total revenue, total profit, and the date of the last purchase. The JOIN and LEFT JOIN clauses link sales to product details, production capacity, and customer information, ensuring comprehensive data collection. The GROUP BY clause groups data by product and customer, while the ORDER BY clause sorts results by total revenue, making it easier to identify top-performing products.

```sql
SELECT
    p.product_id,
    p.product_name,
    p.unit_price,
    p.cost,
    (p.unit_price - p.cost) AS profit_per_unit,
    pr.production_capacity,
    c.customer_id,
    c.customer_name,
    SUM(s.quantity_sold) AS total_quantity_sold,
    SUM(s.quantity_sold * p.unit_price) AS total_revenue,
    SUM(s.quantity_sold * (p.unit_price - p.cost)) AS total_profit,
    MAX(s.sale_date) AS last_purchase_date
FROM
    sales s
JOIN
    products p ON s.product_id = p.product_id
LEFT JOIN
    productions pr ON p.product_id = pr.product_id
LEFT JOIN
    customers c ON s.customer_id = c.customer_id
GROUP BY
    p.product_id, c.customer_id, p.product_name, p.unit_price, p.cost, pr.
        production_capacity, c.customer_name
ORDER BY
    total_revenue DESC;
```

Figure 3.3: SQL query to analyze sales performance based on the Sales Analytics dataset.

As shown in the exercises, SQL queries can be used to extract and process data from multiple tables to create a dataset for analysis and modeling. SQL queries can be used to combine data from different sources, calculate aggregated metrics, and generate insights that can help organizations make informed decisions. SQL queries are essential for data collection and processing in prescriptive analytics projects, enabling organizations to extract valuable information from their databases and use it to drive business decisions.

3.1.2 *Spreadsheets*

Spreadsheets are another common source of data for prescriptive analytics projects. Spreadsheets can contain a wide range of information, including financial data, sales data, customer data, and other relevant information. Spreadsheets are easy to use and can be accessed using tools such as Microsoft Excel or Google Sheets Spreadsheets can be exported to CSV files or other formats for further analysis and modeling.

A common challenge when working with spreadsheets is handling large datasets that contain multiple sheets or tabs. This can be a time-consuming process, especially if the data is spread across multiple sheets or tabs. To address this challenge, it is important to have a clear understanding of the data structure and relationships between sheets, as well as the ability to merge and process data from multiple sheets.

3.1.3 *Public Datasets*

Public datasets are another valuable source of data for prescriptive analytics projects. Public datasets are freely available datasets that can be used for analysis and modeling. Public datasets can include a wide range of information, such as government data, research data, and other relevant information. Public datasets are available in a variety of formats, including CSV, JSON, and Parquet, and can be easily loaded into Python using libraries such as Pandas and NumPy.

A common way to access public datasets is through online repositories such as Kaggle[2], UCI Machine Learning Repository[3], or HuggingFace Datasets[4]. These repositories contain a wide range of datasets that can be used for analysis and modeling.

[2] `https://www.kaggle.com/`

[3] `https://archive.ics.uci.edu/`

[4] `https://huggingface.co/docs/datasets/en/index`

From these repositories, you can download datasets in various formats, such as CSV or JSON, which can be processed using various libraries. In Python, you can use libraries such as Pandas, NumPy, and Scikit-learn to load, process, and analyze public datasets. Public datasets are a valuable source of data for prescriptive analytics projects, providing access to a wide range of information that can be used to prototype and develop models. Table 3.2 shows and example of how datasets can be represented in a Pandas DataFrame style in Python.

	TransID	**ProductID**	**Qty**	**Date**	**Total**
0	T001	P001	2	2024-03-01	$299.98
1	T002	P003	5	2024-03-02	$149.95
2	T003	P002	3	2024-03-02	$104.97
3	T004	P001	1	2024-03-03	$149.99
4	T005	P004	2	2024-03-03	$159.98

```
DataFrame[... rows × ... columns]
```

Table 3.2: The same data in Pandas DataFrame style

3.1.4 *Web Scraping*

Web scraping is a technique used to extract data from websites. Web scraping can be used to collect a wide range of information, including product prices, customer reviews, news articles, and other relevant data. Web scraping can be done using tools such as BeautifulSoup, Scrapy, and Selenium, which allow you to extract data from websites by parsing HTML and XML documents. More recently, LLM (Large Language Models) have been used to scrape data from websites. An example to extract data from a website is shown in figure 3.4.

```python
import requests
from bs4 import BeautifulSoup
import json

url = 'https://www.its.ac.id/simt/en/publication/'
response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')
```

Figure 3.4: Python code to scrape web content using BeautifulSoup.

The Python code in figure 3.4 demonstrates a practical example of web scraping using the requests library to fetch web content and BeautifulSoup to parse HTML data. Once the web content is fetched and parsed, you can extract specific information from the HTML document, such as publication data, using BeautifulSoup's methods. The extracted data can then be processed and stored in a structured format, such as JSON, for further analysis and modeling. Figure figure 3.5 shows an example of parsing the data from the `soup` object from figure 3.4.

Here's a breakdown of what happens in both of these code chunks. The first code chunk fetches the web content from a URL and parses it using BeautifulSoup. The second code chunk iterates through all tables with the class 'tg' on the page, extracts structured publication data from each table, and organizes the data into a nested list structure. The extracted data is then converted to JSON format for easy storage or further processing. This example demonstrates how web scraping can be used to automatically collect structured data from websites, such as academic publications, for analysis and modeling. However, it is important to note that web scraping should be done ethically and in compliance with the website's terms of service.

```python
all_table_data = []  # List to store data from all tables

for table in soup.find_all('table', {'class': 'tg'}):  # Iterate through all
     tables with class 'tg'
    table_data = []  # List to store data for the current table
    for row in table.find_all('tr')[1:]:
        cells = row.find_all('td')
        if len(cells) == 4:  # Ensure row has 4 cells (No, Penulis, Tahun, Judul
             )
            entry = {
                'No': cells[0].text,
                'Penulis': cells[1].text,
                'Tahun': cells[2].text,
                'Judul': cells[3].text
            }
            table_data.append(entry)
    all_table_data.append(table_data)  # Append the current table's data to the
         main list

# Convert the data to JSON format
json_data = json.dumps(all_table_data, indent=4)
print(json_data)  # Print the JSON data for all tables
```

Figure 3.5: Python code to parse structured data from web content using BeautifulSoup.

3.2 *Data Cleaning*

Data cleaning is an important step in the data collection process. It involves identifying and correcting errors, missing values, and inconsistencies in the data to ensure that the data is accurate and reliable. Data cleaning can involve a wide range of tasks, including removing duplicates, filling missing values, correcting errors, and standardizing data formats. Data cleaning is essential for ensuring that the data is suitable for analysis and modeling.

Suppose that we have a dataset that contains information about customer_id, product_id, quantity_sold, and sale_date and we want to clean the data to ensure that it is accurate and reliable. The data is stored in a CSV file that can be downloaded from the Sales Dataset[5]. Once downloaded, the data can be loaded into a Pandas DataFrame as shown in figure 3.6.

[5] Available at `https://drive.google.com/file/d/1WN21Dgy3oWSxITTMzW69hjUesuInvyvd/view?usp=sharing`.

```python
import pandas as pd

df = pd.read_csv('sales_data.csv')
```

Figure 3.6: Python code to load data from a CSV file into a Pandas DataFrame.

3.2.1 *Handling Missing Values*

Missing values are a common issue in datasets and can occur for a variety of reasons, such as data entry errors, data corruption, or incomplete data. Missing values can affect the accuracy and reliability of the analysis and modeling results, so it is important to handle missing values appropriately. There are several strategies for handling missing values, including removing rows with missing values, imputing missing values, and using predictive models to fill missing values based on other features. Figure 3.7 shows an example of handling missing values in a Pandas DataFrame by filling missing customer_id values with some default value, say, -1, to represent anonymous customers.

```python
print("Number of values missing:" , df.isnull().sum())

default_value = -1
df['customer_id'] = df['customer_id'].fillna(default_value).astype(int)

print("Number of values missing after filling:" , df.isnull().sum())
```

Figure 3.7: Handling missing values in Python Pandas DataFrame.

3.2.2 *Removing Duplicates*

Duplicates are another common issue in datasets and can occur when the same data is recorded multiple times. Duplicates can affect the accuracy and reliability of the analysis and modeling results, so it is important to remove duplicates from the dataset. Duplicates can be identified by comparing rows in the dataset and removing rows that are identical or nearly identical. Duplicates can be removed based on a single column or multiple columns, depending on the criteria used to identify duplicates. Figure 3.8 shows an example of removing duplicates from a Pandas DataFrame, say, from the sale_id column. The optional parameter keep

```python
print("\n=== Removing Duplicates ===")
print("Duplicated sale_ids before:", df['sale_id'].duplicated().sum())

# Keep first occurrence of each sale_id
df = df.drop_duplicates(subset=['sale_id'], keep='first')

print("Duplicated sale_ids after:", df['sale_id'].duplicated().sum())
```

Figure 3.8: Python code to remove duplicates from a Pandas DataFrame based on the sale_id column.

specifies which duplicates to keep. The default value is 'first', which keeps the first occurrence of the duplicate and removes the subsequent occurrences. Other possible values are 'last' and False, which keep the last occurrence of the duplicate and remove all duplicates, respectively.

3.2.3 *Standardizing Data Formats*

Data formats can vary widely in datasets, which can make it difficult to analyze and model the data. Standardizing data formats involves converting data into a consistent format to ensure that the data is suitable for analysis and modeling. Data formats can include date formats, currency formats, and other formats that need to be standardized to ensure that the data is accurate and reliable. Standardizing data formats can involve converting data types, parsing strings, and formatting data according to a specific format. Figure 3.9 shows an example of standardizing the column sale_date to a consistent datetime format in a Pandas DataFrame, while handling multiple input date formats and removing invalid future dates.

```python
from datetime import datetime

print("\n=== Standardizing Dates ===")

def convert_date(date_str):
    try:
        # Try different date formats
        for fmt in ('%Y-%m-%d', '%m/%d/%Y', '%d-%b-%Y'):
            try:
                return pd.to_datetime(date_str, format=fmt)
            except:
                continue
        return pd.NaT
    except:
        return pd.NaT

df['sale_date'] = df['sale_date'].apply(convert_date)

# Remove future dates
current_date = pd.Timestamp('2024-01-01')
df = df[df['sale_date'] <= current_date]
```

Figure 3.9: Python code to standardize the sale_date column to a specific date format in a Pandas DataFrame.

The code defines a convert_date() function that attempts to parse dates in multiple common formats ('\%Y-\%m-\%d', '\%m/\%d/\%Y', and '\%d-\%b-\%Y'). For each date string, it tries each format until one succeeds, returning a pandas Timestamp object. If no format matches or there's an error, it returns pd.NaT (Not a Time) to handle invalid dates gracefully. After converting all dates, the code removes any records with future dates (after January 1, 2024) to ensure data validity. This robust approach handles the common challenges of inconsistent date formats and invalid dates in real-world datasets.

3.2.4 *Correcting Errors*

Errors in datasets can occur for a variety of reasons, such as data entry errors, data corruption, or incomplete data. Errors can affect the accuracy and reliability of the analysis and modeling results, so it is important to correct errors in the dataset. Common errors include negative values in quantity fields, missing values, and outliers that deviate significantly from the normal range. Figure 3.10 shows an example of cleaning quantity data by removing negative values and handling outliers using the Interquartile Range (IQR) method.

```python
print("\n=== Cleaning Quantities ===")
print("Negative quantities:", (df['quantity_sold'] < 0).sum())

# Remove negative quantities
df = df[df['quantity_sold'] >= 0]

# Handle outliers using IQR method
Q1 = df['quantity_sold'].quantile(0.25)
Q3 = df['quantity_sold'].quantile(0.75)
IQR = Q3 - Q1
outlier_mask = (df['quantity_sold'] > Q3 + 1.5 * IQR)
print("Outliers detected:", outlier_mask.sum())

# Cap outliers at 99th percentile
cap_value = df['quantity_sold'].quantile(0.99)
df.loc[outlier_mask, 'quantity_sold'] = cap_value
```

Figure 3.10: Python code to clean quantity data by removing negative values and handling outliers.

The code first identifies and removes any negative quantities, which are likely data entry errors since quantities sold should always be non-negative. Then, it handles outliers using the IQR method, which is a robust statistical technique for detecting extreme values. The IQR is calculated as the difference between the

75th percentile (Q3) and 25th percentile (Q1). Values that are more than 1.5 times the IQR above Q3 are considered outliers. Rather than removing these outliers completely, the code caps them at the 99th percentile to preserve the data while mitigating their impact on analysis. This approach maintains data integrity while ensuring that extreme values don't unduly influence statistical calculations and modeling results.

3.3 *Data Transformation*

Data transformation is an important step in the data collection process. It involves converting raw data into a format that is suitable for analysis and modeling. Data transformation can involve a wide range of tasks, including aggregating data, normalizing data, and encoding categorical variables. Data transformation is essential for ensuring that the data is accurate and reliable and can be used for analysis and modeling.

3.3.1 *Aggregating Data*

We have seen an example of data aggregation in the previous exercise where we calculated the total revenue, total profit, and total quantity sold for each product and customer. Data aggregation involves combining data from multiple rows into a single row to create summary statistics or metrics. Data aggregation can be done using functions such as SUM, COUNT, MAX, and MIN to calculate aggregated metrics based on specific criteria. Data aggregation is useful for summarizing data and creating meaningful insights that can be used for analysis and modeling. Figure 3.11 shows an example of aggregating the quantity_sold as a sum for each product_id in a Pandas DataFrame.

```
df_agg = df.groupby('product_id')['quantity_sold'].sum().reset_index()
```

Figure 3.11: Python code to aggregate the quantity_sold as a sum for each product_id in a Pandas DataFrame.

The groupby() function groups the data by the product_id column, and the sum() function calculates the sum of the quantity_sold column for each group. The reset_index() function resets the index of the resulting DataFrame to create a new DataFrame with the aggregated data. Figure 3.12 shows more advanced examples of data aggregation, including daily sales summaries with metrics like total sales and unique customers per day, product performance tracking quantities sold, and customer purchase patterns analyzing buying behavior. The code uses the agg() function to calculate multiple aggregations like sum, mean, count, and number of unique values for different columns simultaneously.

```python
print("=== Aggregating Data ===")

# Daily sales summary
daily_sales = df.groupby('sale_date').agg({
    'sale_id': 'count',
    'quantity_sold': ['sum', 'mean', 'std'],
    'customer_id': 'nunique'
}).round(2)

# Product performance
product_performance = df.groupby('product_id').agg({
    'quantity_sold': ['sum', 'mean'],
    'sale_id': 'count'
}).round(2)

# Customer purchase patterns
customer_patterns = df[df['customer_id'] != -1].groupby('customer_id').agg({
    'sale_id': 'count',
    'quantity_sold': ['sum', 'mean'],
    'product_id': 'nunique'
}).round(2)
```

Figure 3.12: Python code to aggregate data in multiple ways: daily sales summaries showing total sales and unique customers per day, product performance metrics tracking quantities sold, and customer purchase patterns analyzing buying behavior.

3.3.2 *Normalizing Data*

Another example of data transformation is normalizing data. Normalizing data involves scaling the features in the dataset to ensure that the data is standardized and comparable. There are several common normalization techniques:

- **Min-Max Scaling**: Scales features to a fixed range, typically 0 to 1, by subtracting the minimum value and dividing by the range
- **Z-score Standardization**: Transforms data to have a mean of 0 and standard deviation of 1 by subtracting the mean and dividing by the standard deviation

Normalizing data is important for ensuring that features with different scales and units can be compared fairly. This is especially crucial for machine learning algorithms that are sensitive to the scale of input features. Figure 3.13 shows an example of applying both min-max scaling and z-score standardization to the quantity_sold column.

```python
# Import scaler from sklearn
from sklearn.preprocessing import MinMaxScaler

# Normalize quantity_sold using Min-Max scaling
scaler = MinMaxScaler()
df['quantity_sold_normalized'] = scaler.fit_transform(df[['quantity_sold']])

# Create z-scores for quantity_sold
df['quantity_sold_zscore'] = (df['quantity_sold'] - df['quantity_sold'].mean())
     / df['quantity_sold'].std()
```

Figure 3.13: Python code demonstrating two normalization techniques: Min-Max scaling using sklearn and z-score standardization using statistical formulas.

The code uses scikit-learn's MinMaxScaler to perform min-max scaling, which transforms the data to the [0,1] range. For z-score standardization, it applies the standard formula directly using Pandas operations. After normalization, the features become more suitable for analysis and modeling, as they are on comparable scales regardless of their original units.

3.3.3 *Encoding Categorical Variables*

Another example of data transformation is encoding categorical variables. Categorical variables are variables that represent categories or groups. Oftentimes, categorical variables need to be encoded into numerical values to be used in analysis and modeling, as most machine learning algorithms require numerical input. In this example, we'll demonstrate how to encode temporal features like days of the week and months using one-hot encoding. Figure 3.14 shows an example of encoding these categorical variables using Pandas.

The code first extracts temporal components (year, month, day, day of week) from the sale dates. Then, it uses Pandas' get_dummies() function to perform one-hot encoding on the day of week and month values. One-hot encoding creates binary columns for each unique category - for example, each day of the week (0-6) gets its own column where 1 indicates that day and 0 indicates other days. The encoded features are concatenated with the original DataFrame, and the original

```python
print("\n=== Engineering Features ===")

# Extract date components
df['sale_year'] = df['sale_date'].dt.year
df['sale_month'] = df['sale_date'].dt.month
df['sale_day'] = df['sale_date'].dt.day
df['sale_dayofweek'] = df['sale_date'].dt.dayofweek

print("\n=== Encoding Categorical Variables ===")

# 1. One-hot encode day of week (0-6)
day_encoded = pd.get_dummies(df['sale_dayofweek'], prefix='day', dtype=int)

# 2. One-hot encode month
month_encoded = pd.get_dummies(df['sale_month'], prefix='month', dtype=int)

# Combine all encoded features
df = pd.concat([
    df,
    day_encoded,
    month_encoded
], axis=1)

# Remove original categorical columns if desired
columns_to_drop = ['sale_dayofweek', 'sale_month']
df = df.drop(columns=columns_to_drop)

print("Added encoded columns:")
print("- Days of week:", list(day_encoded.columns))
print("- Months:", list(month_encoded.columns))

def show_encoding_example(df):
    print("\n=== Example of Encoded Data ===")
    encoded_columns = (
        list(df.filter(like='day_').columns) +
        list(df.filter(like='month_').columns)
    )
    print("\nSample of encoded data (first 5 rows):")
    print(df[encoded_columns].head())

# Call the function to print the example
show_encoding_example(df)
```

Figure 3.14: Python code demonstrating feature engineering and one-hot encoding of temporal categorical variables (days of week and months) using Pandas.

categorical columns are dropped. The code includes helper functions to display the newly created encoded columns and show example rows of the transformed data. This type of encoding is particularly useful for machine learning models that need numerical inputs and can benefit from capturing cyclical patterns in temporal data.

Table 3.3 shows an example of the one-hot encoded data, where each row represents a sale transaction and the columns show the encoded values for days and months. The first section shows the days of the week encoded as binary values (0 or 1), with columns for Sunday through Saturday. The second section shows the months encoded similarly, with columns for January through December. A value of 1 indicates that the sale occurred on that specific day or month, while 0 indicates it did not. For example, looking at index 0, we can see this sale occurred on a Saturday (1 in the Sat column) in November (1 in the Nov column). Similarly, the sale at index 1 occurred on a Monday in July, as indicated by the 1s in those respective columns. This binary representation transforms categorical temporal data into a numerical format that machine learning models can process effectively.

3.3.4 *Dimensionality Reduction*

Another more complex example of data transformation is dimensionality reduction. Dimensionality reduction is a technique used to reduce the number of features in a dataset while preserving the most important information. Dimensionality reduction can be done using techniques such as principal component analysis (PCA), t-distributed stochastic neighbor embedding (t-SNE), and autoencoders. Dimensionality reduction is useful for reducing the computational complexity of the analysis and modeling process and can help improve the accuracy and reliability of the results.

The code demonstrates the application of PCA using scikit-learn's implementation. First, we select numerical features that have been previously normalized or standardized, as PCA is sensitive to the scale of input features. The `PCA` class is initialized with `n\_components=3`, meaning we want to reduce our data to three principal components. The `fit\_transform()` method both fits the PCA model to our data and transforms it into the new lower-dimensional space. The transformed components are then added back to the DataFrame as new columns. The explained variance ratio shows how much of the original data's variance is preserved in

	Days of Week						
	Sun	Mon	Tue	Wed	Thu	Fri	Sat
0	0	0	0	0	0	0	1
1	0	1	0	0	0	0	0
2	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0
4	0	1	0	0	0	0	0

	Months											
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	1	0
3	0	0	1	0	0	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0	0	0	0

```
DataFrame[5 rows × 19 columns]
```

Table 3.3: Example of one-hot encoded temporal data

```python
from sklearn.decomposition import PCA

print("\n=== Dimensionality Reduction ===")

# Select numerical features for PCA
numerical_features = ['quantity_sold_normalized', 'quantity_sold_zscore',
                      'sale_year', 'sale_day']

# Apply PCA
pca = PCA(n_components=3)
pca_result = pca.fit_transform(df[numerical_features])

# Add PCA components to dataframe
df['pca_1'] = pca_result[:, 0]
df['pca_2'] = pca_result[:, 1]
df['pca_3'] = pca_result[:, 2]

print("Explained variance ratio:", pca.explained_variance_ratio_)
```

Figure 3.15: Python code demonstrating dimensionality reduction using Principal Component Analysis (PCA) on numerical features.

each principal component, helping us understand the information retention in the dimensionality reduction process.

3.4 *Data Formatting*

Data formatting is an important step in the data collection process. It involves formatting the data into a specific structure that is suitable for analysis and modeling. We have seen a useful data format in the previous chapter, which is the tabular format. This format is widely used in data analysis and modeling and consists of rows and columns that represent observations and features, respectively.

We have also seen a dictionary format, which is used to represent data in a key-value pair format. The dictionary format is useful for storing data that is structured as key-value pairs, such as customer information, product information, and other relevant data, where each key corresponds to a specific value.

Another common data format is the time series format, which is used to represent data that changes over time. Time series data consists of a sequence of data points that are indexed by time. Time series data can be used to analyze and model trends, patterns, and anomalies in the data. An example of time series data is sales transactions, where each data point represents the quantity sold logged at a specific time.

3.5 *Data-driven Modeling*

A model that is built based on data is called a data-driven model. We have seen a modular approach to building data-driven models in the previous chapter, in which the parameters of the model are loaded from the data. This approach allows us to build models that are flexible and adaptable to different datasets.

In your project, you will be building a data-driven model to solve a specific problem. The model that you build will be based on the data that you collect and process, and will be used both for verification and validation, as well as for solving the problem at hand. During verification, you will use a small-scale dataset to verify that the model is working as expected. After the model is verified, you will use a larger dataset and run the model on it to obtain the optimal solution.

3.6 Summary

In this chapter, we discussed the process of collecting and processing data to create a dataset that can be used for analysis and modeling. We covered the steps involved in data collection, including gathering data from various sources, cleaning the data to ensure accuracy and reliability, transforming the data into a suitable format, and formatting the data for analysis and modeling. We also discussed the importance of data cleaning, data transformation, and data formatting in the data collection process, and provided examples of how these steps can be implemented using Python and Pandas. We also discussed the concept of data-driven modeling and how it can be used to build models that are based on data. In the next chapter, we will discuss the process of verifying and validating the data-driven model to ensure that it is accurate and reliable for solving the problem at hand.

3.7 End of Chapter Exercises

Exercise 3.3. Use the Sales Analytics SQL dataset. Identify the top 5 products by total quantity sold. Write a SQL query to retrieve the product_name and the total quantity sold for each product, and order the results by the total quantity sold in descending order.

Exercise 3.4. Use the Sales Analytics SQL dataset. Calculate the total sales for each month of the current year. Write a SQL query to retrieve the month, the number of sales, and the total items sold for each month, and order the results by the month.

Exercise 3.5. Use the Sales Analytics SQL dataset. Identify customers who have made more than 3 purchases. Write a SQL query to retrieve the customer_name and the number of purchases for each customer, and order the results by the number of purchases in descending order.

Exercise 3.6. Use the Sales Analytics SQL dataset. List the most recent purchase for each customer. Write a SQL query to retrieve the customer_name, the last purchase date, and the last product bought for each customer.

Exercise 3.7. Use the Sales Analytics SQL dataset. Compare current sales with production capacity. Write a SQL query to retrieve the product_name, the monthly production capacity, and the monthly sales for each product, and order the results by the product name.

Exercise 3.8. Use the Sales Analytics SQL dataset. Calculate simple profit margins for each product. Write a SQL query to retrieve the product_name, the unit_price, the cost, the profit per unit, and the profit margin percentage for each product, and order the results by the profit margin percentage in descending order.

Exercise 3.9. Try the following variations on your own:

1. Modify the top selling products query to show sales for a specific month.
2. Find all products that haven't had any sales in the last month.
3. List customers who have bought a specific product.
4. Calculate the average order size for each product.

End-of-Chapter Exercises Solutions

```sql
SELECT
    p.product_name,
    SUM(s.quantity_sold) as total_quantity
FROM products p
JOIN sales s ON p.product_id = s.product_id
GROUP BY p.product_name
ORDER BY total_quantity DESC
LIMIT 5;
```

Figure 3.16: SQL query to find the top 5 products by total quantity sold.

```sql
SELECT
    strftime('%Y-%m', sale_date) as month,
    COUNT(*) as number_of_sales,
    SUM(quantity_sold) as total_items_sold
FROM sales
WHERE sale_date >= date('now', 'start of year')
GROUP BY month
ORDER BY month;
```

Figure 3.17: SQL query to calculate total sales for each month of the current year.

```sql
SELECT
    c.customer_name,
    COUNT(s.sale_id) as purchase_count
FROM customers c
JOIN sales s ON c.customer_id = s.customer_id
GROUP BY c.customer_name
HAVING purchase_count > 3
ORDER BY purchase_count DESC;
```

Figure 3.18: SQL query to identify customers who have made more than 3 purchases.

```sql
SELECT
    c.customer_name,
    MAX(s.sale_date) as last_purchase_date,
    p.product_name as last_product_bought
FROM customers c
JOIN sales s ON c.customer_id = s.customer_id
JOIN products p ON s.product_id = p.product_id
GROUP BY c.customer_name;
```

Figure 3.19: SQL query to list the most recent purchase for each customer.

```sql
SELECT
    p.product_name,
    pr.production_capacity as monthly_capacity,
    SUM(s.quantity_sold) as monthly_sales
FROM products p
JOIN productions pr ON p.product_id = pr.product_id
JOIN sales s ON p.product_id = s.product_id
WHERE s.sale_date >= date('now', 'start of month')
GROUP BY p.product_name;
```

Figure 3.20: SQL query to compare current sales with production capacity.

```sql
SELECT
    product_name,
    unit_price,
    cost,
    (unit_price - cost) as profit_per_unit,
    ROUND((unit_price - cost) * 100.0 / unit_price, 2)
    as profit_margin_percentage
FROM products
ORDER BY profit_margin_percentage DESC;
```

Figure 3.21: SQL query to calculate simple profit margins for each product.