MACHINE LEARNING MASTERY

THE BEGINNER'S GUIDE TO Data Science

A Journey from Data to Insight with Statistical Techniques

Vinod Chugani

Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

Credits

Founder: Jason Brownlee
Lead Editor: Adrian Tam
Author: Vinod Chugani
Technical Reviewers: Yoyo Chan and Kanwal Mehreen

Copyright

Edition: v1.00

Contents

Preface

Data science involves applying scientific techniques to extract insights from data, essentially telling a story through data analysis. It involves utilizing sophisticated methods like machine learning models to identify patterns or validate hypotheses. However, cultivating a data science mindset is the most fundamental and crucial aspect.

This mindset does not rely on complex models; even basic statistical techniques are sufficient to illustrate a data science workflow. This book operates on this premise: without delving into advanced machine learning, it teaches how to approach data and substantiate hypotheses. The essence of data science lies in constructing a compelling *narrative* supported by empirical evidence.

In this guide, you will be led through each step, from data manipulation in Python to employing fundamental statistical functions in NumPy and SciPy to draw insights from the data. Whether you are new to data science or seeking to enhance your skills, this guide requires minimal prerequisites yet instills the mindset of an experienced data scientist.

Introduction

Welcome to The Beginner's Guide to Data Science. This book is a primer to data science but does not involve any advanced machine learning models. The advanced models are deliberately avoided because you should focus on what data science is about——telling stories.

Data science can be difficult because there are unlimited tools you can use as long as they can help you tell a story. As a beginner, you can get lost because you see a lot of statistical tests, models, equations, and plots being used, and you may feel like you are jumping around. However, you will find data science is easier to grasp once you focus on the objective rather than the tool.

How to Learn Data Science

Watch how a data scientist works. What do you need to get the job done?

- ▷ You must know where to find and collect the data.
- ▷ You must be very proficient in programming.
- ▷ You must know a lot about probability and statistics.
- ▷ You must be an expert in machine learning models.
- ▷ You must be an artist who can create beautiful charts.
- ▷ You need to be a very good presenter.

None of these are wrong. However, you do not need to spend years mastering everything before you can work on a data science project. Because data science is about telling stories, you can quickly ramp up your data science journey by asking two questions: What story do you want to tell, and how?

This book is created to walk you through a data science project but using only the bare minimum of mathematics and programming. No sophisticated machine learning models are involved. Building fancy charts and web pages is also not the focus. Rather, you will start with a data set you can readily download and ask various questions about it. As a data scientist, you will back your statement with data derived from some statistical model by juggling the dataset with a few lines of Python code.

Book Organization

This book is in two parts. The first part is about data wrangling, particularly with the pandas library. This is useful because you often need to process the dataset to create derived data or filter for useful data. This part includes the following chapters:

1. Revealing the Invisible: Visualizing Missing Values in Ames Housing
2. Exploring Dictionaries, Classifying Variables, and Imputing Data
3. Beyond SQL: Transforming Real Estate Data into Actionable Insights with Pandas
4. Harmonizing Data: A Symphony of Segmenting, Concatenating, Pivoting, and Merging

These chapters guide you using Python and the pandas library to manipulate a dataset. This is a useful and important skill because you should manipulate the entire dataset rather than processing each number from the dataset. With pandas, you can filter the data or find the average with just one line of code.

The second part of the book assumes you can process the data readily. Then, you will be shown how to obtain information from the raw data to make a statement.

This part includes the following chapters:

4. Decoding Data: An Introduction to Descriptive Statistics
5. From Data to Map: Visualizing Ames House Prices with Python
6. Feature Relationships 101: Lessons from the Ames Housing Data
7. Mastering Pair Plots for Visualization and Hypothesis Creation
8. Inferential Insights: How Confidence Intervals Illuminate the Ames Real Estate Market
9. Testing Assumptions in Real Estate: A Dive into Hypothesis Testing
10. Garage or Not? Housing Insights Through the Chi-Squared Test
11. Leveraging ANOVA and Kruskal-Wallis Tests to Analyze the Impact of the Great Recession on Housing Prices
12. Spotting the Exception: Classical Methods for Outlier Detection in Data Science
13. Skewness Be Gone: Transformative Tricks for Data Scientists
14. Finding Value with Data: The Cohesive Force Behind Luxury Real Estate Decisions

All these chapters are based on the same dataset, but each asks a different question. This is not a simple question like what is the average price of a house, but one like whether having a garage makes a house more expensive. To answer such a question, you must find the right mathematical tool, apply the appropriate Python function, and interpret the result correctly.

To tell a story about the data, you focus on the objective, and there can be multiple ways to achieve that. Therefore, you should consider the methods and models used in each chapter as examples rather than the only golden rule for the task. It may not be the best way for a task either, but surely, it is a simple solution so you can learn the *thought process* easily.

Appendix B outlines the tools you might find useful. In Appendix C you will find a summary of how you should apply the thought process to a data science project.

Requirements for This Book

This book focuses heavily on statistical models rather than more advanced machine learning ones. Even so, you do not need to be an expert in statistics to enjoy it since most statistical concepts are well explained when they are introduced. However, if you are interested in learning more, you may find the other book by Machine Learning Mastery, *Statistical Methods for Machine Learning*[1] useful.

The code in this book is in Python. It is the most popular language in this domain, and you can easily find help online. It does not mean you need to be a Python expert. All you need to know is to install and setup Python with the NumPy and SciPy libraries. The examples assume that you have a Python environment available. This may be on your workstation or laptop, it may be in a VM or a Docker instance that you run, or it may be a server instance that you can configure in the cloud. Please find Appendix A if you need instructions to setup your Python environment.

The most important requirement of this book is a good mindset. There is a lot to cover in this book. Each topic can be drilled down and expanded into a much larger context. However, remember that you should focus on the high-level goal of the data instead of all the details in the models and code. In many cases, you will see that a function from SciPy is used, and there are many arguments you can specify for that function. However, the example only covered a minimum variation to complete the job. You should finish the story in each chapter first and go back to change the code to try a different idea afterward. This would be the fastest way to learn.

Your Outcomes from Reading This Book

This book will lead you from being a programmer with little to no data science knowledge to a developer with sufficient knowledge to carry out basic workflow in data science.

Specifically, you will know:

- How to develop and evaluate an idea about a dataset.
- The basic but commonly used statistical models in data science.
- How to present data in visualizations.
- How to deliver your argument quantitatively.

This book is not a textbook. You are provided with the theoretical background at a bare minimum. This book, however, is a cookbook or playbook such that there are step-by-step instructions you can follow to finish a project.

You do not need to read the chapters in their order. While the chapters are ordered to help you go from the beginning to the end of a data science journey, you can skip to chapters

[1] https://machinelearningmastery.com/statistics_for_machine_learning/

as your needs or interests motivate you. To get the most from this book, you should also attempt to improve the results, try a different function or model, apply the method to a similar but different problem, and so on. You are welcome to share your findings with us at jason@MachineLearningMastery.com.

Next

Let's dive in. Next up is Part I, where you will take a tour of the pandas library in Python to wrangle with data, which is an essential skill you will be using throughout this book.

I Wrangling with Data

Revealing the Invisible: Visualizing Missing Values in Ames Housing

1

The digital age has ushered in an era where data-driven decision-making is pivotal in various domains, real estate being a prime example. Comprehensive datasets, like the *Ames housing dataset*, offer a treasure trove for data enthusiasts. Through meticulous exploration and analysis of such datasets, one can uncover patterns, gain insights, and make informed decisions.

Starting from this chapter, you will embark on a captivating journey through the intricate lanes of Ames properties, focusing primarily on data science techniques.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ The Ames Properties Dataset
- ▷ Loading & Sizing Up the Dataset
- ▷ Uncovering & Visualizing Missing Values

1.1 The Ames Properties Dataset

Every dataset has a story to tell, and understanding its background can offer invaluable context. While the Ames Housing Dataset is widely known in academic circles, the dataset we're analyzing today, `Ames.csv`[1], which is a more comprehensive collection of property details from Ames.

Dr. Dean De Cock, a dedicated academician, recognized the need for a new, robust dataset in the domain of real estate. He meticulously compiled the *Ames Housing Dataset*, which has since become a cornerstone for budding data scientists and researchers. This dataset stands out due to its comprehensive details, capturing myriad facets of real estate properties. It has been a foundation for numerous predictive modeling exercises and offers a rich landscape for exploratory data analysis.

[1]The original dataset was in a tab-separated value format. The CSV version is available at https://raw.githubusercontent.com/Padre-Media/dataset/main/Ames.csv

The Ames Housing Dataset was envisioned as a modern alternative to the older *Boston Housing Dataset*. Covering residential sales in Ames, Iowa between 2006 and 2010, it presents a diverse array of variables, setting the stage for advanced regression techniques.

This time frame is particularly significant in U.S. history. The period leading up to 2007–2008 saw the dramatic inflation of housing prices, fueled by speculative frenzy and subprime mortgages. This culminated in the devastating collapse of the housing bubble in late 2007, an event vividly captured in narratives like "The Big Short." The aftermath of this collapse rippled across the nation, leading to the Great Recession. Housing prices plummeted, foreclosures skyrocketed, and many Americans found themselves underwater on their mortgages. The Ames dataset provides a glimpse into this turbulent period, capturing property sales during national economic upheaval.

1.2 Loading & Sizing Up the Dataset

For those who are venturing into the realm of data science, having the right tools in your arsenal is paramount. If you require some help to setup your Python environment, the guide is in Appendix A.

Dataset Dimensions. Before diving into intricate analyses, it's essential to familiarize yourself with the basic structure and data types of the dataset. This step provides a roadmap for subsequent exploration and ensures you tailor your analyses based on the nature of data. With the Python environment in place, let's load and gauge the dataset in terms of rows (individual properties) and columns (attributes of these properties).

```
# Load the Ames dataset
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Dataset shape
print(Ames.shape)

rows, columns = Ames.shape
print(f"The dataset comprises {rows} properties described across {columns} attributes.")
```

Listing 1.1: Loading the dataset

This prints:

```
(2579, 85)
The dataset comprises 2579 properties described across 85 attributes.
```

Output 1.1: Shape of the loaded DataFrame

Data Types. Recognizing the data type of each attribute helps shape our analysis approach. Numerical attributes might be summarized using measures like mean or median, while mode (the most frequently seen value) is apt for categorical attributes.

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Determine the data type for each feature
data_types = Ames.dtypes

# Tally the total by data type
type_counts = data_types.value_counts()
print(type_counts)
```

Listing 1.2: Investigating the data type

This shows:

```
object     44
int64      27
float64    14
dtype: int64
```

Output 1.2: Count of attribute data types

The Data Dictionary. Each dataset usually comes with a data dictionary to describe each of its features. It specifies the meaning, possible values, and even the logic behind its collection. Whether you're deciphering the meaning behind an unfamiliar feature or discerning the significance of particular values, the data dictionary serves as a comprehensive guide. It bridges the gap between raw data and actionable insights, ensuring that the analyses and decisions are well-informed.

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Determine the data type for each feature
data_types = Ames.dtypes

# View a few datatypes from the dataset (first and last 5 features)
print(data_types)
```

Listing 1.3: Investigating the data type

This shows, for example, Ground Living Area and Sale Price are numerical (int64) data types, while Sale Condition (object) is a categorical data type:

```
PID               int64
GrLivArea         int64
SalePrice         int64
MSSubClass        int64
MSZoning         object
                 ...
SaleCondition    object
GeoRefNo        float64
Prop_Addr        object
Latitude        float64
```

```
Longitude        float64
Length: 85, dtype: object
```

Output 1.3: Data type of each attribute column

and you can match that with the data dictionary, an excerpt as follows:

```
Sale Condition (Nominal): Condition of sale

     Normal    Normal Sale
     Abnorml    Abnormal Sale -  trade, foreclosure, short sale
     AdjLand    Adjoining Land Purchase
     Alloca    Allocation - two linked properties with separate deeds, typically condo
     ↳  with a garage unit
     Family    Sale between family members
     Partial    Home was not completed when last assessed (associated with New Homes)

SalePrice (Continuous): Sale price $$
```

Output 1.4: Excerpt from the data dictionary showing the "SaleCondition" has string values and "SalePrice" is numerical

1.3 Uncovering & Visualizing Missing Values

Real-world datasets seldom arrive perfectly curated, often presenting analysts with the challenge of missing values. These gaps in data can arise due to various reasons, such as errors in data collection, system limitations, or the absence of information. Addressing missing values is not merely a technical necessity but a critical step that significantly impacts the integrity and reliability of subsequent analyses.

Understanding the patterns of missing values is essential for informed data analysis. This insight guides the selection of appropriate imputation methods, which fill in missing data based on available information, thereby influencing the accuracy and interpretability of results. Additionally, assessing missing value patterns informs decisions on feature selection; features with extensive missing data may be excluded to enhance model performance and focus on more reliable information. In essence, grasping the patterns of missing values ensures robust and trustworthy data analyses, guiding imputation strategies and optimizing feature inclusion for more accurate insights.

NaN or None? In pandas, the `isnull()` function is used to detect missing values in a DataFrame or Series. Specifically, it identifies the following types of missing data:

- ▷ `np.nan` (Not a Number), often used to denote missing numerical data
- ▷ `None`, which is a built-in object in Python to denote the absence of a value or a null value

Both nan and NaN are just different ways to refer to `np.nan` in NumPy, and `isnull()` will identify them as missing values. Here is a quick example.

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

# Create a DataFrame with various types of missing values
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4, 5],
    'B': ['a', 'b', None, 'd', 'e'],
    'C': [np.nan, np.nan, np.nan, np.nan, np.nan],
    'D': [1, 2, 3, 4, 5]
})

# Use isnull() to identify missing values
missing_data = df.isnull().sum()

print(df)
print()
print(missing_data)
```

Listing 1.4: Showing NaN values

This prints:

```
     A     B   C  D
0  1.0     a NaN  1
1  2.0     b NaN  2
2  NaN  None NaN  3
3  4.0     d NaN  4
4  5.0     e NaN  5

A    1
B    1
C    5
D    0
dtype: int64
```

Output 1.5: A DataFrame with NaN values

Visualizing Missing Values. When it comes to visualizing missing data, tools like `missingno`, `matplotlib`, and `seaborn` come in handy. By sorting the features based on the percentage of missing values and placing them into a DataFrame, you can easily rank the features most affected by missing data.

```python
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Calculating the percentage of missing values for each column
missing_data = Ames.isnull().sum()
missing_percentage = (missing_data / len(Ames)) * 100

# Combining the counts and percentages into a DataFrame for better visualization
missing_info = pd.DataFrame({'Missing Values': missing_data,
                             'Percentage': missing_percentage})
```

```
# Sorting the DataFrame by the percentage of missing values in descending order
missing_info = missing_info.sort_values(by='Percentage', ascending=False)

# Display columns with missing values
print(missing_info[missing_info['Missing Values'] > 0])
```

Listing 1.5: Calculating exposure of missing values from a DataFrame

This shows:

```
              Missing Values  Percentage
PoolQC                  2570   99.651028
MiscFeature             2482   96.238852
Alley                   2411   93.485847
Fence                   2054   79.643273
FireplaceQu             1241   48.119426
LotFrontage              462   17.913920
GarageCond               129    5.001939
GarageQual               129    5.001939
GarageFinish             129    5.001939
GarageYrBlt              129    5.001939
GarageType               127    4.924389
Longitude                 97    3.761148
Latitude                  97    3.761148
BsmtExposure              71    2.753005
BsmtFinType2              70    2.714230
BsmtFinType1              69    2.675456
BsmtQual                  69    2.675456
BsmtCond                  69    2.675456
GeoRefNo                  20    0.775494
Prop_Addr                 20    0.775494
MasVnrArea                14    0.542846
MasVnrType                14    0.542846
BsmtFullBath               2    0.077549
BsmtHalfBath               2    0.077549
GarageArea                 1    0.038775
BsmtFinSF1                 1    0.038775
Electrical                 1    0.038775
TotalBsmtSF                1    0.038775
BsmtUnfSF                  1    0.038775
BsmtFinSF2                 1    0.038775
GarageCars                 1    0.038775
```

Output 1.6: Count of missing values in each attribute

The `missingno` package facilitates a swift, graphical representation of missing data. The white lines or gaps in the visualization denote missing values. However, it will only accommodate up to 50 labeled variables. Past that range, labels begin to overlap or become unreadable, and by default large displays omit them.

```python
import pandas as pd
import missingno as msno
import matplotlib.pyplot as plt

Ames = pd.read_csv('Ames.csv')
msno.matrix(Ames, sparkline=False, fontsize=20)
plt.show()
```

Listing 1.6: Illustrating missing values using the missingno package

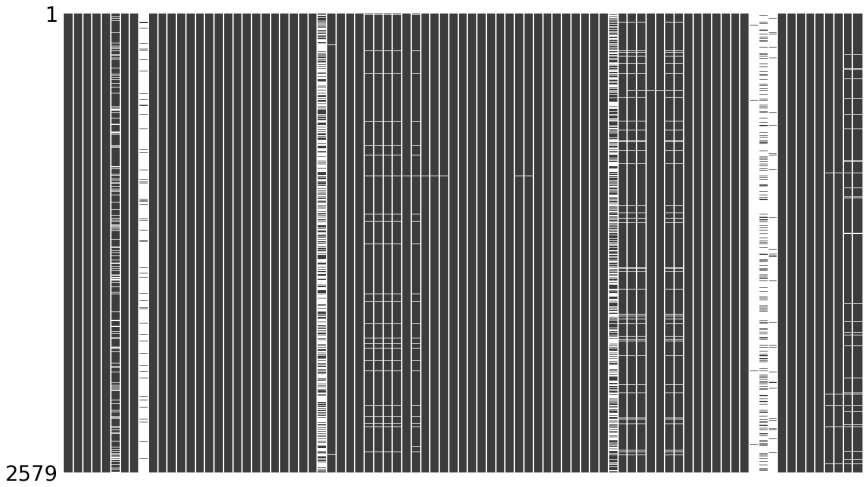

Figure 1.1: Visual representation of missing values using `missingno.matrix()`.

You can get a bar chart using `msno.bar` to show the top 15 features with *least* count of non-missing values, i.e., those with the most missing values.

```python
import pandas as pd
import missingno as msno
import matplotlib.pyplot as plt
Ames = pd.read_csv('Ames.csv')

# Calculating the percentage of missing values for each column
missing_data = Ames.isnull().sum()
missing_percentage = (missing_data / len(Ames)) * 100

# Combining the counts and percentages into a DataFrame for better visualization
missing_info = pd.DataFrame({'Missing Values': missing_data,
                             'Percentage': missing_percentage})

# Sort the DataFrame columns by the percentage of missing values
sorted_df = Ames[missing_info.sort_values(by='Percentage', ascending=False).index]

# Select the top 15 columns with the most missing values
top_15_missing = sorted_df.iloc[:, :15]
```

```
#Visual with missingno
msno.bar(top_15_missing)
plt.show()
```

Listing 1.7: Showing the number of missing values in a bar chart

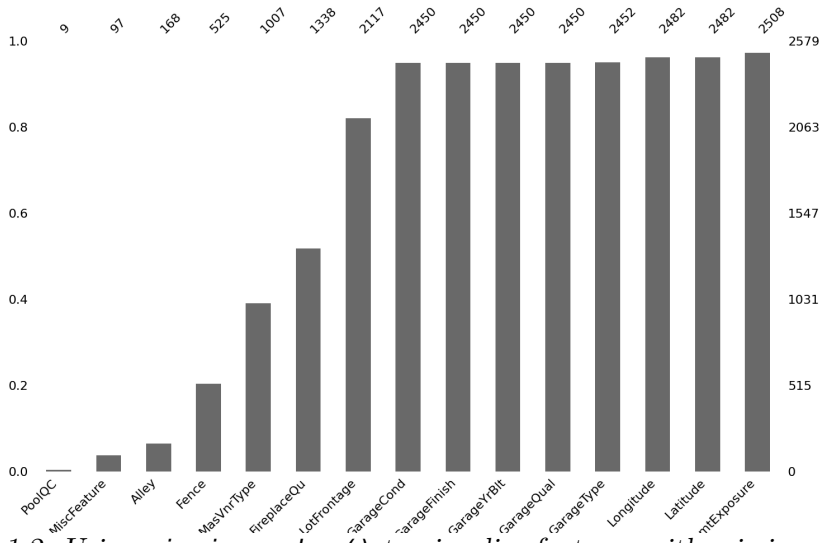

Figure 1.2: Using `missingno.bar()` to visualize features with missing data.

The illustration above denotes that Pool Quality, Miscellaneous Feature, and the type of Alley access to the property are the three features with the highest number of missing values since their bar is the shortest.

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

Ames = pd.read_csv('Ames.csv')
missing_data = Ames.isnull().sum()
missing_percentage = (missing_data / len(Ames)) * 100
missing_info = pd.DataFrame({'Missing Values': missing_data,
                             'Percentage': missing_percentage})

# Filter to show only the top 15 columns with the most missing values
top_15_missing_info = missing_info.nlargest(15, 'Percentage')

# Create the horizontal bar plot using seaborn
plt.figure(figsize=(12, 8))
sns.barplot(x='Percentage', y=top_15_missing_info.index, hue=top_15_missing_info.index,
            data=top_15_missing_info, orient='h')
plt.title('Top 15 Features with Missing Percentages', fontsize=20)
plt.xlabel('Percentage of Missing Values', fontsize=16)
plt.ylabel('Features', fontsize=16)
#plt.yticks(fontsize=11)
plt.show()
```

Listing 1.8: Showing top columns with the highest percentage of missing values

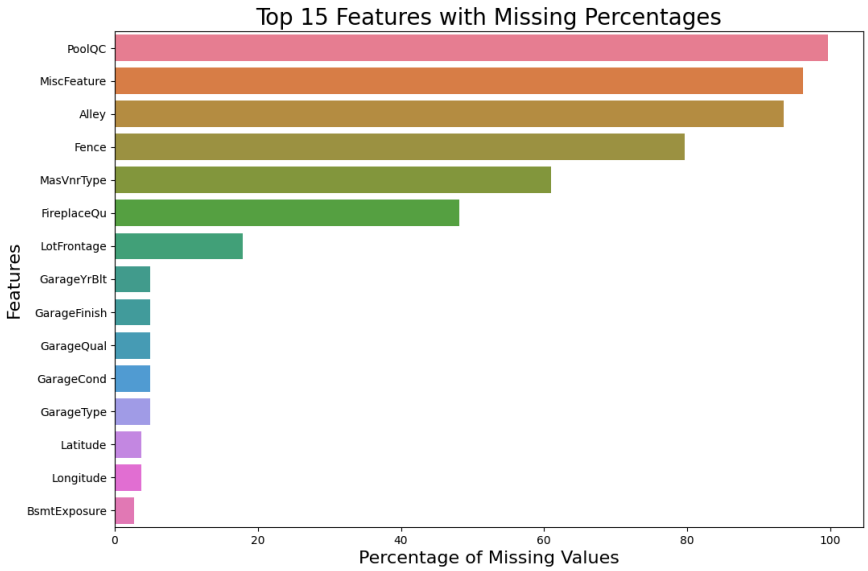

Figure 1.3: Using seaborn horizontal bar plots to visualize missing data.

A horizontal bar plot using seaborn allows you to list features with the highest missing values in a vertical format, adding both readability and aesthetic value.

Handling missing values is more than just a technical requirement; it's a significant step that can influence the quality of your machine learning models. Understanding and visualizing these missing values are the first steps in this intricate dance.

1.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Papers

Dean De Cock. "Ames, Iowa: Alternative to the Boston Housing Data as an End of Semester Regression Project". *Journal of Statistics Education*, 19(3), 2011.
https://jse.amstat.org/v19n3/decock.pdf

Resources

Dean De Cock. *Ames Housing Dataset*. 2011.
https://jse.amstat.org/v19n3/decock/AmesHousing.txt
Dean De Cock. *Ames Housing Data Dictionary*. 2011.
https://jse.amstat.org/v19n3/decock/DataDocumentation.txt

1.5 Summary

In this chapter, you embarked on an exploration of the Ames Properties dataset, a comprehensive collection of housing data tailored for data science applications.

Specifically, you learned:

- About the context of the Ames dataset, including the pioneers and academic importance behind it.
- How to extract dataset dimensions, data types, and missing values.
- How to use packages like `missingno`, Matplotlib, and seaborn to quickly visualize your missing data.

As you learn about missing data, you will see how you can fill in the missing values in the next chapter.

Exploring Dictionaries, Classifying Variables, and Imputing Data

2

The real estate market is a complex ecosystem driven by numerous variables such as location, property features, market trends, and economic indicators. One dataset that offers a deep dive into this complexity is the Ames Housing dataset. Originating from Ames, Iowa, this dataset comprises various properties and their characteristics, ranging from the type of alley access to the overall condition of the property.

In this chapter, you aim to take a closer look at this dataset using data science techniques. Specifically, you'll focus on how to identify categorical and numerical variables, as understanding these variables is crucial for any data-driven decision-making process.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ The Importance of a Data Dictionary
- ▷ Identifying Categorical and Numerical Variables
- ▷ Missing Data Imputation

2.1 The Importance of a Data Dictionary

Before you delve into the code and dataset, let's discuss the value of a data dictionary. A data dictionary is essentially a *map* that describes the nature of the data you're dealing with. It explains each variable, indicating whether it is categorical or numerical and, in the case of categorical, what the variable codes mean. This is particularly helpful when working with comprehensive datasets like the Ames Housing dataset.

Understanding the levels of measurement is fundamental in this context. Levels of measurement, namely *nominal, ordinal, interval, and ratio*, provide a framework for interpreting variables. Nominal variables represent categories without any inherent order (such as user ID), ordinal variables have a meaningful order but unequal intervals (such as ranking), interval variables have consistent intervals but no true zero point to make division

meaningful (such as timestamp), and ratio variables have a meaningful order, consistent intervals, and a true zero point (such as length and weight). This distinction is crucial as it guides the appropriate statistical analyses and informs the interpretation of the dataset. In your case, the Ames Data Dictionary provides invaluable insights into the variables you will be analyzing, aiding in the accurate classification of variables and subsequent data analysis.

For example, the variable `MSSubClass` is explained as "Identifies the type of dwelling involved in the sale," with codes like "20" for "1-STORY 1946 & NEWER ALL STYLES" and "30" for "1-STORY 1945 & OLDER." Despite its numerical appearance, `MSSubClass` is *nominal*, i.e., a categorical variable. Without additional context or domain knowledge, these numerical codes are labels without inherent order. As labels, you cannot say the average of 20 and 30 is 25 and attempt to interpret what 25 means.

However, it's essential to note that the accurate classification of `MSSubClass` hinges on a nuanced understanding of the housing domain. With domain knowledge, one could discern the meaningful order among the categories, potentially reclassifying `MSSubClass` as an *ordinal variable*, in which you can conclude that 20 is better than 30. This emphasizes the critical role of domain expertise in refining the interpretation of variables, ensuring a more accurate representation of their nature and relationships.

2.2 Identifying Categorical and Numerical Variables

Identifying the nature of variables in a dataset is a crucial step in any data analysis task. While it's tempting to consider any variable with numerical values as a numerical variable, that's not always the case. As you saw with `MSSubClass` in the Ames data dictionary, some numerical variables are categorical—they represent codes, not measurable quantities.

Few Basic Methods To Classify Data Types

▷ `select_dtypes()`: Find the columns in a DataFrame of certain data types

```
# Load the Ames dataset
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Using select_dtypes()
numerical_features = Ames.select_dtypes(include=['int64', 'float64']) \
                         .columns.tolist()
categorical_features = Ames.select_dtypes(include=['object', 'category']) \
                           .columns.tolist()
```

Listing 2.1: Find columns using `select_dtypes()`

▷ `describe()`: Gather basic descriptive statistics for numerical columns (and ignore non-numerical columns). Hence you can shortlist all the numerical columns from their output

```
# Using describe() to automatically extract numerical features
numerical_features = Ames.describe().columns.tolist()
```

Listing 2.2: Using `describe()` to find numerical columns

▷ `nunique()`: Count the number of unique values in each column, in which categorical columns should only have a few unique values

```
# Data dictionary and domain knowledge could be useful in setting the threshold
threshold = 10
categorical_features = Ames.columns[Ames.nunique() <= threshold].tolist()
```

Listing 2.3: Find categorical columns using the heuristic of unique values

However, the `threshold` above is set arbitrarily, usually depending on your domain knowledge about the data.

▷ `value_counts()`: Count the number of occurrences of each unique value

```
# Using value_counts() on each column or feature
for column in Ames.columns:
    print(Ames[column].value_counts())
```

Listing 2.4: Counting values from each column

The `value_counts()` method can be used to explore the unique values and their counts for each feature. If a feature has a limited set of unique values, it might be categorical.

▷ `info()`: This prints a concise summary of the DataFrame, including the count of non-null values and the data type of each column

```
# Using info() on the Ames Dataset
Ames.info()
```

Listing 2.5: Using `info()` to get summary of a DataFrame

In summary, this code combines all the above snippets to investigate a DataFrame:

```
# Load the Ames dataset
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Using select_dtypes()
numerical_features = Ames.select_dtypes(include=['int64', 'float64']).columns.tolist()
categorical_features = Ames.select_dtypes(include=['object', 'category']).columns.tolist()
print("Numerical features (int64 and float64):", numerical_features)
print("Categorical features (object and category):", categorical_features)

# Using describe() to automatically extract numerical features
numerical_features = Ames.describe().columns.tolist()
print("Numerical features from describe():", numerical_features)

# Data dictionary and domain knowledge could be useful in setting the threshold
threshold = 10
categorical_features = Ames.columns[Ames.nunique() <= threshold].tolist()
print("Categorical features based on unique values:", categorical_features)

# Using value_counts() on each column or feature
```

```
print("Value counts:")
for column in Ames.columns:
    print(Ames[column].value_counts())

# Using info() on the Ames Dataset
print("info():")
Ames.info()
```

Listing 2.6: Investigate a DataFrame using pandas functions

Running the above code will print you the following:

```
Numerical features (int64 and float64): ['PID', 'GrLivArea', 'SalePrice', 'MSSubClass',
  'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
  'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',
  '2ndFlrSF', 'LowQualFinSF', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath',
  'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt',
  'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch',
  'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold', 'GeoRefNo', 'Latitude',
  'Longitude']
Categorical features (object and category): ['MSZoning', 'Street', 'Alley', 'LotShape',
  'LandContour', 'Utilities', 'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1',
  'Condition2', 'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
  'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
  'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'Heating', 'HeatingQC',
  'CentralAir', 'Electrical', 'KitchenQual', 'Functional', 'FireplaceQu', 'GarageType',
  'GarageFinish', 'GarageQual', 'GarageCond', 'PavedDrive', 'PoolQC', 'Fence',
  'MiscFeature', 'SaleType', 'SaleCondition', 'Prop_Addr']
Numerical features from describe(): ['PID', 'GrLivArea', 'SalePrice', 'MSSubClass',
  'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
  'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',
  '2ndFlrSF', 'LowQualFinSF', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath',
  'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt',
  'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch',
  'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold', 'GeoRefNo', 'Latitude',
  'Longitude']
Categorical features based on unique values: ['MSZoning', 'Street', 'Alley', 'LotShape',
  'LandContour', 'Utilities', 'LotConfig', 'LandSlope', 'Condition1', 'Condition2',
  'BldgType', 'HouseStyle', 'OverallQual', 'OverallCond', 'RoofStyle', 'RoofMatl',
  'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual', 'BsmtCond',
  'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'Heating', 'HeatingQC', 'CentralAir',
  'Electrical', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath', 'BedroomAbvGr',
  'KitchenAbvGr', 'KitchenQual', 'Functional', 'Fireplaces', 'FireplaceQu',
  'GarageType', 'GarageFinish', 'GarageCars', 'GarageQual', 'GarageCond', 'PavedDrive',
  'PoolArea', 'PoolQC', 'Fence', 'MiscFeature', 'YrSold', 'SaleType', 'SaleCondition']
Value counts:
PID
909176150    1
923203100    1
909250220    1
            ..
923229010    1
528382020    1
906223180    1
```

```
Name: count, Length: 2579, dtype: int64

...

SaleCondition
Normal     2413
Partial      82
Abnorml      61
Family       17
Alloca        4
AdjLand       2
Name: count, dtype: int64

...

Longitude
-93.638398    5
-93.636432    5
-93.621618    5
             ..
-93.643655    1
-93.605005    1
-93.682220    1
Name: count, Length: 2423, dtype: int64
info():
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2579 entries, 0 to 2578
Data columns (total 85 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   PID            2579 non-null   int64
 1   GrLivArea      2579 non-null   int64
 2   SalePrice      2579 non-null   int64
...
 82  Prop_Addr      2559 non-null   object
 83  Latitude       2482 non-null   float64
 84  Longitude      2482 non-null   float64
dtypes: float64(14), int64(27), object(44)
memory usage: 1.7+ MB
```

Output 2.1: Overview of a DataFrame

Converting Numerical Features to Categorical Features

Most of the basic methods above would classify `MSSubClass` as a numerical feature. However, as highlighted earlier, this feature is, in fact, a categorical variable. Similarly, features like `MoSold` (Month Sold) and `YrSold` (Year Sold) are numerical in nature, but they can often be treated as categorical variables, especially when there is no interest in performing mathematical operations on them. You can use the `astype()` method in pandas to convert `MSSubClass`, `MoSold`, and `YrSold` to categorical features.

```
Ames['MSSubClass'] = Ames['MSSubClass'].astype('object')
Ames['YrSold'] = Ames['YrSold'].astype('object')
Ames['MoSold'] = Ames['MoSold'].astype('object')
```

Listing 2.7: Converting data type

After performing this conversion, the count of columns with the "object" data type has increased to 47 (from the previous 44), while "int64" has dropped to 24 (from 27).

```
# Determine the data type for each feature after conversion
data_types = Ames.dtypes

# Tally the total by data type
type_counts = data_types.value_counts()

print(type_counts)
```

Listing 2.8: Tallying column types from a DataFrame

This shows:

```
object     47
int64      24
float64    14
dtype: int64
```

Output 2.2: Count of columns of different type

A careful assessment of the data dictionary, the nature of the dataset, and domain expertise can contribute to properly reclassifying data types.

The following is the complete code to count the number of columns of each data type:

```
# Load the Ames dataset
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Reassign data type
Ames['MSSubClass'] = Ames['MSSubClass'].astype('object')
Ames['YrSold'] = Ames['YrSold'].astype('object')
Ames['MoSold'] = Ames['MoSold'].astype('object')

# Determine the data type for each feature after conversion
data_types = Ames.dtypes

# Tally the total by data type
type_counts = data_types.value_counts()

print(type_counts)
```

Listing 2.9: Count the number of columns

2.3 Missing Data Imputation

Dealing with missing data is a challenge that every data scientist faces. Ignoring missing values or handling them inadequately can lead to skewed analysis and incorrect conclusions. The choice of imputation technique often depends on the nature of the data—categorical or numerical. In addition, information in the data dictionary will be useful (such as in the case of Pool Quality) where a missing value ("NA") has a meaning, namely the absence of this feature for a particular property.

Data Imputation For Categorical Features

You can identify categorical data types and rank them in the order in which they are most affected by missing data.

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Calculating the percentage of missing values for each column
missing_data = Ames.isnull().sum()
missing_percentage = (missing_data / len(Ames)) * 100
data_type = Ames.dtypes

# Combining the counts and percentages into a DataFrame for better visualization
missing_info = pd.DataFrame({'Missing Values': missing_data,
                             'Percentage': missing_percentage,
                             'Data Type': data_type})

# Sorting the DataFrame by the percentage of missing values in descending order
missing_info = missing_info.sort_values(by='Percentage', ascending=False)

# Display columns with missing values of 'object' data type
print(missing_info[(missing_info['Missing Values'] > 0) &
                   (missing_info['Data Type'] == 'object')])
```

Listing 2.10: Find columns with the highest percentage of missing values

This shows:

```
              Missing Values  Percentage Data Type
PoolQC                  2570   99.651028    object
MiscFeature             2482   96.238852    object
Alley                   2411   93.485847    object
Fence                   2054   79.643273    object
FireplaceQu             1241   48.119426    object
GarageCond               129    5.001939    object
GarageQual               129    5.001939    object
GarageFinish             129    5.001939    object
GarageType               127    4.924389    object
BsmtExposure              71    2.753005    object
BsmtFinType2              70    2.714230    object
BsmtFinType1              69    2.675456    object
BsmtQual                  69    2.675456    object
```

```
BsmtCond                 69    2.675456    object
Prop_Addr                20    0.775494    object
MasVnrType               14    0.542846    object
Electrical                1    0.038775    object
```

Output 2.3: Columns with the highest percentage of missing values

If you check the data dictionary, you should know that missing values for the categorical features above indicate the absence of those features for a given property. The exception is "Electrical" (surely a house can't be with no electricity). Hence you can impute that one missing data point of "Electrical" with the *mode*. The other features are imputed with the Python string `"None"`.

```
mode_value = Ames['Electrical'].mode()[0]
Ames['Electrical'].fillna(mode_value, inplace=True)
print(mode_value)
print(Ames['Electrical'].isnull().sum())
```

Listing 2.11: Filling missing values

This verifies the mode value is `"SBrkr"` and the missing value in "Electrical" column is fixed:

```
SBrkr
0
```

Output 2.4: The mode value in "Electrical" column and the number of missing values

Afterward, you can replace all missing values with the string `"None"`:

```
missing_categorical = missing_info[(missing_info['Missing Values'] > 0)
                                   & (missing_info['Data Type'] == 'object')]

for item in missing_categorical.index.tolist():
    Ames[item].fillna("None", inplace=True)

print(Ames[missing_categorical.index].isnull().sum())
```

Listing 2.12: Replacing missing values with string `"None"`

This confirms that there are now no more missing values for categorical features:

```
PoolQC          0
MiscFeature     0
Alley           0
Fence           0
FireplaceQu     0
GarageCond      0
GarageQual      0
GarageFinish    0
GarageType      0
BsmtExposure    0
BsmtFinType2    0
BsmtFinType1    0
BsmtQual        0
```

```
BsmtCond        0
Prop_Addr       0
MasVnrType      0
Electrical      0
```

Output 2.5: Verify the number of missing values in each column

The complete code is as follows:

```
# Load the Ames dataset
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Calculating the percentage of missing values for each column
missing_data = Ames.isnull().sum()
missing_percentage = (missing_data / len(Ames)) * 100
data_type = Ames.dtypes

# Combining the counts and percentages into a DataFrame for better visualization
missing_info = pd.DataFrame({'Missing Values': missing_data,
                             'Percentage': missing_percentage,
                             'Data Type': data_type})

# Sorting the DataFrame by the percentage of missing values in descending order
missing_info = missing_info.sort_values(by='Percentage', ascending=False)

# Display columns with missing values of 'object' data type
print(missing_info[(missing_info['Missing Values'] > 0) &
                   (missing_info['Data Type'] == 'object')])

mode_value = Ames['Electrical'].mode()[0]
Ames['Electrical'].fillna(mode_value, inplace=True)
print(mode_value)
print(Ames['Electrical'].isnull().sum())

missing_categorical = missing_info[(missing_info['Missing Values'] > 0)
                           & (missing_info['Data Type'] == 'object')]

for item in missing_categorical.index.tolist():
    Ames[item].fillna("None", inplace=True)

print(Ames[missing_categorical.index].isnull().sum())
```

Listing 2.13: Imputation of missing values

Data Imputation For Numerical Features

You can apply the same technique demonstrated above to identify numerical data types and rank them in the order in which they are most affected by missing data.

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

Ames = pd.read_csv('Ames.csv')

# Calculating the percentage of missing values for each column
missing_data = Ames.isnull().sum()
missing_percentage = (missing_data / len(Ames)) * 100
data_type = Ames.dtypes

# Combining the counts and percentages into a DataFrame for better visualization
missing_info = pd.DataFrame({'Missing Values': missing_data,
                             'Percentage': missing_percentage,
                             'Data Type': data_type})

# Sorting the DataFrame by the percentage of missing values in descending order
missing_info = missing_info.sort_values(by='Percentage', ascending=False)

# Display columns with missing values of numeric data type
print(missing_info[(missing_info['Missing Values'] > 0)
                   & (missing_info['Data Type'] == np.number)])
```

Listing 2.14: Finding numerical columns with missing data

This prints:

```
              Missing Values  Percentage Data Type
LotFrontage              462   17.913920   float64
GarageYrBlt              129    5.001939   float64
Longitude                 97    3.761148   float64
Latitude                  97    3.761148   float64
GeoRefNo                  20    0.775494   float64
MasVnrArea                14    0.542846   float64
BsmtFullBath               2    0.077549   float64
BsmtHalfBath               2    0.077549   float64
BsmtFinSF2                 1    0.038775   float64
GarageArea                 1    0.038775   float64
BsmtFinSF1                 1    0.038775   float64
BsmtUnfSF                  1    0.038775   float64
TotalBsmtSF                1    0.038775   float64
GarageCars                 1    0.038775   float64
```

Output 2.6: Numerical columns with most missing data

The above illustrates that there are fewer instances of missing numerical data than missing categorical data. However, the data dictionary is not as useful for a straightforward imputation. Whether you should impute missing data in data science largely depends on the goal of the analysis. Often, a data scientist may generate multiple imputations to account for the uncertainty in the imputation process. Common multiple imputation methods include (but are not limited to) mean, median, and regression imputation. As a baseline, you will use mean imputation here, but you may refer to other techniques depending on the task at hand.

```
# Initialize a DataFrame to store the concise information
concise_info = pd.DataFrame(columns=['Feature',
                                     'Missing Values After Imputation',
                                     'Mean Value Used to Impute'])

# Identify and impute missing numerical values, and store the related concise information
missing_numeric_df = missing_info[(missing_info['Missing Values'] > 0) &
                                  (missing_info['Data Type'] == np.number)]

for item in missing_numeric_df.index.tolist():
    mean_value = Ames[item].mean(skipna=True)
    Ames[item].fillna(mean_value, inplace=True)

    # Append the concise information to the concise_info DataFrame
    concise_info.loc[len(concise_info)] = pd.Series({
        'Feature': item,
        'Missing Values After Imputation': Ames[item].isnull().sum(),
        # This should be 0 as you are imputing all missing values
        'Mean Value Used to Impute': mean_value
    })

# Display the concise_info DataFrame
print(concise_info)
```

Listing 2.15: Filling missing values for numerical columns

This prints:

```
         Feature Missing Values After Imputation  Mean Value Used to Impute
0    LotFrontage                                0               6.851063e+01
1    GarageYrBlt                                0               1.976997e+03
2      Longitude                                0              -9.364254e+01
3       Latitude                                0               4.203456e+01
4       GeoRefNo                                0               7.136762e+08
5     MasVnrArea                                0               9.934698e+01
6   BsmtFullBath                                0               4.353900e-01
7   BsmtHalfBath                                0               6.208770e-02
8     BsmtFinSF2                                0               5.325950e+01
9     GarageArea                                0               4.668646e+02
10    BsmtFinSF1                                0               4.442851e+02
11     BsmtUnfSF                                0               5.391947e+02
12   TotalBsmtSF                                0               1.036739e+03
13    GarageCars                                0               1.747867e+00
```

Output 2.7: Numerical columns with missing values fixed

At times, you may also opt to leave the missing value without any imputation to retain the authenticity of the original dataset and remove observations that lack complete and accurate data if necessary. Alternatively, you may also try to build a machine learning model to *guess* the missing value based on other data within the same row, a principle behind imputation by regression. As a final step in the above baseline imputation, let's cross-check if there are any remaining missing values.

```
missing_values_count = Ames.isnull().sum().sum()
print(f'The DataFrame has a total of {missing_values_count} missing values.')
```

Listing 2.16: Counting number of missing values over the entire DataFrame

You should see:

```
The DataFrame has a total of 0 missing values.
```

Output 2.8: Count of missing values in the DataFrame

Congratulations! You have successfully imputed every missing value in the Ames dataset using baseline operations. It's important to note that numerous other techniques exist for imputing missing data. As a data scientist, exploring various options and determining the most appropriate method for the given context is crucial to producing reliable and meaningful results.

The complete code is as follows:

```
# Load the Ames dataset
import pandas as pd
import numpy as np
Ames = pd.read_csv('Ames.csv')

# Calculating the percentage of missing values for each column
missing_data = Ames.isnull().sum()
missing_percentage = (missing_data / len(Ames)) * 100
data_type = Ames.dtypes

# Combining the counts and percentages into a DataFrame for better visualization
missing_info = pd.DataFrame({'Missing Values': missing_data,
                             'Percentage': missing_percentage,
                             'Data Type': data_type})

# Sorting the DataFrame by the percentage of missing values in descending order
missing_info = missing_info.sort_values(by='Percentage', ascending=False)

# Display columns with missing values of numeric data type
print(missing_info[(missing_info['Missing Values'] > 0)
                   & (missing_info['Data Type'] == np.number)])

# Initialize a DataFrame to store the concise information
concise_info = pd.DataFrame(columns=['Feature',
                                     'Missing Values After Imputation',
                                     'Mean Value Used to Impute'])

# Identify and impute missing numerical values, and store the related concise information
missing_numeric_df = missing_info[(missing_info['Missing Values'] > 0) &
                                  (missing_info['Data Type'] == np.number)]

for item in missing_numeric_df.index.tolist():
    mean_value = Ames[item].mean(skipna=True)
    Ames[item].fillna(mean_value, inplace=True)
```

```
    # Append the concise information to the concise_info DataFrame
    concise_info.loc[len(concise_info)] = pd.Series({
        'Feature': item,
        'Missing Values After Imputation': Ames[item].isnull().sum(),
        # This should be 0 as you are imputing all missing values
        'Mean Value Used to Impute': mean_value
    })

# Display the concise_info DataFrame
print(concise_info)

missing_values_count = Ames.isnull().sum().sum()
print(f'The DataFrame has a total of {missing_values_count} missing values.')
```

Listing 2.17: Data imputation on the DataFrame

2.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

Imputation of missing values. Scikit-Learn User Guide.
https://scikit-learn.org/stable/modules/impute.html
Imputation (statistics). Wikipedia.
https://en.wikipedia.org/wiki/Imputation_(statistics)

2.5 Summary

In this chapter, you explored the Ames Housing dataset through the lens of data science techniques. You saw the importance of a data dictionary in understanding the variables of the dataset and dove into Python code snippets that help identify and handle these variables effectively.

Understanding the nature of the variables you're working with is crucial for any data-driven decision-making process. As you've seen, the Ames data dictionary serves as a valuable guide in this respect. Coupled with the powerful data manipulation libraries in Python, navigating complex datasets like the Ames Housing dataset becomes a much more manageable task.

Specifically, you learned:

- ▷ The importance of a data dictionary when assessing data types and imputation strategies.
- ▷ Identification and reclassification methods for numerical and categorical features.
- ▷ How to impute missing categorical and numerical features using the pandas library.

In the next chapter, you will learn the more advanced pandas operations.

Beyond SQL: Transforming Real Estate Data into Actionable Insights with Pandas

3

In data analysis, SQL stands as a mighty tool, renowned for its robust capabilities in managing and querying databases. The pandas library in Python brings SQL-like functionalities to data scientists, enabling sophisticated data manipulation and analysis without the need for a traditional SQL database. In the following, you will apply SQL-like functions in Python to dissect and understand data.

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ▷ Exploring Data with Pandas' `DataFrame.query()` Method
- ▷ Aggregating and Grouping Data
- ▷ Mastering Row and Column Selection in Pandas
- ▷ Harnessing Pivot Table for In-Depth Housing Market Analysis

3.1 Exploring Data with Pandas' `DataFrame.query()` Method

The `DataFrame.query()` method in pandas allows for the selection of rows based on a specified condition, similar to the SQL `SELECT` statement. Starting with the basics, you filter data based on single and multiple conditions, thereby laying the foundation for more complex data querying.

```
import pandas as pd

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Simple querying: Select houses priced above $600,000
high_value_houses = Ames.query('SalePrice > 600000')
print(high_value_houses)
```

Listing 3.1: Filtering rows from DataFrame using `query()` method

In the code above, you utilize the `DataFrame.query()` method from pandas to filter out houses priced above $600,000, storing the result in a new DataFrame called `high_value_houses`. This method allows for concise and readable querying of the data based on a condition specified as a string. In this case, `'SalePrice > 600000'`.

The resulting DataFrame below showcases the selected high-value properties. The query effectively narrows down the dataset to houses with a sale price exceeding $600,000, where only five houses meet this criterion. The filtered view provides a focused look at the upper echelon of the housing market in the Ames dataset, offering insights into the characteristics and locations of the highest-valued properties.

```
            PID  GrLivArea  ...   Latitude  Longitude
65    528164060       2470  ...  42.058475 -93.656810
584   528150070       2364  ...  42.060462 -93.655516
1007  528351010       4316  ...  42.051982 -93.657450
1325  528320060       3627  ...  42.053228 -93.657649
1639  528110020       2674  ...  42.063049 -93.655918

[5 rows x 85 columns]
```

Output 3.1: Five rows from the DataFrame that matched the query

In the next example below, let's further explore the capabilities of the `DataFrame.query()` method to filter the Ames Housing dataset based on more specific criteria. The query selects houses that have more than 3 bedrooms (`BedroomAbvGr > 3`) and are priced below $300,000 (`SalePrice < 300000`). This combination of conditions is achieved using the logical AND operator (`&`), allowing you to apply multiple filters to the dataset simultaneously.

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

# Advanced querying: Select houses with more than 3 bedrooms and priced below $300,000
specific_houses = Ames.query('BedroomAbvGr > 3 & SalePrice < 300000')
print(specific_houses)
```

Listing 3.2: Running compound query

The result of this query is stored in a new DataFrame called `specific_houses`, which contains all the properties that satisfy both conditions. By printing `specific_houses`, you can examine the details of homes that are both relatively large (in terms of bedrooms) and affordable, targeting a specific segment of the housing market that could interest families looking for spacious living options within a certain budget range.

```
            PID  GrLivArea  ...   Latitude  Longitude
5     908128060       1922  ...  42.018988 -93.671572
23    902326030       2640  ...  42.029358 -93.612289
33    903400180       1848  ...  42.029544 -93.627377
38    527327050       2030  ...  42.054506 -93.631560
40    528326110       2172  ...  42.055785 -93.651102
...         ...        ...  ...        ...        ...
```

```
2539  905101310       1768  ...  42.033393 -93.671295
2557  905107250       1440  ...  42.031349 -93.673578
2562  535101110       1584  ...  42.048256 -93.619860
2575  905402060       1733  ...  42.027669 -93.666138
2576  909275030       2002  ...        NaN        NaN

[352 rows x 85 columns]
```

Output 3.2: Rows matching the compound query

The advanced query successfully identified a total of 352 houses from the Ames Housing dataset that meet the specified criteria: having more than 3 bedrooms and a sale price below $300,000. This subset of properties highlights a significant portion of the market that offers spacious living options without breaking the budget, catering to families or individuals searching for affordable yet ample housing. To further explore the dynamics of this subset, let's visualize the relationship between sale prices and ground living areas, with a color code indicating the number of bedrooms. This graphical representation will help you understand how living space and bedroom count influence the affordability and appeal of these homes within the specified criteria.

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

Ames = pd.read_csv('Ames.csv')
specific_houses = Ames.query('BedroomAbvGr > 3 & SalePrice < 300000')

# Visualizing the advanced query results
plt.figure(figsize=(10, 6))
sns.scatterplot(x='GrLivArea', y='SalePrice', hue='BedroomAbvGr',
                data=specific_houses, palette='viridis')
plt.title('Sales Price vs. Ground Living Area')
plt.xlabel('Ground Living Area (sqft)')
plt.ylabel('Sales Price ($)')
plt.legend(title='Bedrooms Above Ground')
plt.show()
```

Listing 3.3: Creating a scatter plot to show the sales price

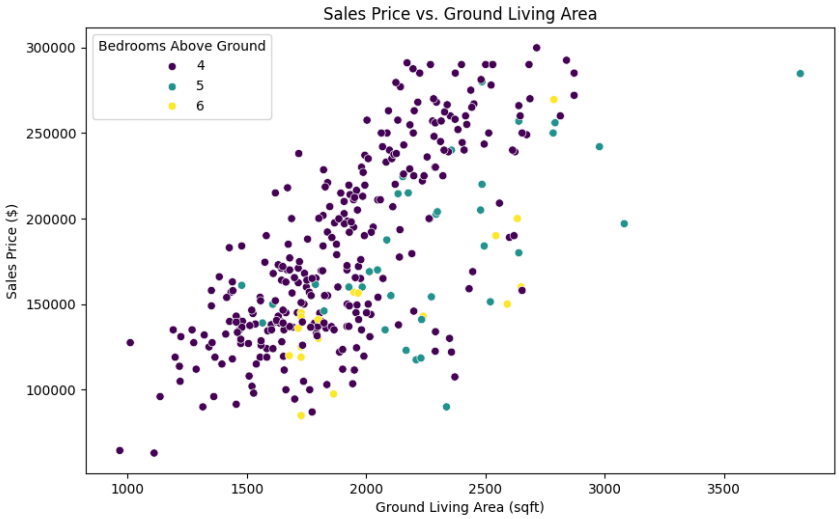

Figure 3.1: Scatter plot showing the distribution of sales price related to the number of bedrooms and living area

The scatter plot above demonstrates the connection between sale price, living area, and bedroom count within this segment of the housing market. It highlights how larger living spaces and additional bedrooms contribute to the value, offering insights for potential buyers and investors focusing on spacious yet affordable homes. This visual analysis not only makes the data more accessible but also underpins the practical utility of Pandas in uncovering key market trends.

3.2 Aggregating and Grouping Data

Aggregation and grouping are pivotal in summarizing data insights. Building on the foundational querying techniques explored in the first part of your exploration, let's delve deeper into the power of data aggregation and grouping in Python. Similar to `GROUP BY` clause in SQL, pandas offers a robust `groupby()` method, enabling you to segment your data into subsets for detailed analysis. This next phase of your journey focuses on leveraging these capabilities to uncover hidden patterns and insights within the Ames Housing dataset. Specifically, you'll examine the average sale prices of homes with more than three bedrooms, priced below $300,000, across different neighborhoods. By aggregating this data, you aim to highlight the variability in housing affordability and inventory across the spatial canvas of Ames, Iowa.

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

# Advanced querying: Select houses with more than 3 bedrooms and priced below $300,000
specific_houses = Ames.query('BedroomAbvGr > 3 & SalePrice < 300000')

# Group by neighborhood, then calculate the average and total price, and count the houses
```

```
grouped_data = specific_houses.groupby('Neighborhood').agg({
    'SalePrice': ['mean', 'count']
})

# 'Neighborhood' is the index but you should rename the columns for clarity
grouped_data.columns = ['Average Sales Price', 'House Count']

# Round the average sale price to 2 decimal places
grouped_data['Averages Sales Price'] = grouped_data['Average Sales Price'].round(2)

print(grouped_data)
```

Listing 3.4: Aggregating data using `groupby()`

This shows:

```
              Average Sales Price  House Count
Neighborhood
BrDale                  113700.00            1
BrkSide                 154840.00           10
ClearCr                 206756.31           13
CollgCr                 233504.17           12
Crawfor                 199946.68           19
Edwards                 142372.41           29
Gilbert                 222554.74           19
IDOTRR                  146953.85           13
MeadowV                 135966.67            3
Mitchel                 152030.77           13
NAmes                   158835.59           59
NPkVill                 143000.00            1
NWAmes                  203846.28           39
NoRidge                 272222.22           18
NridgHt                 275000.00            3
OldTown                 142586.72           43
SWISU                   147493.33           15
Sawyer                  148375.00           16
SawyerW                 217952.06           16
Somerst                 247333.33            3
StoneBr                 270000.00            1
Timber                  247652.17            6
```

Output 3.3: Average sales price of each neighborhood

Using Seaborn, let's create a chart to showcase the average sales price by neighborhood, complemented by annotations of house counts to illustrate both price and volume in a single, cohesive graph.

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

Ames = pd.read_csv('Ames.csv')
specific_houses = Ames.query('BedroomAbvGr > 3 & SalePrice < 300000')
grouped_data = specific_houses.groupby('Neighborhood').agg({
```

```
    'SalePrice': ['mean', 'count']
})
grouped_data.columns = ['Average Sales Price', 'House Count']
grouped_data['Averages Sales Price'] = grouped_data['Average Sales Price'].round(2)

# 'Neighborhood' was index, reset to make it a column then sort by price
grouped_data_reset = grouped_data.reset_index().sort_values(by='Average Sales Price')

# Set the aesthetic style of the plots
sns.set_theme(style="whitegrid")

# Create the bar plot
plt.figure(figsize=(12, 8))
barplot = sns.barplot(
    x='Neighborhood',
    y='Average Sales Price',
    data=grouped_data_reset,
    palette="coolwarm",
    hue='Neighborhood',
    legend=False,
    errorbar=None  # Removes the confidence interval bars
)

# Rotate the x-axis labels for better readability
plt.xticks(rotation=45)

# Annotate each bar with the house count, using enumerate to get the index for positioning
for index, value in enumerate(grouped_data_reset['Average Sales Price']):
    house_count = grouped_data_reset.loc[index, 'House Count']
    plt.text(index, value, f'{house_count}', ha='center', va='bottom')

plt.title('Average Sales Price by Neighborhood', fontsize=18)
plt.xlabel('Neighborhood')
plt.ylabel('Average Sales Price ($)')

plt.tight_layout()  # Adjust the layout
plt.show()
```

Listing 3.5: Create a bar chart showing the neighborhood in order of ascending sales price

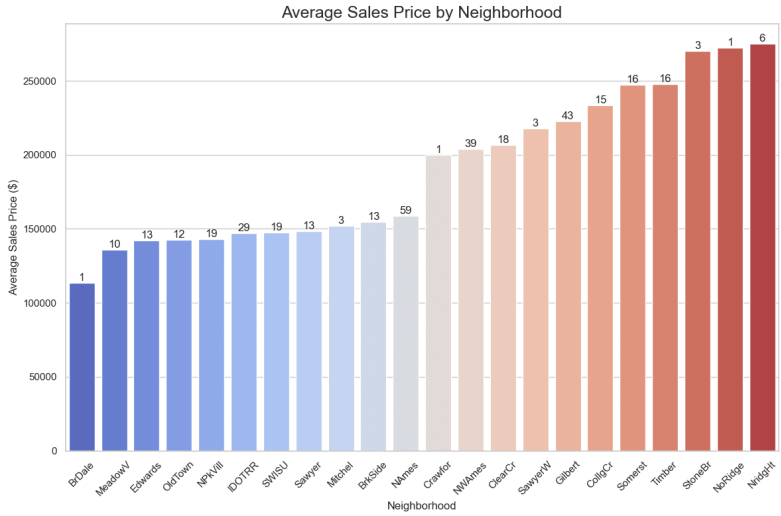

Figure 3.2: Comparing neighborhoods by ascending average sales price

Same data as the table printed by Listing 3.4, but easier to see the distribution and better to convey your idea. Such presentation is greatly helped by the SQL-like functions from pandas to distill the data into a useful format.

3.3 Mastering Row and Column Selection in Pandas

Selecting specific subsets of data from DataFrames is a frequent necessity. Two powerful methods at your disposal are `DataFrame.loc[]` and `DataFrame.iloc[]`. Both serve similar purposes—to select data—but they differ in how they reference the rows and columns.

Understanding The `DataFrame.loc[]` Method

`DataFrame.loc[]` is a label-based data selection method, meaning you use the labels of rows and columns to select the data. It's highly intuitive for selecting data based on column names and row indexes when you know the specific labels you're interested in.

Syntax: `DataFrame.loc[row_label, column_label]`

The labels for rows and columns can be a single value of the index or column name, a vector of Boolean value for Boolean indexing, or a Python list of many such values if multiple rows or columns are desired. As an example, let's select all houses with more than 3 bedrooms, priced below $300,000, in specific neighborhoods known for their higher average sale prices (based on your earlier findings), and display their "Neighborhood", "SalePrice" and "GrLivArea". You will use Boolean indexing for rows and a list of column names for columns.

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

# This is a list of neighborhoods with higher average sale prices
```

```
high_value_neighborhoods = ['NridgHt', 'NoRidge', 'StoneBr']

# Use df.loc[] to select houses in high-value neighborhoods based on your conditions
high_value_houses = Ames.loc[(Ames['BedroomAbvGr'] > 3) &
                             (Ames['SalePrice'] < 300000) &
                             (Ames['Neighborhood'].isin(high_value_neighborhoods)),
                             ['Neighborhood', 'SalePrice', 'GrLivArea']]

print(high_value_houses.head())
```

Listing 3.6: Using `DataFrame.loc[]`

```
    Neighborhood  SalePrice  GrLivArea
40       NoRidge     291000       2172
162      NoRidge     285000       2225
460      NridgHt     250000       2088
468      NoRidge     268000       2295
490      NoRidge     260000       2417
```

Output 3.4: Rows and columns filtered using `DataFrame.loc[]`

Understanding The DataFrame.iloc[] Method

In contrast, `DataFrame.iloc[]` is an integer-location based indexing method. This means you use integers to specify the rows and columns you want to select. It's particularly useful to access data by its position in the DataFrame.

Syntax: `DataFrame.iloc[row_position, column_position]`

Instead of names, `iloc[]` expects integer offsets such that you can access rows and columns like an array. As an example, let's uncover affordable housing options within the Ames dataset that do not compromise on space, specifically targeting homes with at least 3 bedrooms above grade and priced below $300,000 outside of high-value neighborhoods. Boolean indexing with `loc[]` is used to find the match and Python slicing syntax is used as in `iloc[:5]` to find the lowest value results.

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

high_value_neighborhoods = ['NridgHt', 'NoRidge', 'StoneBr']

# Filter for houses not in the 'high_value_neighborhoods',
# with at least 3 bedrooms above grade, and priced below $300,000
low_value_spacious = Ames.loc[(~Ames['Neighborhood'].isin(high_value_neighborhoods)) &
                              (Ames['BedroomAbvGr'] >= 3) &
                              (Ames['SalePrice'] < 300000)]

# Sort these houses by 'SalePrice' to highlight the lower end explicitly
low_value_spacious = low_value_spacious.sort_values(by='SalePrice').reset_index(drop=True)
```

```
# Using df.iloc to select and print the first 5 observations of such low-value houses
low_value_spacious_first_5 = low_value_spacious.iloc[:5, :]

# Print only relevant columns
print(low_value_spacious_first_5[['Neighborhood', 'SalePrice', 'GrLivArea']])
```

Listing 3.7: Using `DataFrame.iloc[]`

```
  Neighborhood  SalePrice  GrLivArea
0       IDOTRR      40000       1317
1       IDOTRR      50000       1484
2       IDOTRR      55000       1092
3       Sawyer      62383        864
4      Edwards      63000       1112
```

Output 3.5: Rows and columns filtered by `DataFrame.iloc[]`

In your exploration of `DataFrame.loc[]` and `DataFrame.iloc[]`, you've uncovered the capabilities of pandas for row and column selection, demonstrating the flexibility and power of these methods in data analysis. You've seen `loc[]` is for label-based selection. It helps if you have specific data to look for. On the other hand, `iloc[]` depends on the ordering of the rows and columns in the DataFrame since access is by the integer offset. It is a tool for positional selection. It is useful to segment a DataFrame, such as cropping the first five rows from a sorted DataFrame.

3.4 Harnessing Pivot Tables for In-depth Housing Market Analysis

As you venture further into the depths of the Ames Housing dataset, your analytical journey introduces you to the potent capabilities of pivot tables within pandas. Pivot tables serve as an invaluable tool for summarizing, analyzing, and presenting complex data in an easily digestible format. This technique allows you to cross-tabulate and segment data to uncover patterns and insights that might otherwise remain hidden. In this section, you'll use pivot tables to examine the relationship between neighborhood characteristics, bedroom count, and sale prices.

To set the stage for your pivot table analysis, you filter the dataset for homes priced below $300,000 and with at least one bedroom above grade. This criterion focuses on more affordable housing options, ensuring your analysis remains relevant to a broader audience. You then proceed to construct a pivot table that segments the average sale price by neighborhood and bedroom count, aiming to uncover patterns that dictate housing affordability and preference within Ames.

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Filter for houses priced below $300,000 and with at least 1 bedroom above grade
affordable_houses = Ames.query('SalePrice < 300000 & BedroomAbvGr > 0')
```

```
# Create pivot table to analyze average sale price by neighborhood and number of bedrooms
pivot_table = affordable_houses.pivot_table(values='SalePrice',
                                            index='Neighborhood',
                                            columns='BedroomAbvGr',
                                            aggfunc='mean').round(2)

# Fill missing values (combination not exist) with 0 to avoid seeing NaN
pivot_table = pivot_table.fillna(0)

# Adjust pandas display options to ensure all columns are shown
pd.set_option('display.max_columns', None)
print(pivot_table)
```

Listing 3.8: Creating a new DataFrame using pivot table

Let's take a quick view of the pivot table before we discuss some insights.

BedroomAbvGr	1	2	3	4	5	6
Neighborhood						
Blmngtn	178450.00	197931.19	0.00	0.00	0.00	0.00
Blueste	192500.00	128557.14	151000.00	0.00	0.00	0.00
BrDale	0.00	99700.00	111946.43	113700.00	0.00	0.00
BrkSide	77583.33	108007.89	140058.67	148211.11	214500.00	0.00
ClearCr	212250.00	220237.50	190136.36	209883.20	196333.33	0.00
CollgCr	154890.00	181650.00	196650.98	233504.17	0.00	0.00
Crawfor	289000.00	166345.00	193433.75	198763.94	210000.00	0.00
Edwards	59500.00	117286.27	134660.65	137332.00	191866.67	119900.00
Gilbert	0.00	172000.00	182178.30	223585.56	204000.00	0.00
Greens	193531.25	0.00	0.00	0.00	0.00	0.00
GrnHill	0.00	230000.00	0.00	0.00	0.00	0.00
IDOTRR	67378.00	93503.57	111681.13	144081.82	162750.00	0.00
Landmrk	0.00	0.00	137000.00	0.00	0.00	0.00
MeadowV	82128.57	105500.00	94382.00	128250.00	151400.00	0.00
Mitchel	176750.00	150366.67	168759.09	149581.82	165500.00	0.00
NAmes	139500.00	133098.93	146260.96	159065.22	180360.00	144062.50
NPkVill	0.00	134555.00	146163.64	143000.00	0.00	0.00
NWAmes	0.00	177765.00	183317.12	201165.00	253450.00	0.00
NoRidge	0.00	262000.00	259436.67	272222.22	0.00	0.00
NridgHt	211700.00	215458.55	264852.71	275000.00	0.00	0.00
OldTown	83333.33	105564.32	136843.57	136350.91	167050.00	97500.00
SWISU	60000.00	121044.44	132257.88	143444.44	158500.00	148633.33
Sawyer	185000.00	124694.23	138583.77	148884.62	0.00	146166.67
SawyerW	216000.00	156147.41	185192.14	211315.00	0.00	237863.25
Somerst	205216.67	191070.18	225570.39	247333.33	0.00	0.00
StoneBr	223966.67	211468.75	233750.00	270000.00	0.00	0.00
Timber	0.00	217263.64	200536.04	241202.60	279900.00	0.00
Veenker	247566.67	245150.00	214090.91	0.00	0.00	0.00

Output 3.6: Average price by neighborhood and number of bedrooms

The pivot table above provides a comprehensive snapshot of how the average sale price varies across neighborhoods with the inclusion of different bedroom counts. This analysis reveals several key insights:

- ▷ *Affordability by Neighborhood*: You can see at a glance which neighborhoods offer the most affordable options for homes with specific bedroom counts, aiding in targeted home searches.
- ▷ *Impact of Bedrooms on Price*: The table highlights how the number of bedrooms influences sale prices within each neighborhood, offering a gauge of the premium placed on larger homes.
- ▷ *Market Gaps and Opportunities*: Areas with zero values indicate a lack of homes meeting certain criteria, signaling potential market gaps or opportunities for developers and investors.

By leveraging pivot tables for this analysis, you've managed to distill complex relationships within the Ames housing market into a format that's both accessible and informative. This process not only showcases the powerful synergy between pandas and SQL-like analysis techniques but also emphasizes the importance of sophisticated data manipulation tools in uncovering actionable insights within real estate markets. As insightful as pivot tables are, their true potential is unleashed when combined with visual analysis.

To further illuminate your findings and make them more intuitive, you'll transition from numerical analysis to visual representation. A heatmap is an excellent tool for this purpose, especially when dealing with multidimensional data like this. However, to enhance the clarity of your heatmap and direct attention towards actionable data, you will employ a custom color scheme that distinctly highlights nonexistent combinations of neighborhood and bedroom counts.

```
import pandas as pd
import matplotlib.colors
import matplotlib.pyplot as plt
import seaborn as sns

Ames = pd.read_csv('Ames.csv')
affordable_houses = Ames.query('SalePrice < 300000 & BedroomAbvGr > 0')
pivot_table = affordable_houses \
              .pivot_table(values='SalePrice', index='Neighborhood',
                           columns='BedroomAbvGr', aggfunc='mean') \
              .round(2) \
              .fillna(0)

# Create a custom color map
cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", ["red", "yellow", "green"])

# Mask for "zero" values to be colored with a different shade
mask = pivot_table == 0

# Set the size of the plot
plt.figure(figsize=(14, 10))

# Create a heatmap with the mask
sns.heatmap(pivot_table,
            cmap=cmap,
            annot=True,
```

```
            fmt=".0f",
            linewidths=.5,
            mask=mask,
            cbar_kws={'label': 'Average Sales Price ($)'})

# Adding title and labels for clarity
plt.title('Average Sales Price by Neighborhood and Number of Bedrooms', fontsize=16)
plt.xlabel('Number of Bedrooms Above Grade', fontsize=12)
plt.ylabel('Neighborhood', fontsize=12)

# Display the heatmap
plt.show()
```

Listing 3.9: Showing average sales price in a heatmap

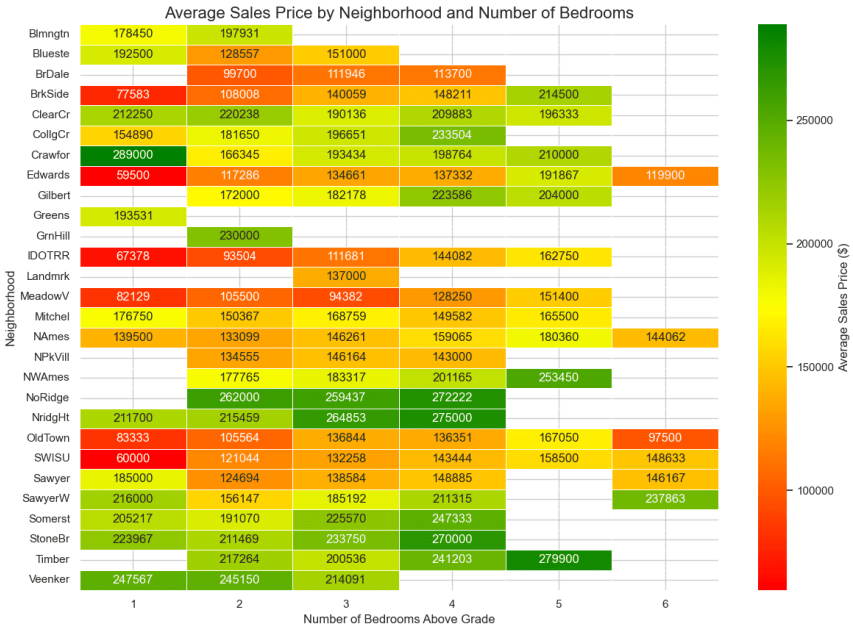

Figure 3.3: Heatmap showing the average sales price by neighborhood

The heatmap illustrates the distribution of average sale prices across neighborhoods, segmented by the number of bedrooms. This color-coded visual aid makes it immediately apparent which areas of Ames offer the most affordable housing options for families of various sizes. Moreover, the distinct shading for zero values—indicating combinations of neighborhoods and bedroom counts that do not exist—is a critical tool for market analysis. It highlights gaps in the market where demand might exist, but supply does not, offering valuable insights for developers and investors alike. Remarkably, your analysis also highlights that homes with 6 bedrooms in the "Old Town" neighborhood are listed below $100,000. This discovery points to exceptional value for larger families or investors looking for properties with high bedroom counts at affordable price points.

Through this visual exploration, you understand the dynamics of the housing market. The pivot table, complemented by the heatmap, exemplifies how sophisticated data manipulation and visualization techniques can reveal informative insights into the housing sector.

3.5 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

`DataFrame.query()` *method*. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.query.html

`DataFrame.groupby()` *method*. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html

`DataFrame.loc[]` *method*. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.loc.html

`DataFrame.iloc[]` *method*. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html

`DataFrame.pivot_table()` *method*. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot_table.html

3.6 Summary

This comprehensive journey through the Ames Housing dataset underscores the versatility and strength of pandas for conducting sophisticated data analysis, often achieving or exceeding what's possible with SQL in an environment that doesn't rely on traditional databases. From pinpointing detailed housing market trends to identifying unique investment opportunities, you've showcased a range of techniques that equip analysts with the tools needed for deep data exploration. Specifically, you learned how to:

- Leverage the `DataFrame.query()` for data selection akin to `SELECT` statement in SQL.
- Use `DataFrame.groupby()` for aggregating and summarizing data, similar to `GROUP BY` in SQL.
- Apply advanced data manipulation techniques like `DataFrame.loc[]`, `DataFrame.iloc[]`, and `DataFrame.pivot_table()` for deeper analysis.

In the next chapter, you will see more examples of manipulating DataFrames.

Harmonizing Data: A Symphony of Segmenting, Concatenating, Pivoting, and Merging

4

In a data science project, the data you collect is often not in the shape that you want it to be. Often you will need to create derived features, aggregate subsets of data into a summarized form, or select a portion of the data according to some complex logic. This is not a hypothetical situation. In a project big or small, the data you obtained at the first step is very likely far from ideal.

As a data scientist, you must be handy to format the data into the right shape to make your subsequent steps easier. In the following, you will learn how to slice and dice the dataset in pandas as well as reassemble them into a very different form to make the useful data more pronounced, so that analysis can be easier.

Let's get started.

Overview

This chapter is divided into two parts; they are:

- ▷ Segmenting and Concatenating: Choreographing with Pandas
- ▷ Pivoting and Merging: Dancing with Pandas

4.1 Segmenting and Concatenating: Choreographing with Pandas

One intriguing question you might pose is: How does the year a property was built influence its price? To investigate this, you can segment the dataset by "SalePrice" into four quartiles—Low, Medium, High, and Premium—and analyze the construction years within these segments. This methodical division of the dataset not only paves the way for a focused analysis but also reveals trends that might be concealed within a collective review.

Segmentation Strategy: Quartiles of "SalePrice"

Let's begin by creating a new column that neatly classifies the "SalePrice" of properties into your defined price categories:

```python
import pandas as pd

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Define the quartiles
quantiles = Ames['SalePrice'].quantile([0.25, 0.5, 0.75])

# Function to categorize each row
def categorize_by_price(row):
    if row['SalePrice'] <= quantiles.iloc[0]:
        return 'Low'
    elif row['SalePrice'] <= quantiles.iloc[1]:
        return 'Medium'
    elif row['SalePrice'] <= quantiles.iloc[2]:
        return 'High'
    else:
        return 'Premium'

# Apply the function to create a new column
Ames['Price_Category'] = Ames.apply(categorize_by_price, axis=1)
print(Ames[['SalePrice','Price_Category']])
```

Listing 4.1: Adding the price category column

By executing the above code, you have enriched your dataset with a new column entitled "`Price_Category`." Here's a glimpse of the output you've obtained:

```
      SalePrice Price_Category
0        126000            Low
1        139500         Medium
2        124900            Low
3        114000            Low
4        227000        Premium
...         ...            ...
2574     121000            Low
2575     139600         Medium
2576     145000         Medium
2577     217500        Premium
2578     215000        Premium

[2579 rows x 2 columns]
```

Output 4.1: Price and the corresponding category

Visualizing Trends with the Empirical Cumulative Distribution Function

The empirical cumulative distribution function (ECDF) is an increasing function range from 0 to 1. Below you will see how you can use ECDF to show the year a house was built is a distinctive feature for different price categories.

You can now split the original dataset into four DataFrames and proceed to visualize the cumulative distribution of construction years within each price category. This visual will

help you understand at a glance the historical trends in property construction as they relate to pricing.

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

def categorize_by_price(row):
    if row['SalePrice'] <= quantiles.iloc[0]:
        return 'Low'
    elif row['SalePrice'] <= quantiles.iloc[1]:
        return 'Medium'
    elif row['SalePrice'] <= quantiles.iloc[2]:
        return 'High'
    else:
        return 'Premium'

Ames = pd.read_csv('Ames.csv')
quantiles = Ames['SalePrice'].quantile([0.25, 0.5, 0.75])
Ames['Price_Category'] = Ames.apply(categorize_by_price, axis=1)

# Split original dataset into 4 DataFrames by Price Category
low_priced_homes = Ames.query('Price_Category == "Low"')
medium_priced_homes = Ames.query('Price_Category == "Medium"')
high_priced_homes = Ames.query('Price_Category == "High"')
premium_priced_homes = Ames.query('Price_Category == "Premium"')

# Setting the style for aesthetic looks
sns.set_style("whitegrid")

# Create a figure
plt.figure(figsize=(10, 6))

# Plot each ECDF on the same figure
sns.ecdfplot(data=low_priced_homes, x='YearBuilt', color='skyblue', label='Low')
sns.ecdfplot(data=medium_priced_homes, x='YearBuilt', color='orange', label='Medium')
sns.ecdfplot(data=high_priced_homes, x='YearBuilt', color='green', label='High')
sns.ecdfplot(data=premium_priced_homes, x='YearBuilt', color='red', label='Premium')

# Adding labels and title for clarity
plt.title('ECDF of Year Built by Price Category', fontsize=16)
plt.xlabel('Year Built', fontsize=14)
plt.ylabel('ECDF', fontsize=14)
plt.legend(title='Price Category', title_fontsize=14, fontsize=14)

# Show the plot
plt.show()
```

Listing 4.2: Creating the ECDF plot

Below is the ECDF plot, which provides a visual summation of the data you've categorized. An ECDF, or Empirical Cumulative Distribution Function, is a statistical tool used to describe the distribution of data points in a dataset. It represents the proportion or percentage of data points that fall below or at a certain value. Essentially, it gives you a way to visualize the

distribution of data points across different values, providing insights into the shape, spread, and central tendency of the data. ECDF plots are particularly useful because they allow for easy comparison between different datasets. Notice how the curves for each price category give you a narrative of housing trends over the years.

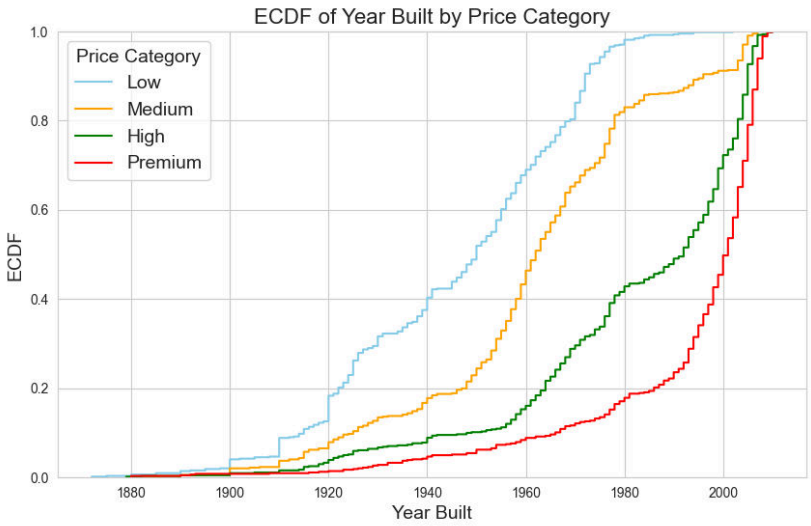

Figure 4.1: The ECDF plot

From the plot, it is evident that lower and medium-priced homes have a higher frequency of being built in earlier years, while high and premium-priced homes tend to be of more recent construction.

Stacking Datasets with Pandas.concat()

As data scientists, you often need to stack datasets or their segments to extract deeper insights. The `Pandas.concat()` function is your Swiss Army knife for such tasks, enabling you to combine DataFrames with precision and flexibility. This powerful function is similar to `UNION` operation in SQL when it comes to combining rows from different datasets. Yet, `Pandas.concat()` stands out by offering greater flexibility—it allows both vertical and horizontal concatenation of DataFrames. This feature becomes indispensable when datasets have non-matching columns but you need to align them by common columns. Here's how you can combine the segmented DataFrames to compare the broader market categories of "affordable" and "luxury" homes:

```
...
# Stacking Low and Medium categories into an "affordable_homes" DataFrame
affordable_homes = pd.concat([low_priced_homes, medium_priced_homes])

# Stacking High and Premium categories into a "luxury_homes" DataFrame
luxury_homes = pd.concat([high_priced_homes, premium_priced_homes])
```

Listing 4.3: Using `concat()` to create DataFrame

Through this, you can compare and analyze the characteristics that differentiate more accessible homes from their expensive counterparts. Should the two DataFrames you pass on to `concat()`

have different columns, the resulting DataFrame will have the superset of columns from both, while some rows filled in NaN for the columns that they have no data from the original DataFrame.

4.2 Pivoting and Merging: Dancing with Pandas

Having segmented the dataset into "affordable" and "luxury" homes and explored the distribution of their construction years, you now turn your attention to another dimension that influences property value: amenities, with a focus on the number of fireplaces. Before you delve into merging datasets—a task for which `Pandas.merge()` stands as a robust tool comparable to `JOIN` in SQL—you must first examine your data through a finer lens.

Pivot tables are an excellent tool for summarizing and analyzing specific data points within the segments. They provide you with the ability to aggregate data and reveal patterns that can inform your subsequent merge operations. By creating pivot tables, you can compile a clear and organized overview of the average living area and the count of homes, categorized by the number of fireplaces. This preliminary analysis will not only enrich your understanding of the two market segments but also set a solid foundation for the intricate merging techniques you want to show.

Creating Insightful Summaries with Pivot Tables

Let's commence by constructing pivot tables for the "affordable" and "luxury" home categories. These tables will summarize the average gross living area (GrLivArea) and provide a count of homes for each category of fireplaces present. Such analysis is crucial as it illustrates a key aspect of home desirability and value—the presence and number of fireplaces—and how these features vary across different segments of the housing market.

```
...
# Creating pivot tables with both mean living area and home count
aggfunc = {'GrLivArea': 'mean', 'Fireplaces': 'count'}
pivot_affordable = affordable_homes.pivot_table(index='Fireplaces', aggfunc=aggfunc)
pivot_luxury = luxury_homes.pivot_table(index='Fireplaces', aggfunc=aggfunc)

# Renaming columns and index labels separately
rename_rules = {'GrLivArea': 'AvLivArea', 'Fireplaces': 'HmCount'}

pivot_affordable.rename(columns=rename_rules, inplace=True)
pivot_affordable.index.name = 'Fire'

pivot_luxury.rename(columns=rename_rules, inplace=True)
pivot_luxury.index.name = 'Fire'

# View the pivot tables
print(pivot_affordable)
print(pivot_luxury)
```

Listing 4.4: Using pivot table to summarize the number of homes and the average area

With these pivot tables, you can now easily visualize and compare how features like fireplaces correlate with the living area and how frequently they occur within each segment. The first pivot table was crafted from the "affordable" homes DataFrame and demonstrates that most properties within this grouping do not have any fireplaces.

```
      HmCount    AvLivArea
Fire
0         931  1159.050483
1         323  1296.808050
2          38  1379.947368
```

Output 4.2: DataFrame of affordable homes

The second pivot table which was derived from the "luxury" homes DataFrame illustrates that properties within this subset have a range of zero to four fireplaces, with one fireplace being the most common.

```
      HmCount    AvLivArea
Fire
0         310  1560.987097
1         808  1805.243812
2         157  1998.248408
3          11  2088.090909
4           1  2646.000000
```

Output 4.3: DataFrame of luxury homes

With the creation of the pivot tables, you've distilled the data into a form that's ripe for the next analytical step—melding these insights using `Pandas.merge()` to see how these features interplay across the broader market.

The pivot table above is the simplest one. The more advanced version allows you to specify not only the index but also the columns in the argument. The idea is similar: you pick two columns, one specified as `index` and the other as `columns` argument, in which the values of these two columns are aggregated and become a matrix. The value in the matrix is then the result as specified by the `aggfunc` argument.

You can consider the following example, which produces a similar result as above:

```
...
pivot = Ames.pivot_table(index="Fireplaces",
                         columns="Price_Category",
                         aggfunc={'GrLivArea':'mean', 'Fireplaces':'count'})
print(pivot)
```

Listing 4.5: More advanced pivot table syntax

This prints:

	Fireplaces				GrLivArea			
Price_Category	High	Low	Medium	Premium	High	Low	Medium	Premium
Fireplaces								
0	228.0	520.0	411.0	82.0	1511.912281	1081.496154	1257.172749	1697.439024
1	357.0	116.0	207.0	451.0	1580.644258	1184.112069	1359.961353	1983.031042
2	52.0	9.0	29.0	105.0	1627.384615	1184.888889	1440.482759	2181.914286
3	5.0	NaN	NaN	6.0	1834.600000	NaN	NaN	2299.333333
4	NaN	NaN	NaN	1.0	NaN	NaN	NaN	2646.000000

Output 4.4: The number of homes and average living area by the number of fireplaces and price category

You can see the result is the same by comparing, for example, the count of low and medium homes of zero fireplaces to be 520 and 411, respectively, which 931 = 520+411 as you obtained previously. You see the second-level columns are labeled with Low, Medium, High, and Premium because you specified "`Price_Category`" as `columns` argument in `pivot_table()`. The dictionary to the `aggfunc` argument gives the top-level columns.

Towards Deeper Insights: Leveraging Pandas.merge() for Comparative Analysis

Having illuminated the relationship between fireplaces, home count, and living areas within the segmented datasets, you are well-positioned to take your analysis one step further. With `Pandas.merge()`, you can overlay these insights, akin to how `JOIN` operation in SQL combines records from two or more tables based on a related column. This technique will allow you to merge the segmented data on a common attribute, enabling a comparative analysis that transcends categorization.

You first operation uses an *outer join* to combine the affordable and luxury home datasets, ensuring no data is lost from either category. This method is particularly illuminating as it reveals the full spectrum of homes, regardless of whether they share a common number of fireplaces.

```
...
pivot_outer_join = pd.merge(pivot_affordable, pivot_luxury, on='Fire', how='outer',
                            suffixes=('_aff', '_lux')).fillna(0)
print(pivot_outer_join)
```

Listing 4.6: Outer join using `merge()` function

	HmCount_aff	AvLivArea_aff	HmCount_lux	AvLivArea_lux
Fire				
0	931.0	1159.050483	310	1560.987097
1	323.0	1296.808050	808	1805.243812
2	38.0	1379.947368	157	1998.248408
3	0.0	0.000000	11	2088.090909
4	0.0	0.000000	1	2646.000000

Output 4.5: Count of homes and average living area by number of fireplaces

In this case, the outer join functions similarly to a right join, capturing every distinct category of fireplaces present across both market segments. It is interesting to note that there are no properties within the affordable price range that have 3 or 4 fireplaces. You need to specify

two strings for the `suffixes` argument because the "`HmCount`" and "`AvLivArea`" columns exist in both DataFrames `pivot_affordable` and `pivot_luxury`. You see "`HmCount_aff`" is zero for 3 and 4 fireplaces because you need them as a placeholder for the outer join to match the rows in `pivot_luxury`.

Next, you can use the *inner join*, focusing on the intersection where affordable and luxury homes share the same number of fireplaces. This approach highlights the core similarities between the two segments.

```
...
pivot_inner_join = pd.merge(pivot_affordable, pivot_luxury, on='Fire', how='inner',
                            suffixes=('_aff', '_lux'))
print(pivot_inner_join)
```

Listing 4.7: Inner join using `merge()` function

```
      HmCount_aff  AvLivArea_aff  HmCount_lux  AvLivArea_lux
Fire
0             931    1159.050483          310    1560.987097
1             323    1296.808050          808    1805.243812
2              38    1379.947368          157    1998.248408
```

Output 4.6: Result of inner join shows fewer number of rows

Interestingly, in this context, the inner join mirrors the functionality of a left join, showcasing categories present in both datasets. You do not see the rows corresponding to 3 and 4 fireplaces because it is the result of an inner join, and there are no such rows in the DataFrame `pivot_affordable`.

Lastly, a *cross join* allows you to examine every possible combination of affordable and luxury home attributes, offering a comprehensive view of how different features interact across the entire dataset. The result is sometimes called the *Cartesian product* of rows from the two DataFrames.

```
...
# Resetting index to display cross join
pivot_affordable.reset_index(inplace=True)
pivot_luxury.reset_index(inplace=True)

pivot_cross_join = pd.merge(pivot_affordable, pivot_luxury, how='cross',
                            suffixes=('_aff', '_lux')).round(2)
print(pivot_cross_join)
```

Listing 4.8: Cross join using `merge()` function

The result is as follows, which demonstrates the result of cross-join but does not provide any special insight into the context of this dataset.

```
    Fire_aff  HmCount_aff  AvLivArea_aff  Fire_lux  HmCount_lux  AvLivArea_lux
0          0          931        1159.05         0          310        1560.99
1          0          931        1159.05         1          808        1805.24
2          0          931        1159.05         2          157        1998.25
3          0          931        1159.05         3           11        2088.09
```

```
4          0          931        1159.05          4            1        2646.00
5          1          323        1296.81          0          310        1560.99
6          1          323        1296.81          1          808        1805.24
7          1          323        1296.81          2          157        1998.25
8          1          323        1296.81          3           11        2088.09
9          1          323        1296.81          4            1        2646.00
10         2           38        1379.95          0          310        1560.99
11         2           38        1379.95          1          808        1805.24
12         2           38        1379.95          2          157        1998.25
13         2           38        1379.95          3           11        2088.09
14         2           38        1379.95          4            1        2646.00
```

Output 4.7: Result of cross join is a Cartesian product

Deriving Insights from Merged Data

Each join type demonstrated above sheds light on different aspects of the housing market:

- ▷ The *outer join* reveals the broadest range of properties, emphasizing the diversity in amenities like fireplaces across all price points.
- ▷ The *inner join* refines your view, focusing on the direct comparisons where affordable and luxury homes overlap in their fireplace counts, providing a clearer picture of standard market offerings.
- ▷ The *cross join* offers an exhaustive combination of features, you may need that to generate an exhaustive list of all possible conditions.

After conducting these merges, you observe *amongst affordable homes* that:

- ▷ Homes with no fireplaces have an average gross living area of approximately 1159 square feet and constitute the largest segment.
- ▷ As the number of fireplaces increases to one, the average living area expands to around 1296 square feet, underscoring a noticeable uptick in living space.
- ▷ Homes with two fireplaces, though fewer in number, boast an even larger average living area of approximately 1379 square feet, highlighting a trend where additional amenities correlate with more generous living spaces.

In contrast, you observe *amongst luxury homes* that:

- ▷ The luxury segment presents a starting point with homes without fireplaces averaging 1560 square feet, significantly larger than their affordable counterparts.
- ▷ The leap in the average living area is more pronounced in luxury homes as the number of fireplaces increases, with one-fireplace homes averaging about 1805 square feet.
- ▷ Homes with two fireplaces further amplify this trend, offering an average living area of nearly 1998 square feet. Homes with three to four fireplaces, although not many, have even larger living space, peaking at 2646 square feet.

These observations offer a fascinating glimpse into how amenities such as fireplaces not only add to the desirability of homes but also appear to be a marker of larger living spaces, particularly as you move from affordable to luxury market segments.

4.3 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

`concat()` method. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.concat.html

`DataFrame.pivot_table()` method. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot_table.html

`merge()` method. pandas.
https://pandas.pydata.org/docs/reference/api/pandas.merge.html

4.4 Summary

In this comprehensive exploration of data harmonization techniques using Python and Pandas, you've delved into the intricacies of segmenting, concatenating, pivoting, and merging datasets. From dividing datasets into meaningful segments based on price categories to visualizing trends in construction years, and from stacking datasets to analyzing broader market categories using `Pandas.concat()`, to summarizing and analyzing data points within segments using pivot tables, you've covered a wide array of essential data manipulation and analysis techniques. Additionally, by leveraging `Pandas.merge()` to compare segmented datasets and derive insights from different types of merge operations (outer, inner, cross), you've unlocked the power of data integration and exploration. Armed with these techniques, data scientists and analysts can navigate the complex landscape of data with confidence, uncovering hidden patterns, and extracting valuable insights that drive informed decision-making.

Specifically, you learned:

- How to divide datasets into meaningful segments based on price categories and visualize trends in construction years.
- The use of `Pandas.concat()` to stack datasets and analyze broader market categories.
- The role of pivot tables in summarizing and analyzing data points within segments.
- How to leverage `Pandas.merge()` to compare segmented datasets and derive insights from different types of merge operations (outer, inner, cross).

Starting from the next chapter, you will see examples of making a statement from data using data science techniques. The first would be the simplest one: Describe what you can see from the data from the most superficial aspect.

II From Data to Information

Decoding Data: An Introduction to Descriptive Statistics

5

You start your data science journey on the Ames dataset with descriptive statistics. The richness of the Ames housing dataset allows descriptive statistics to distill data into meaningful summaries. It is the initial step in analysis, offering a concise summary of the main aspects of a dataset. Their significance lies in simplifying complexity, aiding data exploration, facilitating comparative analysis, and enabling data-driven narratives.

As you delve into the Ames properties dataset, you'll explore the transformative power of descriptive statistics, distilling vast volumes of data into meaningful summaries. Along the way, you'll discover the nuances of key metrics and their interpretations, such as the implications of the average being greater than the median in terms of skewness.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ Fundamentals of Descriptive Statistics
- ▷ Data Dive with the Ames Dataset
- ▷ Visual Narratives

5.1 Fundamentals of Descriptive Statistics

This chapter will show you how to use descriptive statistics to make sense of data. Let's have a refresher on how statistics can help describe data.

Central Tendency: The Heart of the Data

Central tendency captures the core or typical value of the dataset. The most common measures include:

- ▷ **Mean (average):** The sum of all values divided by the number of values.
- ▷ **Median:** The middle value when the data is ordered.

▷ **Mode:** The value(s) that appear most frequently.

Dispersion: The Spread and Variability

Dispersion uncovers the spread and variability within the dataset. Key measures comprise:

▷ **Range:** Difference between the maximum and minimum values.

▷ **Variance:** Average of the squared differences from the mean.

▷ **Standard Deviation:** Square root of the variance.

▷ **Interquartile Range (IQR):** Range between the 25th and 75th percentiles.

Shape and Position: The Contour and Landmarks of Data

Shape and position reveal the distributional form of the data and critical markers, characterized by the following measures:

▷ **Skewness:** Asymmetry of the distribution. If the median is greater than the mean, we say the data is left-skewed (large values are more common). Conversely, it is right-skewed.

▷ **Kurtosis:** "Tailedness" of the distribution. In other words, how often you can see outliers. If you can see extremely large or extremely small values more often than normal distribution, you say the data is *leptokurtic*.

▷ **Percentiles:** Values below which a percentage of observations fall. The 25th, 50th, and 75th percentiles are also called the *quartiles*.

Descriptive Statistics gives voice to data, allowing it to tell its story succinctly and understandably.

5.2 Data Dive with the Ames Dataset

To delve into the Ames dataset, your spotlight is on the "SalePrice" attribute.

```
# Importing libraries and loading the dataset
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Descriptive Statistics of Sales Price
sales_price_description = Ames['SalePrice'].describe()
print(sales_price_description)
```

Listing 5.1: Descriptive statistics on one column

This summarizes "SalePrice," showcasing count, mean, standard deviation, and quantiles:

```
count 2579.000000
mean 178053.442420
std 75044.983207
min 12789.000000
```

```
25% 129950.000000
50% 159900.000000
75% 209750.000000
max 755000.000000
Name: SalePrice, dtype: float64
```

Output 5.1: The basic statistics of the "SalePrice" column

Furthermore, the following code:

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

median_saleprice = Ames['SalePrice'].median()
print("Median Sale Price:", median_saleprice)

mode_saleprice = Ames['SalePrice'].mode().values[0]
print("Mode Sale Price:", mode_saleprice)
```

Listing 5.2: Compute the median and mode

Shows the median and mode of the column:

```
Median Sale Price: 159900.0
Mode Sale Price: 135000
```

Output 5.2: The median and mode as computed

The average "SalePrice" (or mean) of homes in Ames is approximately $178,053.44. Meanwhile, the median price of $159,900 suggests that half the homes are sold below this value. The difference between these measures hints at high-value homes influencing the average, with the mode offering insights into the most frequent sale prices.

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

range_saleprice = Ames['SalePrice'].max() - Ames['SalePrice'].min()
print("Range of Sale Price:", range_saleprice)

variance_saleprice = Ames['SalePrice'].var()
print("Variance of Sale Price:", variance_saleprice)

std_dev_saleprice = Ames['SalePrice'].std()
print("Standard Deviation of Sale Price:", std_dev_saleprice)

iqr_saleprice = Ames['SalePrice'].quantile(0.75) - Ames['SalePrice'].quantile(0.25)
print("IQR of Sale Price:", iqr_saleprice)
```

Listing 5.3: Compute dispersion statistics

This shows:

```
Range of Sale Price: 742211
Variance of Sale Price: 5631749504.563301
Standard Deviation of Sale Price: 75044.98320716253
IQR of Sale Price: 79800.0
```

Output 5.3: Range, variance, standard deviation, and IQR as computed

The range of "SalePrice", spanning from \$12,789 to \$755,000, showcases the vast diversity in Ames' property values. With a variance of approximately 5.63×10^9 , the corresponding standard derivation is \$75,045, which is a substantial variability in prices. The interquartile range (IQR), representing the middle 50% of the data, stands at \$79,800, reflecting the spread of the central bulk of housing prices.

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

skewness_saleprice = Ames['SalePrice'].skew()
print("Skewness of Sale Price:", skewness_saleprice)

kurtosis_saleprice = Ames['SalePrice'].kurt()
print("Kurtosis of Sale Price:", kurtosis_saleprice)

tenth_percentile = Ames['SalePrice'].quantile(0.10)
ninetieth_percentile = Ames['SalePrice'].quantile(0.90)
print("10th Percentile:", tenth_percentile)
print("90th Percentile:", ninetieth_percentile)

q1_saleprice = Ames['SalePrice'].quantile(0.25)
q2_saleprice = Ames['SalePrice'].quantile(0.50)
q3_saleprice = Ames['SalePrice'].quantile(0.75)
print("Q1 (25th Percentile):", q1_saleprice)
print("Q2 (Median/50th Percentile):", q2_saleprice)
print("Q3 (75th Percentile):", q3_saleprice)
```

Listing 5.4: Computing the skewness, kurtosis, and percentiles

This shows:

```
Skewness of Sale Price: 1.7607507033716905
Kurtosis of Sale Price: 5.430410648673599
10th Percentile: 107500.0
90th Percentile: 272100.0000000001
Q1 (25th Percentile): 129950.0
Q2 (Median/50th Percentile): 159900.0
Q3 (75th Percentile): 209750.0
```

Output 5.4: The statistics as computed

The "SalePrice" in Ames displays a positive skewness of 1.76, indicative of a longer or fatter tail on the right side of the distribution. This skewness means the average sale price is influenced by a few but significantly high-priced properties, while the majority of homes are transacted at prices below this average. You can see why the majority is on the lower end by observing that the mean exceeds the median. The kurtosis value at 5.43 further accentuates these insights, suggesting potential outliers or extreme values that augment the heavier tails of the

distribution. Normal distribution should have both skewness and kurtosis at zero. Values above 1 or below −1 strongly suggest a deviation from normal distribution.

Delving deeper, the quartile values offer insights into the central tendencies of the data. With Q1 at $129,950 and Q3 at $209,750, these quartiles encapsulate the interquartile range, representing the middle 50% of the data. The difference Q3−Q1 is the central spread of prices, portraying one aspect of the pricing spectrum. Additionally, the 10th and 90th percentiles, positioned at $107,500 and $272,100, respectively, function as pivotal demarcations. These percentiles demarcate the boundaries within which 80% of the home prices reside, highlighting the expansive range in property valuations and accentuating the multifaceted nature of the Ames housing market.

5.3 Visual Narratives

Visualizations breathe life into data, narrating its story. Let's dive into the visual narrative of the "SalePrice" feature from the Ames dataset.

```
import pandas as pd
# Importing visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns

Ames = pd.read_csv('Ames.csv')

# Setting up the style
sns.set_style("whitegrid")

# Calculate Mean, Median, Mode for SalePrice
mean_saleprice = Ames['SalePrice'].mean()
median_saleprice = Ames['SalePrice'].median()
mode_saleprice = Ames['SalePrice'].mode().values[0]

# Plotting the histogram
plt.figure(figsize=(14, 7))
sns.histplot(x=Ames['SalePrice'], bins=30, kde=True, color="skyblue")
plt.axvline(mean_saleprice, color='r', linestyle='--',
            label=f"Mean: ${mean_saleprice:.2f}")
plt.axvline(median_saleprice, color='g', linestyle='-',
            label=f"Median: ${median_saleprice:.2f}")
plt.axvline(mode_saleprice, color='b', linestyle='-.',
            label=f"Mode: ${mode_saleprice:.2f}")

# Calculating skewness and kurtosis for SalePrice
skewness_saleprice = Ames['SalePrice'].skew()
kurtosis_saleprice = Ames['SalePrice'].kurt()

# Annotations for skewness and kurtosis
text = 'Skewness: {:.2f}\nKurtosis: {:.2f}' \
        .format(Ames['SalePrice'].skew(), Ames['SalePrice'].kurt())
plt.annotate(text, xy=(500000, 100), fontsize=14,
             bbox={"boxstyle": "round,pad=0.3",
```

```
                    "edgecolor": "black",
                    "facecolor": "aliceblue"})
plt.title('Histogram of Ames\' Housing Prices with KDE and Reference Lines')
plt.xlabel('Housing Prices')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

Listing 5.5: Visualizing a column as a histogram

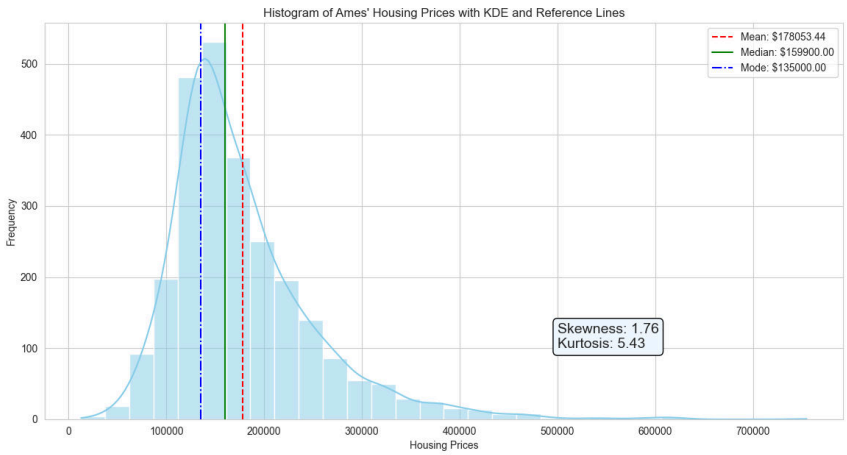

Figure 5.1: Histogram of the "SalePrice" column

The histogram above offers a compelling visual representation of Ames' housing prices. The pronounced peak near $150,000 underscores a significant concentration of homes within this particular price bracket. Complementing the histogram is the Kernel Density Estimation (KDE) curve, which provides a smoothed representation of the data distribution. The KDE is essentially an estimate of the histogram but with the advantage of infinitely narrow bins, offering a more continuous view of the data. It serves as a "limit" or refined version of the histogram, capturing nuances that might be missed in a discrete binning approach.

Notably, the rightward tail of KDE curve aligns with the positive skewness you previously computed, emphasizing a denser concentration of homes priced below the mean. The colored lines—red for mean, green for median, and blue for mode—act as pivotal markers, allowing for a quick comparison and understanding of the central tendencies of the distribution against the broader data landscape. Together, these visual elements provide a comprehensive insight into the distribution and characteristics of Ames' housing prices.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.lines import Line2D
Ames = pd.read_csv('Ames.csv')

# Horizontal box plot with annotations
plt.figure(figsize=(12, 8))
```

```
# Plotting the box plot with specified color and style
sns.boxplot(x=Ames['SalePrice'], color='skyblue', showmeans=True,
            meanprops={"marker": "D", "markerfacecolor": "red",
                       "markeredgecolor": "red", "markersize":10})

# Plotting arrows for Q1, Median and Q3
q1_saleprice = Ames['SalePrice'].quantile(0.25)
q2_saleprice = Ames['SalePrice'].quantile(0.50)
q3_saleprice = Ames['SalePrice'].quantile(0.75)
plt.annotate('Q1', xy=(q1_saleprice, 0.30), xytext=(q1_saleprice - 70000, 0.45),
             arrowprops={'edgecolor': 'black', 'arrowstyle': '->'}, fontsize=14)
plt.annotate('Q3', xy=(q3_saleprice, 0.30), xytext=(q3_saleprice + 20000, 0.45),
             arrowprops={'edgecolor': 'black', 'arrowstyle': '->'}, fontsize=14)
plt.annotate('Median', xy=(q2_saleprice, 0.20), xytext=(q2_saleprice - 90000, 0.05),
             arrowprops={'edgecolor': 'black', 'arrowstyle': '->'}, fontsize=14)

# Titles, labels, and legends
plt.title('Box Plot Ames\' Housing Prices', fontsize=16)
plt.xlabel('Housing Prices', fontsize=14)
plt.yticks([])  # Hide y-axis tick labels
plt.legend(handles=[Line2D([0], [0], marker='D', color='w', markerfacecolor='red',
           markersize=10, label='Mean')], loc='upper left', fontsize=14)

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

Listing 5.6: Visualizing the price in box plot

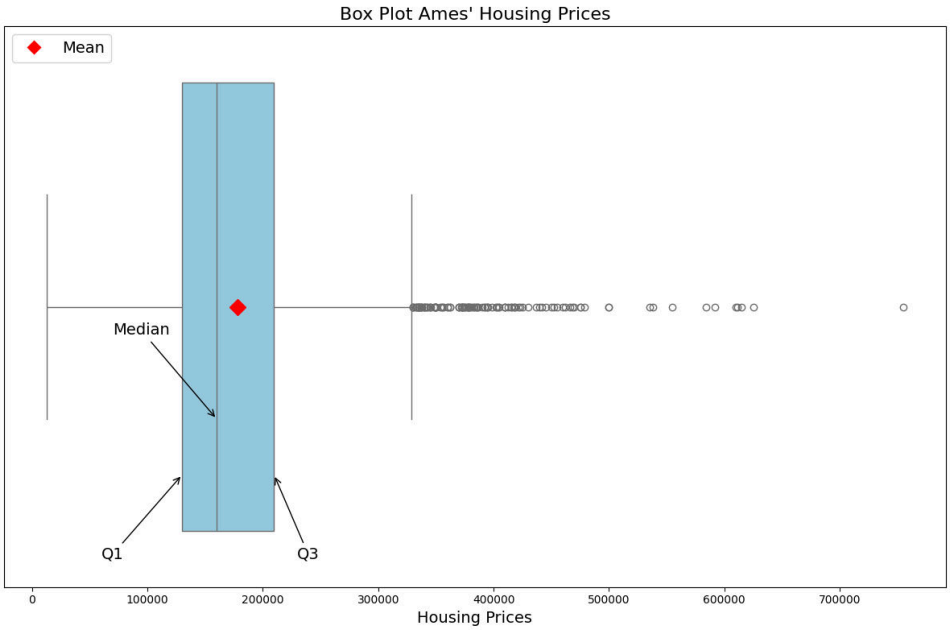

Figure 5.2: Box plot of the "SalePrice" column

The box plot provides a concise representation of central tendencies, ranges, and outliers, offering insights not readily depicted by the KDE curve or histogram. The interquartile range

(IQR), which spans from Q1 to Q3, captures the middle 50% of the data, providing a clear view of the central range of prices. Additionally, the positioning of the red diamond, representing the mean, to the right of the median emphasizes the influence of high-value properties on the average.

Central to interpreting the box plot are its "whiskers." The left whisker extends from the left edge of the box to the smallest data point within the lower fence, indicating prices that fall within 1.5 times the IQR below Q1. In contrast, the right whisker stretches from the right edge of the box to the largest data point within the upper fence, encompassing prices that lie within 1.5 times the IQR above Q3. These whiskers serve as boundaries that delineate the spread of data beyond the central 50%, with points lying outside them often flagged as potential outliers.

Outliers, depicted as individual points, spotlight exceptionally priced homes, potentially luxury properties, or those with distinct features. Recognizing and understanding these outliers is crucial, as they can highlight unique market dynamics or anomalies within the Ames housing market.

Visualizations like these are a gateway to understanding complex datasets. As you move forward, it's crucial to recognize and embrace the profound impact of visualization in data analysis—it has the unique ability to convey nuances and complexities that words or figures alone cannot capture.

5.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

Adam Hayes. *Descriptive Statistics: Definition, Overview, Types, Example*. Investopedia. Mar. 2024.
https://www.investopedia.com/terms/d/descriptive_statistics.asp

Density Estimation. Scikit-Learn User Guide.
https://scikit-learn.org/stable/modules/density.html

`kdeplot()` *API*. Seaborn.
https://seaborn.pydata.org/generated/seaborn.kdeplot.html

5.5 Summary

In this chapter, you delved into the Ames Housing dataset using Descriptive Statistics to uncover key insights about property sales. You computed and visualized essential statistical measures, emphasizing the value of central tendency, dispersion, and shape. By harnessing visual narratives and data analytics, you transformed raw data into compelling stories, revealing the intricacies and patterns of Ames' housing prices.

Specifically, you learned:

- ▷ How to utilize Descriptive Statistics to extract meaningful insights from the Ames Housing dataset, focusing on the "SalePrice" attribute.
- ▷ The significance of measures like mean, median, mode, range, and IQR, and how they narrate the story of housing prices in Ames.
- ▷ The power of visual narratives, particularly histograms and box plots, in visually representing and interpreting the distribution and variability of data.

Since you're working on a dataset that involves geographical locations, you will see how you can visualize them on a map in the next chapter.

From Data to Map: Visualizing Ames House Prices with Python

6

Geospatial visualization has become an essential tool for understanding and representing data in a geographical context. It plays a pivotal role in various real-world applications, from urban planning and environmental studies to real estate and transportation. For instance, city planners might use geospatial data to optimize public transportation routes, while real estate professionals could leverage it to analyze property value trends in specific regions. Using Python, you can harness the power of libraries like geopandas, Matplotlib, and contextily to create compelling visualizations. In this chapter, you'll dive deep into a code snippet that visualizes house sale prices in Ames, Iowa, breaking down each step to understand its purpose and functionality.

Let's get started.

Overview

This chapter is divided into five parts; they are:

- ▷ Installing Essential Python Packages
- ▷ Loading and Preparing the Data
- ▷ Setting the Coordinate Reference System (CRS)
- ▷ Creating a Convex Hull
- ▷ Visualizing the Data

6.1 Installing Essential Python Packages

Before you dive into the world of geospatial visualization with Python, it's crucial to setup your development environment correctly. On Windows, you can open either Command Prompt or PowerShell. If you're using macOS or Linux, the Terminal application is your gateway to the command-line world.

To install the essential packages, you can use the following commands on your terminal or command-line interface to get them from the Python Package Index (PyPI):

```
pip install pandas
pip install geopandas
pip install matplotlib
pip install contextily
pip install shapely
```

Listing 6.1: Installing essential Python packages

Once you've successfully installed the required packages, you're ready to import the necessary libraries and begin your geospatial visualization journey.

```
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
import contextily as ctx
from shapely.geometry import Point
```

Listing 6.2: Importing essential packages

You'll be using several Python libraries, including:

- ▷ `pandas`: For data manipulation and analysis.
- ▷ `geopandas`: To handle geospatial data.
- ▷ `matplotlib`: For creating static, animated, and interactive visualizations.
- ▷ `contextily`: To add basemaps to your plots.
- ▷ `shapely`: For manipulation and analysis of planar geometric objects.

6.2 Loading and Preparing the Data

The Ames dataset contains detailed information about house sales in Ames, Iowa. This includes various attributes of the houses, such as size, age, and condition, as well as their geographical coordinates (latitude and longitude). These geographical coordinates are crucial for your geospatial visualization, as they allow you to plot each house on a map, providing a spatial context to the sale prices.

```
# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Convert the DataFrame to a GeoDataFrame
geometry = [Point(xy) for xy in zip(Ames['Longitude'], Ames['Latitude'])]
geo_df = gpd.GeoDataFrame(Ames, geometry=geometry)
```

Listing 6.3: Converting a DataFrame into a GeoDataFrame

By converting the pandas DataFrame into a GeoDataFrame, you can leverage geospatial functionalities on your dataset, transforming the raw data into a format suitable for geospatial analysis and visualization.

6.3 Setting the Coordinate Reference System (CRS)

The Coordinate Reference System (CRS) is a fundamental aspect of accurate geospatial operations and cartography, determining how your data aligns with the surface of the Earth. The distance between two points will differ under a different CRS, and the map will look different. Below, you set the CRS for the GeoDataFrame using the notation "EPSG:4326," which corresponds to the widely-used WGS 84 (or World Geodetic System 1984) latitude-longitude coordinate system.

```
...
# Set the CRS for the GeoDataFrame
geo_df.crs = "EPSG:4326"
```

Listing 6.4: Setting the CRS on a GeoDataFrame

WGS 84 is a global reference system established in 1984 and is the de facto standard for satellite positioning, GPS, and various mapping applications. It uses a three-dimensional coordinate system with latitude and longitude defining positions on the surface of Earth and altitude indicating height above or below a *reference ellipsoid*.

Beyond WGS 84, numerous coordinate reference systems cater to diverse mapping needs. Choices include the Universal Transverse Mercator (UTM), providing planar, Cartesian coordinates suitable for regional mapping; the European Petroleum Survey Group (EPSG) options, such as "EPSG:3857" for web-based mapping; and the State Plane Coordinate System (SPCS), offering state-specific systems within the United States. Selecting an appropriate CRS depends on factors like scale, accuracy, and the geographic scope of your data, ensuring precision in geospatial analysis and visualization.

6.4 Creating a Convex Hull

A convex hull provides a boundary that encloses all data points, offering a visual representation of the geographical spread of your data.

```
# Create a convex hull around the points
convex_hull = geo_df.unary_union.convex_hull
convex_hull_geo = gpd.GeoSeries(convex_hull, crs="EPSG:4326")
convex_hull_transformed = convex_hull_geo.to_crs(epsg=3857)
buffered_hull = convex_hull_transformed.buffer(500)
```

Listing 6.5: Creating a convex hull from a GeoDataFrame

The transformation from EPSG:4326 (equirectangular projection) to EPSG:3857 (spherical projection) is crucial for a couple of reasons:

- ▷ *Web-based Visualizations*: The EPSG:3857 is optimized for web-based mapping applications like Google Maps and OpenStreetMap. By transforming your data to this CRS, you ensure it overlays correctly on web-based base maps.
- ▷ *Buffering in Meters*: The buffer operation adds a margin around the convex hull. In EPSG:4326, an area between two degrees of latitude and longitude is assumed to be

rectangular but it is not since the Earth is not flat. This makes buffering in meters problematic. By transforming to EPSG:3857, you can accurately buffer your convex hull by 500 meters, providing a clear boundary around Ames.

By buffering the convex hull, you not only visualize the spread of your data but also provide a geographical context to the visualization, emphasizing the region of interest.

6.5 Visualizing the Data

With your data prepared, it's time to bring it to life through visualization. You'll plot the sales prices of individual houses on a map, using a color gradient to represent different price ranges.

```
# Plotting the map with Sale Prices, a base map, and the buffered convex hull as a border
fig, ax = plt.subplots(figsize=(12, 8))
geo_df.set_crs(epsg=4326).to_crs(epsg=3857) \
      .plot(column='SalePrice', cmap='coolwarm', ax=ax, legend=True, markersize=20)
buffered_hull.boundary.plot(ax=ax, color='black', label='Buffered Boundary of Ames')
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron)
ax.set_axis_off()
ax.legend(loc='upper right')
colorbar = ax.get_figure().get_axes()[1]
colorbar.set_ylabel('Sale Price', rotation=270, labelpad=20, fontsize=15)
plt.title('Sales Prices of Individual Houses in Ames, Iowa with Buffered Boundary',
          fontsize=18)
plt.show()
```

Listing 6.6: Plot the price on the map

The color gradient used, "`coolwarm`," is a diverging colormap. This means it has two distinct colors representing the two ends of a spectrum, with a neutral color in the middle. In your visualization:

- ▷ *Cooler colors (blues)* represent houses with lower sale prices.
- ▷ *Warmer colors (reds)* signify houses with higher sale prices.

This choice of colormap allows readers to quickly identify areas with high and low property values, offering insights into the distribution of house sale prices in Ames. The buffered boundary further emphasizes the region of interest, providing context to the visualization.

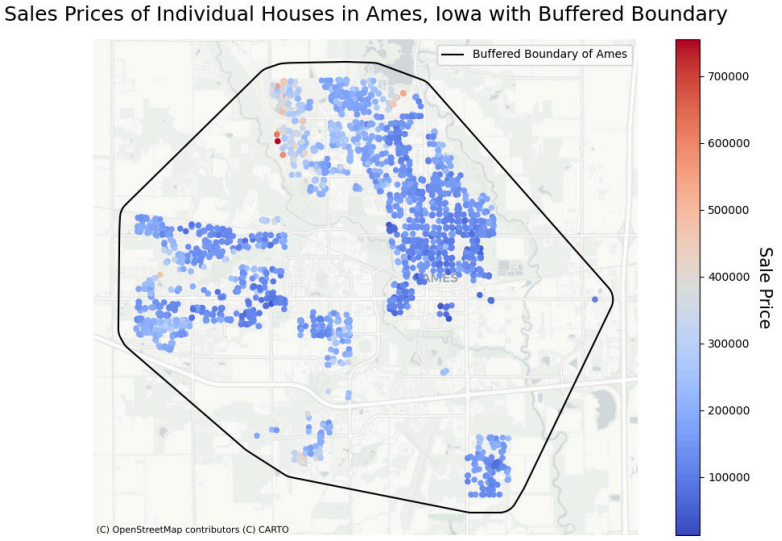

Figure 6.1: Price distribution according to the Ames dataset

This map is a combination of several components: The base map, brought in by `contextily` from OpenStreetMap, depicts the terrain at a particular latitude-longitude. The colored dots are based on the data from the pandas DataFrame but converted to a geographic CRS by geopandas, ensuring alignment with the base map.

Putting all things together, the following is the complete code:

```
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
import contextily as ctx
from shapely.geometry import Point

Ames = pd.read_csv('Ames.csv')

# Convert the DataFrame to a GeoDataFrame
geometry = [Point(xy) for xy in zip(Ames['Longitude'], Ames['Latitude'])]
geo_df = gpd.GeoDataFrame(Ames, geometry=geometry)

# Create a convex hull around the points
convex_hull = geo_df.unary_union.convex_hull
convex_hull_geo = gpd.GeoSeries(convex_hull, crs="EPSG:4326")
convex_hull_transformed = convex_hull_geo.to_crs(epsg=3857)
buffered_hull = convex_hull_transformed.buffer(500)

# Plotting the map with Sale Prices, a basemap, and the buffered convex hull as a border
fig, ax = plt.subplots(figsize=(12, 8))
geo_df.set_crs(epsg=4326).to_crs(epsg=3857) \
      .plot(column='SalePrice', cmap='coolwarm', ax=ax, legend=True, markersize=20)
buffered_hull.boundary.plot(ax=ax, color='black', label='Buffered Boundary of Ames')
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron)
```

```
ax.set_axis_off()
ax.legend(loc='upper right')
colorbar = ax.get_figure().get_axes()[1]
colorbar.set_ylabel('Sale Price', rotation=270, labelpad=20, fontsize=15)
plt.title('Sales Prices of Individual Houses in Ames, Iowa with Buffered Boundary',
          fontsize=18)
plt.show()
```

Listing 6.7: Showing homes in a map using geopandas

6.6 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

Introduction to GeoPandas.
https://geopandas.org/en/stable/getting_started/introduction.html

Leah Wasser. *Lesson 3. Coordinate Reference System and Spatial Projection.* The Earth analytics course. Earth Lab, Sept. 2019.
https://www.earthdatascience.org/courses/earth-analytics/spatial-data-r/intro-to-coordinate-reference-systems/

6.7 Summary

In this chapter, you delved into the intricacies of geospatial visualization using Python, focusing on the visualization of house sale prices in Ames, Iowa. Through a meticulous step-by-step breakdown of the code, you unveiled the various stages involved, from the initial data loading and preparation to the final visualization. Understanding geospatial visualization techniques goes beyond an academic exercise; it holds profound real-world implications. Mastery of these techniques can empower professionals across a spectrum of fields, from urban planning to real estate, enabling them to make informed, data-driven decisions rooted in geographical contexts. As cities grow and the world becomes increasingly data-centric, overlaying data on geographical maps will be indispensable in shaping future strategies and insights.

Specifically, from this chapter, you learned:

- ▷ How to harness essential Python libraries for geospatial visualization.
- ▷ The pivotal role of data preparation and transformation in geospatial operations.
- ▷ Effective techniques for visualizing geospatial data, including the nuances of setting up a color gradient and integrating a base map.

In the next chapter, you will learn how to evaluate the importance of features.

Feature Relationships 101: Lessons from the Ames Housing Data

7

In the realm of real estate, understanding the impact of property features on sale prices is crucial. In this exploration, you'll dive deep into the Ames Housing dataset, shedding light on the relationships between various features and their correlation with the sale price. By leveraging the power of data visualization, you can reveal patterns, trends, and insights that can assist a wide range of stakeholders, from homeowners to real estate developers.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ Unraveling Correlations
- ▷ Visualizing with Heatmaps
- ▷ Dissecting Feature Relationships through Scatter Plots

7.1 Unraveling Correlations

Correlation is a statistical measure that illustrates the extent to which two variables change together. A positive correlation indicates that as one variable increases, the other also tends to increase, and vice versa. Conversely, a negative correlation implies that as one variable increases, the other tends to decrease. Correlation close to 1 or -1 is a strong relationship, while correlation close to 0 is weak.

```
# Load the Dataset
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Calculate the correlation of all features with 'SalePrice'
# Set numeric_only=True to limit the output to numeric columns
correlations = Ames.corr(numeric_only=True)['SalePrice'].sort_values(ascending=False)

# Display the top 10 features most correlated with 'SalePrice'
```

```
top_correlations = correlations[1:11]
print(top_correlations)
```

Listing 7.1: Computing correlation

This prints:

```
OverallQual     0.790661
GrLivArea       0.719980
TotalBsmtSF     0.652268
1stFlrSF        0.642623
GarageCars      0.639017
GarageArea      0.635029
YearBuilt       0.544569
FullBath        0.535175
GarageYrBlt     0.521105
YearRemodAdd    0.514720
Name: SalePrice, dtype: float64
```

Output 7.1: Columns that strongly correlated to "SalePrice"

From the Ames Housing dataset, the top features most correlated with housing prices are:

- ▷ `OverallQual`: Overall Quality of the house, rated on a scale from 1 (worst) to 10 (best).
- ▷ `GrLivArea`: Above Ground Living Area, measured in square feet. It encompasses the living area that is not in the basement.
- ▷ `TotalBsmtSF`: Total Basement Area, represented in square feet. This combines both the finished and unfinished areas of the basement.
- ▷ `1stFlrSF`: First Floor Square Feet, indicating the size of the first floor of the house.
- ▷ `GarageCars`: Size of Garage in terms of car capacity. This represents the number of cars that can fit into the garage.
- ▷ `GarageArea`: Size of Garage, measured in square feet. It gives a sense of the total area covered by the garage.
- ▷ `YearBuilt`: Original Construction Date, indicating the year when the primary construction of the house was completed.
- ▷ `FullBath`: Full Bathrooms Above Grade. This counts the number of full bathrooms (i.e., with a sink, toilet, and either a tub or shower) that are not in the basement.
- ▷ `GarageYrBlt`: Year Garage was Built. This specifies the year the garage was constructed. For houses without a garage, this feature can be null.
- ▷ `YearRemodAdd`: Remodel Date. It indicates the year of remodeling or addition, with the same year as construction if no remodeling or additions.

Features most correlated are the features with the strongest predictive power. If you build a model to predict housing prices, these are the subset of input features with a high possibility of success. Correlated features may also be caused by some other common factor, which itself is a topic in data science that you would like to investigate and elaborate on.

The code above prints `correlations[1:11]` because `correlations[0]` is the `SalePrice` column, which by definition is 1.0. From a feature selection perspective, you should also check `correlations[-10:]` for the most negatively correlated features, which may also be powerful in explaining the prices. There are no features negatively correlated to the price in this particular dataset.

7.2 Visualizing with Heatmaps

Heatmaps provide a powerful visual tool to represent data in a two-dimensional space, with colors indicating magnitudes or frequencies. In the context of correlations, a heatmap can beautifully illustrate the strength and direction of relationships between multiple features. Let's dive into a heatmap showcasing the correlations among the top features most correlated with `SalePrice`.

```
...
import seaborn as sns
import matplotlib.pyplot as plt

# Select the top correlated features including SalePrice
selected_features = list(top_correlations.index) + ['SalePrice']

# Compute the correlations for the selected features
correlation_matrix = Ames[selected_features].corr()

# Setup the matplotlib figure
plt.figure(figsize=(12, 8))

# Generate a heatmap
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", linewidths=.5,
            fmt=".2f", vmin=-1, vmax=1)
plt.title("Heatmap of Correlations among Top Features with SalePrice", fontsize=16)
plt.show()
```

Listing 7.2: Showing correlation with a heatmap

This is the heatmap created with the code above:

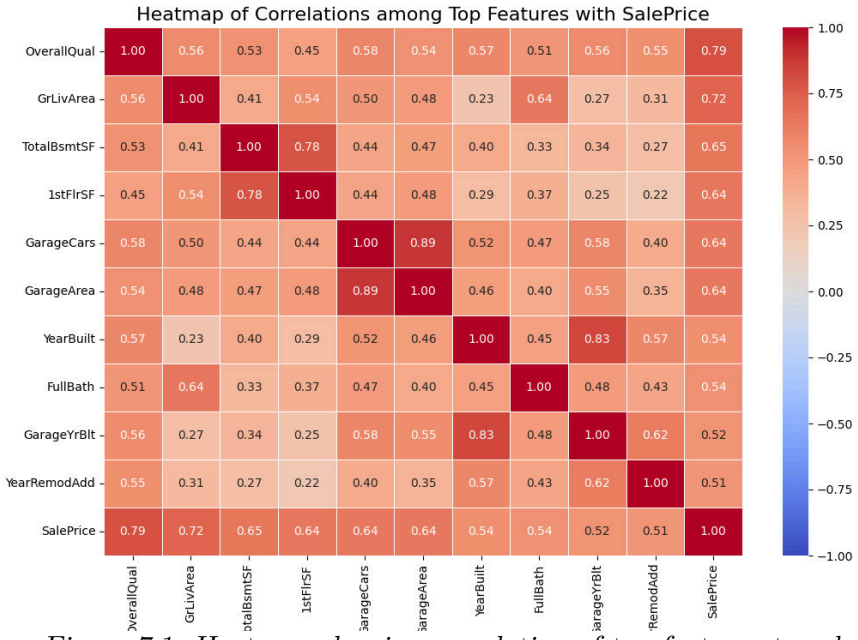

Figure 7.1: Heatmap showing correlation of top features to sales price

Heatmaps are a fantastic way to visualize the strength and direction of relationships between multiple variables simultaneously. The color intensity in each cell of the heatmap corresponds to the magnitude of the correlation, with warmer colors representing positive correlations and cooler colors indicating negative correlations. There is no blue in the heatmap above because only the 10 most positively correlated columns are concerned.

In the heatmap above, you can observe the following:

- ▷ `OverallQual`, representing the overall quality of the house, has the strongest positive correlation with `SalePrice`, with a correlation coefficient of approximately 0.79. This implies that as the quality of the house increases, the sale price also tends to increase.
- ▷ `GrLivArea` and `TotalBsmtSF`, representing the above-ground living area and total basement area respectively, also show strong positive correlations with the sale price.
- ▷ Most of the features have a positive correlation with `SalePrice`, which indicates that as these features increase or improve, the sale price of the house also tends to go up.
- ▷ It's worth noting some features are correlated with each other. For example, `GarageCars` and `GarageArea` are strongly correlated, which makes sense as a larger garage can accommodate more cars.

Such insights can be invaluable for various stakeholders in the real estate sector. For instance, real estate developers can focus on improving specific features in homes to increase their market value.

Below is the complete code:

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

# Load the Dataset
Ames = pd.read_csv('Ames.csv')

# Calculate the top 10 features most correlated with 'SalePrice'
correlations = Ames.corr(numeric_only=True)['SalePrice'].sort_values(ascending=False)
top_correlations = correlations[1:11]

# Select the top correlated features including SalePrice
selected_features = list(top_correlations.index) + ['SalePrice']

# Compute the correlations for the selected features
correlation_matrix = Ames[selected_features].corr()

# Set up the matplotlib figure
plt.figure(figsize=(12, 8))

# Generate a heatmap
sns.heatmap(correlation_matrix, annot=True,
            cmap="coolwarm", linewidths=.5, fmt=".2f", vmin=-1, vmax=1)

# Title
plt.title("Heatmap of Correlations among Top Features with SalePrice", fontsize=16)

# Show the heatmap
plt.show()
```

Listing 7.3: Compute the correlation and visualize in a heatmap

7.3 Dissecting Feature Relationships through Scatter Plots

While correlations provide a preliminary understanding of relationships, it's crucial to visualize these relationships further. Scatter plots, for instance, can show the relationship between two features in the form of a distribution of points.

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

Ames = pd.read_csv('Ames.csv')

# Setting up the figure and axes
fig, ax = plt.subplots(2, 2, figsize=(15, 12))

# Scatter plot for SalePrice vs. OverallQual
sns.scatterplot(x=Ames['OverallQual'], y=Ames['SalePrice'], ax=ax[0, 0],
                color='blue', alpha=0.6)
ax[0, 0].set_title('House Prices vs. Overall Quality')
```

```
ax[0, 0].set_ylabel('House Prices')
ax[0, 0].set_xlabel('Overall Quality')

# Scatter plot for SalePrice vs. GrLivArea
sns.scatterplot(x=Ames['GrLivArea'], y=Ames['SalePrice'], ax=ax[0, 1],
                color='red', alpha=0.6)
ax[0, 1].set_title('House Prices vs. Ground Living Area')
ax[0, 1].set_ylabel('House Prices')
ax[0, 1].set_xlabel('Above Ground Living Area (sq. ft.)')

# Scatter plot for SalePrice vs. TotalBsmtSF
sns.scatterplot(x=Ames['TotalBsmtSF'], y=Ames['SalePrice'], ax=ax[1, 0],
                color='green', alpha=0.6)
ax[1, 0].set_title('House Prices vs. Total Basement Area')
ax[1, 0].set_ylabel('House Prices')
ax[1, 0].set_xlabel('Total Basement Area (sq. ft.)')

# Scatter plot for SalePrice vs. 1stFlrSF
sns.scatterplot(x=Ames['1stFlrSF'], y=Ames['SalePrice'], ax=ax[1, 1],
                color='purple', alpha=0.6)
ax[1, 1].set_title('House Prices vs. First Floor Area')
ax[1, 1].set_ylabel('House Prices')
ax[1, 1].set_xlabel('First Floor Area (sq. ft.)')

# Adjust layout
plt.tight_layout(pad=3.0)
plt.show()
```

Listing 7.4: Showing relationships of columns in scatter plots

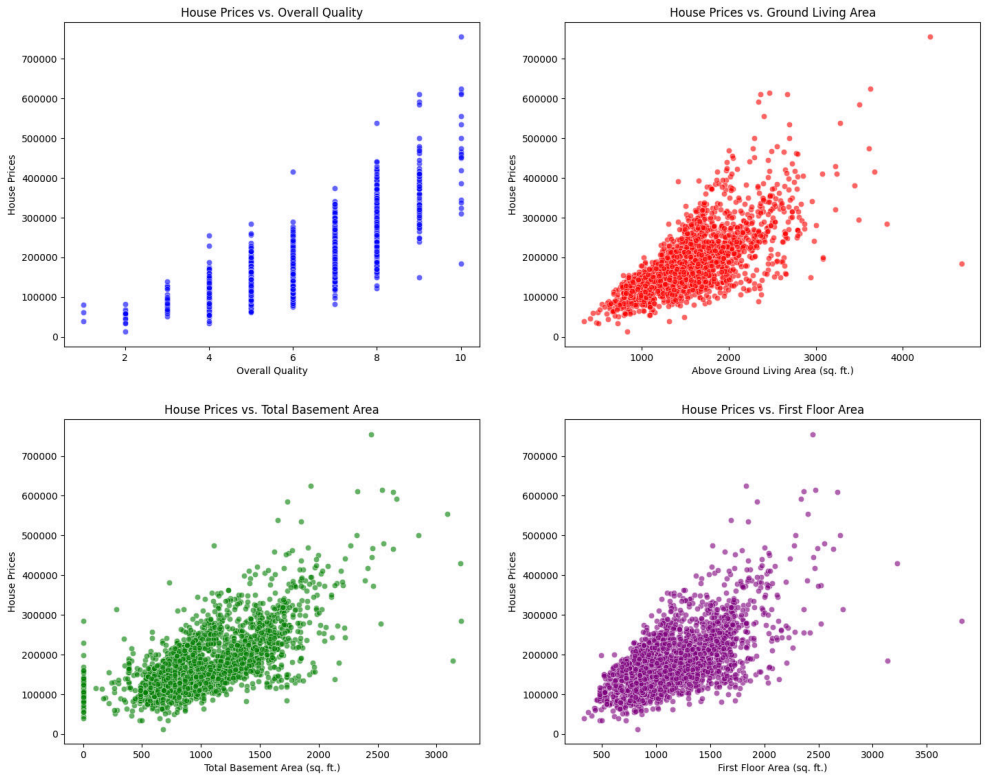

Figure 7.2: Relationship of some columns

The scatter plots emphasize the strong positive relationships between sale price and key features. As the overall quality, ground living area, basement area, and first-floor area increase, houses generally fetch higher prices. However, some exceptions and outliers suggest that other factors also influence the final sale price. One particular example is from the "House Prices vs. Ground Living Area" scatter plot above: At 2500 sq. ft. and above, the dots are dispersed, suggesting that there is a wide range in the house price in which the area is not strongly correlated or not effectively explained.

Lastly, it's essential to discern between correlation and causation. A high correlation does not necessarily imply that one variable causes changes in another. It merely indicates a relationship.

7.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

Pearson correlation coefficient. Wikipedia.
https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

Adam Hayes. *Correlation: What It Means in Finance and the Formula for Calculating It*. Investopedia. July 2023.
https://www.investopedia.com/terms/c/correlation.asp

7.5 Summary

In exploring the Ames Housing dataset, you embarked on a journey to understand the relationships between various property features and their correlation with sale prices. Through heatmaps and scatter plots, you unveiled patterns and insights that can significantly impact real estate stakeholders.

Specifically, you learned:

- The importance of correlation and its significance in understanding relationships between property features and sale prices.
- The utility of heatmaps in visually representing correlations among multiple features.
- The insight provided by scatter plots, emphasizes the feature dynamics beyond mere correlation coefficients.

In the next chapter, you will learn another visualization technique that builds on scatter plots.

Mastering Pair Plots for Visualization and Hypothesis Creation

8

Understanding real estate data involves exploring different property features and their impact on housing market trends. One useful tool for exploring these relationships is the pair plot. This data visualization technique allows you to discover the direction and magnitude of correlations among different features within the dataset.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ Exploring Feature Relationships with Pair Plots
- ▷ Unveiling Deeper Insights: Pair Plots with Categorical Enhancement
- ▷ Inspiring Data-Driven Inquiries: Hypothesis Generation Through Pair Plots

8.1 Exploring Feature Relationships with Pair Plots

A pair plot, also known as a scatter plot matrix, provides a comprehensive view of the interplay between multiple variables in a dataset. Unlike correlation heatmaps, which represent correlation coefficients in a color-coded grid, pair plots depict the actual data points, revealing the dynamic (such as nonlinearity) beyond just their strength and direction.

To illustrate this, let's delve into the Ames Housing dataset. You'll focus on the top five features most strongly correlated with "SalePrice."

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

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Calculate the correlation of all features with 'SalePrice'
correlations = Ames.corr(numeric_only=True)['SalePrice'].sort_values(ascending=False)
```

```
# Top 5 features most correlated with 'SalePrice' (excluding 'SalePrice' itself)
top_5_features = correlations.index[1:6]

# Creating the pair plot for these features and 'SalePrice'
# Adjust the size by setting height and aspect
sns.pairplot(Ames, vars=['SalePrice'] + list(top_5_features), height=1.35, aspect=1.85)

# Displaying the plot
plt.show()
```

Listing 8.1: Creating pair plot

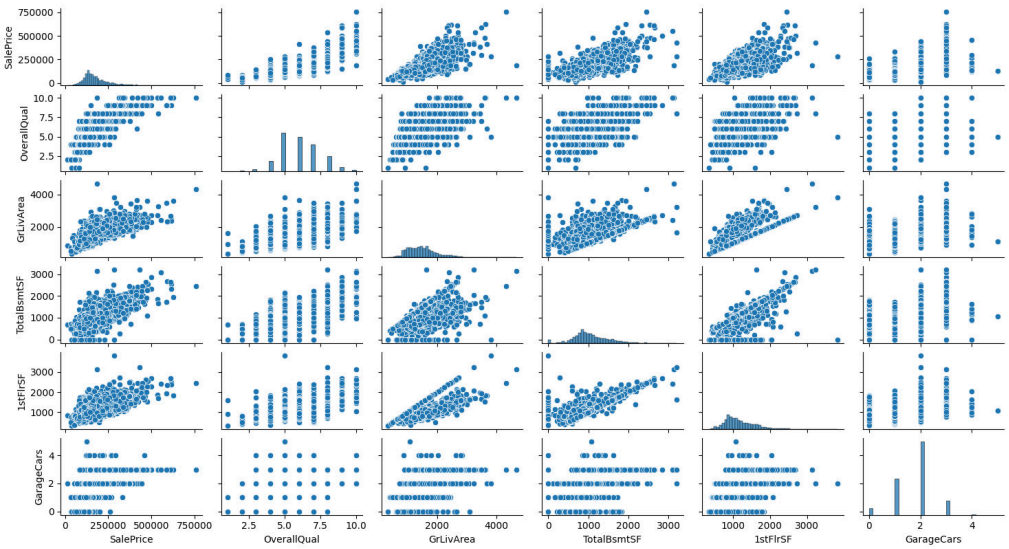

Figure 8.1: Pair plot showing the correlation of top five features with "SalesPrice" column

As seen in the pair plot above, each subplot provides a scatter plot for a pair of features. This visualization method not only allows you to observe the distribution of individual variables but also reveals the intricate relationships between them. The pair plot is particularly adept at uncovering the nature of these relationships. For example, you can see whether the relationships are *linear*, suggesting a steady increase or decrease, or *nonlinear*, indicating more complex dynamics. It also highlights clusters where data points are grouped and outliers that stand apart from the general trend.

Take, for instance, the relationship between "SalePrice" and "GrLivArea." The scatter plot in the pair plot shows a broadly linear relationship, indicating that as "GrLivArea" increases, so does "SalePrice." However, it's not a perfect linear correlation—some data points deviate from this trend, suggesting other factors may also influence the sale price. Moreover, the plot reveals a few outliers, properties with exceptionally high "GrLivArea" or "SalePrice," that could be unique cases or potential data entry errors.

By presenting data in this format, pair plots go beyond mere numerical coefficients, offering a nuanced and detailed view of the data. They enable you to identify patterns, trends, and exceptions within the dataset, which are vital for making informed decisions in

the real estate market. Such insights are especially beneficial for stakeholders looking to understand the multifaceted nature of property value determinants.

8.2 Unveiling Deeper Insights: Pair Plots with Categorical Enhancement

In the continued exploration of real estate data visualization, you now focus on enriching your pair plots with categorical variables. By incorporating a categorical dimension, you can uncover deeper insights and more nuanced relationships within the data. In this section, you transform "LotShape" from the Ames housing dataset into a binary category (Regular vs. Irregular) and integrate it into your pair plot. This enhancement allows you to observe how these lot shapes interact with key variables like "SalePrice", "OverallQual", and "GrLivArea."

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

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Convert 'LotShape' to a binary feature: 'Regular' and 'Irregular'
Ames['LotShape_Binary'] = \
    Ames['LotShape'].apply(lambda x: 'Regular' if x == 'Reg' else 'Irregular')

# Creating the pair plot, color-coded by 'LotShape_Binary'
sns.pairplot(Ames, vars=['SalePrice', 'OverallQual', 'GrLivArea'], hue='LotShape_Binary',
             palette='Set1', height=2.5, aspect=1.75)

# Display the plot
plt.show()
```

Listing 8.2: Creating pair plot with color-coded category

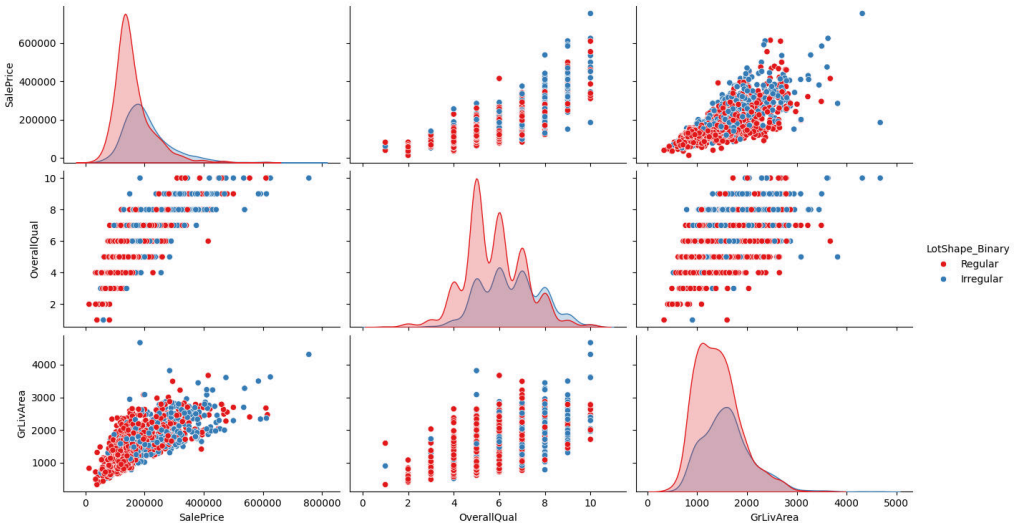

Figure 8.2: Pair plot on three columns with categorical enhancement

The resulting pair plot, color-coded for "Regular" and "Irregular" lot shapes, reveals intriguing patterns. For instance, you should notice that homes with irregular lot shapes tend to have a varied range of sale prices and living areas, potentially indicating a diversity in property types or buyer preferences. Additionally, the KDE plot at the center shows the overall quality ("OverallQual") is spikier, i.e., with excess kurtosis or *leptokurtic*, for regular lots and flatter for irregular lots. This means it is less variable for regular lots, suggesting a possible trend in construction standards or design choices in these areas.

This enhanced visual tool not only deepens your understanding of the housing market dynamics but also invites further exploration. Stakeholders can experiment with different feature combinations and categorical variables to tailor their analysis to specific interests or market segments, making this approach a versatile asset in real estate analytics.

8.3 Inspiring Data-Driven Inquiries: Hypothesis Generation Through Pair Plots

Pair plots serve as a powerful tool not just for visualization but also for hypothesis generation in data analysis. By revealing patterns, trends, and anomalies in a dataset, these plots can inspire insightful questions and hypotheses. For instance, observing a linear relationship between two variables might lead to a hypothesis about a causal connection, or an unexpected cluster of data points could prompt questions about what factors are contributing to that crowd. Essentially, pair plots can act as a springboard for deeper, more targeted statistical testing and exploration.

Hypotheses From the First Visual (Relationships between "SalePrice" and other features):

- ▷ Hypothesis 1: There is a linear relationship between "GrLivArea" and "SalePrice," suggesting that larger living areas directly contribute to higher property values.

- ▷ Hypothesis 2: Outliers observed in the "SalePrice" versus "GrLivArea" plot may indicate data entry errors, cheap large houses, or expensive small houses in the dataset. These warrant further investigation.

Hypotheses From the Second Visual (Incorporating "LotShape" as a binary category):

- ▷ Hypothesis 3: Properties with irregular lot shapes have a wider variance in sale prices than regular lot shapes, possibly due to a higher diversity in property types or unique features.
- ▷ Hypothesis 4: The overall quality of properties on regular-shaped lots tends to be more consistent, suggesting standardized construction practices or buyer preferences in these areas.

These hypotheses, derived from the observed patterns in the pair plots, can then be tested through more rigorous statistical methods to validate or refute the initial observations. This approach underscores the utility of pair plots as a foundational step in hypothesis-driven data analysis.

8.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

`pairplot()` *API*. Seaborn.
https://seaborn.pydata.org/generated/seaborn.pairplot.html
Yan Holtz. *Correlogram*. From Data to Viz.
https://www.data-to-viz.com/graph/correlogram.html

8.5 Summary

In your exploration of the Ames Housing dataset, you have explored the world of pair plots, uncovering the stories within the data. This journey has not only highlighted the importance of visual analysis in real estate analytics but also demonstrated the power of pair plots in revealing complex relationships and guiding data-driven hypothesis generation. Specifically, you learned:

- ▷ The effectiveness of pair plots in illustrating the relationships between various housing market features, especially about "SalePrice."
- ▷ How the integration of categorical variables like "LotShape" into pair plots can provide deeper insights and reveal subtler trends in the data.
- ▷ The potential of pair plots as a foundation for generating hypotheses, setting the stage for more advanced statistical analyses and informed decision-making.

In the next chapter, you will learn how to numerically derive a conclusion rather than rely on your interpretation of graphics.

Inferential Insights: How Confidence Intervals Illuminate the Ames Real Estate Market

9

In the vast universe of data, it's not always about what you can see but rather what you can infer. Confidence intervals, a cornerstone of inferential statistics, empower you to make educated guesses about a larger population based on your sample data. Using the Ames Housing dataset, let's unravel the concept of confidence intervals and see how they can provide actionable insights into the real estate market.

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ▷ The Core of Inferential Statistics
- ▷ What are Confidence Intervals?
- ▷ Estimating Sales Prices with Confidence Intervals
- ▷ Understanding the Assumptions Behind

9.1 The Core of Inferential Statistics

Inferential statistics uses a sample of data to make inferences about the population from which it was drawn. The main components include:

- ▷ *Confidence Intervals*: Range within which a population parameter is likely to lie.
- ▷ *Hypothesis Testing*: Process of making inferences about population parameters.

Inferential statistics is indispensable when it is impractical to study the entire population, but insights need to be derived from a representative sample. You will demonstrate this with the Ames housing dataset.

9.2 What are Confidence Intervals?

Imagine you've taken a random sample of houses from a city and calculated the average sales price. While this gives you a single estimate, wouldn't it be more informative to have a range in which the true average sales price for the entire city likely falls? This range estimate is what a confidence interval provides. In essence, a confidence interval gives you a range of values within which you can be reasonably sure (e.g., 95% confident) that the true population parameter (like the mean or proportion) lies.

9.3 Estimating Sales Prices with Confidence Intervals

While point estimates like means and medians give you an idea about central tendency, they don't inform you about the range in which the true population parameter might lie. Confidence intervals bridge this gap. For instance, if you want to estimate the mean sales price of all houses in Ames, you can use the dataset to compute a 95% confidence interval for the mean sales price. This interval will give you a range in which you can be 95% confident that the true mean sales price of all houses in Ames lies.

You will use the t-distribution to find the confidence interval:

```
import scipy.stats as stats
import pandas as pd
Ames = pd.read_csv('Ames.csv')

#Define the confidence level and degrees of freedom
confidence_level = 0.95
degrees_freedom = Ames['SalePrice'].count() - 1

#Calculate the confidence interval for 'SalePrice'
confidence_interval = stats.t.interval(confidence_level, degrees_freedom,
                                       loc=Ames['SalePrice'].mean(),
                                       scale=Ames['SalePrice'].sem())

# Print out the sentence with the confidence interval figures
print(f"The 95% confidence interval for the true mean sales price of all houses in Ames "
      f"is between ${confidence_interval[0]:.2f} and ${confidence_interval[1]:.2f}.")
```

Listing 9.1: Confidence interval based on t-distribution

This prints:

```
The 95% confidence interval for the true mean sales price of all houses in Ames
is between $175155.78 and $180951.11.
```

Output 9.1: Confidence interval

Confidence intervals provide a range that, with a certain level of confidence, is believed to encompass the true population parameter. Interpreting this range allows you to understand the variability and precision of your estimate. If the 95% confidence interval for the mean "SalePrice" is (175156, 180951), you can be 95% confident that the true mean sales price for all properties in Ames lies between $175,156 and $180,951.

```
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
Ames = pd.read_csv('Ames.csv')

confidence_level = 0.95
degrees_freedom = Ames['SalePrice'].count() - 1
confidence_interval = stats.t.interval(confidence_level, degrees_freedom,
                                       loc=Ames['SalePrice'].mean(),
                                       scale=Ames['SalePrice'].sem())

# Plot the main histogram
plt.figure(figsize=(10, 7))
plt.hist(Ames['SalePrice'], bins=30, color='lightblue', edgecolor='black', alpha=0.5,
         label='Sales Prices Distribution')

# Vertical lines for sample mean and confidence interval with adjusted styles
plt.axvline(Ames['SalePrice'].mean(), color='blue', linestyle='-',
            label=f'Mean: ${Ames["SalePrice"].mean():,.2f}')
plt.axvline(confidence_interval[0], color='red', linestyle='--',
            label=f'Lower 95% CI: ${confidence_interval[0]:,.2f}')
plt.axvline(confidence_interval[1], color='green', linestyle='--',
            label=f'Upper 95% CI: ${confidence_interval[1]:,.2f}')

# Annotations and labels
plt.title('Distribution of Sales Prices with Confidence Interval', fontsize=20)
plt.xlabel('Sales Price', fontsize=16)
plt.ylabel('Frequency', fontsize=16)
plt.xlim([min(Ames['SalePrice']) - 5000, max(Ames['SalePrice']) + 5000])
plt.legend()
plt.grid(axis='y')
plt.show()
```

Listing 9.2: Plotting the sales price distribution with confidence interval

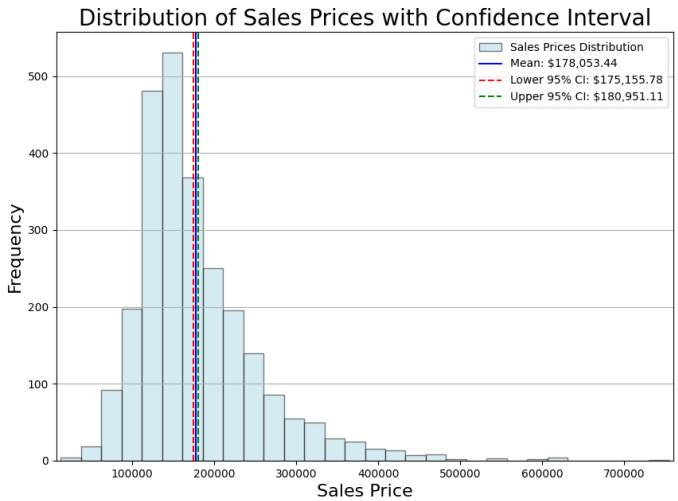

Figure 9.1: Distribution of sales prices and the mean

In the visual above, the histogram represents the distribution of sales prices in the Ames Housing dataset. The blue vertical line corresponds to the sample mean, providing a point estimate for the average sales price. The dashed red and green lines represent the 95% lower and upper confidence intervals, respectively.

Let's delve deeper into the price range between $150,000 and $200,000.

```
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
Ames = pd.read_csv('Ames.csv')

confidence_level = 0.95
degrees_freedom = Ames['SalePrice'].count() - 1
confidence_interval = stats.t.interval(confidence_level, degrees_freedom,
                                       loc=Ames['SalePrice'].mean(),
                                       scale=Ames['SalePrice'].sem())

# Creating a second plot focused on the mean and confidence intervals
plt.figure(figsize=(10, 7))
plt.hist(Ames['SalePrice'], bins=30, color='lightblue', edgecolor='black', alpha=0.5,
         label='Sales Prices')

# Zooming in around the mean and confidence intervals
plt.xlim([150000, 200000])

# Vertical lines for sample mean and confidence interval with adjusted styles
plt.axvline(Ames['SalePrice'].mean(), color='blue', linestyle='-',
            label=f'Mean: ${Ames["SalePrice"].mean():,.2f}')
plt.axvline(confidence_interval[0], color='red', linestyle='--',
            label=f'Lower 95% CI: ${confidence_interval[0]:,.2f}')
plt.axvline(confidence_interval[1], color='green', linestyle='--',
            label=f'Upper 95% CI: ${confidence_interval[1]:,.2f}')

# Annotations and labels for the zoomed-in plot
plt.title('Zoomed-in View of Mean and Confidence Intervals', fontsize=20)
plt.xlabel('Sales Price', fontsize=16)
plt.ylabel('Frequency', fontsize=16)
plt.legend()
plt.grid(axis='y')
plt.show()
```

Listing 9.3: Plotting the sales price distribution zoomed in to the mean

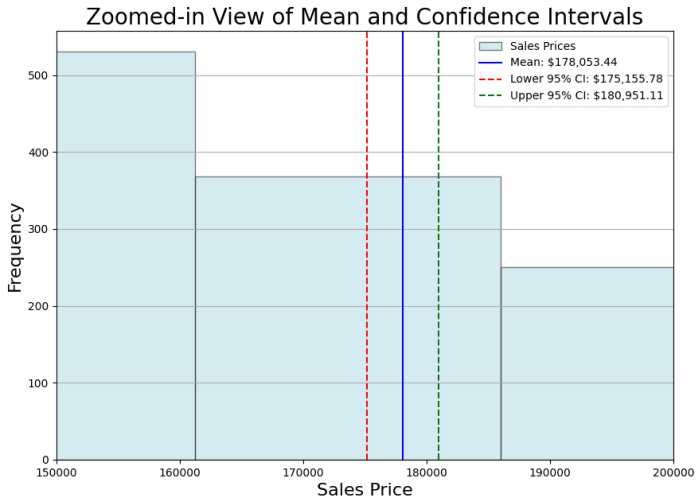

Figure 9.2: Distribution of sales price zoomed in to the mean

The confidence interval is interpreted as follows: You are 95% confident that the true mean sales price for all houses in Ames lies between the lower confidence level of $175,156 and the upper confidence level of $180,951. This range accounts for the inherent variability in estimating population parameters from a sample. The mean computed from the collected sample is $178,053, but the actual value from the entire population may differ. In other words, this interval is narrow because it is calculated from a large number of samples.

The t distribution depends on a paramter named *degrees of freedom*. This equals the number of samples you used to calculate the mean and standard error minus one, as you see from the code as the second parameter to the function `stats.t.interval()`.

9.4 Understanding the Assumptions Behind

To skillfully apply confidence intervals in the dynamic landscape of the Ames real estate market, it's crucial to apprehend the foundational assumptions underpinning your analysis.

Assumption 1: Random Sampling. Your analysis assumes that the data is collected through a random sampling process, ensuring that each property in Ames has an equal chance of being included. This randomness enhances the generalizability of your findings to the entire real estate market.

Assumption 2: The Central Limit Theorem (CLT) and Large Samples. A pivotal assumption in your analysis is the Central Limit Theorem (CLT), which enables the use of the t-distribution in calculating confidence intervals. The CLT holds that for large samples, the sampling distribution of the sample mean approximates a normal distribution, regardless of the distribution of the population. In your case, with 2,579 observations, the CLT is robustly satisfied.

Assumption 3: Independence. You assume that the sales prices of individual houses are independent of each other. This assumption is crucial, ensuring that the sales price of one house does not influence the sales price of another. Imagine that a house is free on the condition that another house is purchased, then the entry with price zero falsely enlarges the standard deviation. It's particularly relevant in the diverse real estate market of Ames.

Assumption 4: Reliable Standard Deviation Estimates If Using z-score. The approach above uses the t-distribution, which is more robust when dealing with smaller sample sizes or the *population* standard derivation is unknown. An alternative would be to use z-score. But this requires that you can reliably estimate the population standard derivation (i.e., about the data not included in the sample).

Assumption 5: Continuous Data. Confidence intervals are applied to continuous data. In your context, the sales prices of houses in Ames are continuous variables, making confidence intervals appropriate for estimating population parameters.

These assumptions form the bedrock of your analysis, and it's imperative to recognize their role and assess their validity for dependable and insightful real estate market analysis. Violations of these assumptions could compromise the reliability of your conclusions. In summary, your methodology, rooted in the t-distribution, leverages these assumptions to provide nuanced insights into market trends and property values in Ames.

9.5 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

"Confidence Intervals". In: *Statistics and Probability*. Unit 11. Khan Academy.
https://www.khanacademy.org/math/statistics-probability/confidence-intervals-one-sample

`scipy.stats.t` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.t.html

9.6 Summary

In this exploration, you introduced the concept of confidence intervals using the Ames Housing dataset. By understanding the range in which the true average sales price of houses in Ames likely falls, stakeholders can make more informed decisions in the real estate market.

Specifically, you learned:

- ▷ The foundational concept of confidence intervals in inferential statistics.
- ▷ How to estimate and interpret the 95% confidence interval for the mean sales price in the Ames housing market.
- ▷ The critical assumptions underlying confidence interval calculations.

In the next chapter, you will revisit the t-distribution in the context of hypothesis testing.

Testing Assumptions in Real Estate: A Dive into Hypothesis Testing

10

In doing inferential statistics, you often want to test your assumptions. Indeed there is a way to quantitatively test an assumption that you thought of. Using the Ames Housing dataset, you'll delve deep into the concept of hypothesis testing and explore if the presence of an air conditioner affects the sale price of a house.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ The Role of Hypothesis Testing in Inferential Statistics.
- ▷ How does Hypothesis Testing work?
- ▷ Does Air Conditioning Affect Sale Price?

10.1 The Role of Hypothesis Testing in Inferential Statistics

Inferential statistics uses a sample of data to make inferences about the population from which it was drawn. Hypothesis testing is crucial if you need to make decisions about a population while the sample data is all you have.

Imagine you've come across a claim stating that houses with air conditioners sell at a higher price than those without. To verify this claim, you'd gather data on house sales and analyze if there's a *significant difference* in prices based on the presence of air conditioning. Merely comparing the difference in the average price is not enough, since you need to tell that the difference is large enough that it must not be due to sampling error.

This process of testing claims or assumptions about a population using sample data is known as hypothesis testing. In essence, hypothesis testing allows you to make an informed decision (either rejecting or failing to reject a starting assumption) based on evidence from the sample and the likelihood that the observation occurred by chance.

10.2 How does Hypothesis Testing work?

Hypothesis Testing is a methodological approach in inferential statistics where you start with an initial claim (hypothesis) about a population parameter. You then use sample data to determine whether there's enough evidence to reject this initial claim. The components of hypothesis testing include:

- ▷ *Null Hypothesis (H_0)*: The *default state* of no effect or no different. A statement that you aim to test against.
- ▷ *Alternative Hypothesis (H_1)*: What you want to prove. It is what you believe if the null hypothesis is wrong.
- ▷ *Test Statistic*: A value computed from the sample data that's used to test the null hypothesis.
- ▷ *P-value*: The probability that the observed effect in the sample occurred by random chance under the null hypothesis situation.
- ▷ *Level of significance (α)*: The threshold you compare to the p-value to determine if H_0 or H_1 should be accepted.

Performing hypothesis testing is like being a detective: Ordinarily, you assume something should happen (H_0), but you suspect something else is happening (H_1). Then you collect your evidence (the test statistic) to argue why H_0 is not reasonable; hence H_1 should be the truth.

In a typical hypothesis test:

1. You state the null and alternative hypotheses. You should carefully design these hypotheses to reflect a reasonable assumption about reality.
2. You choose a level of significance (α); it is common to use $\alpha = 0.05$ in statistical hypothesis tests.
3. You collect and analyze the data to get your test statistic and p-value, based on the situation of H_0.
4. You make a decision based on the p-value: You reject the null hypothesis and accept the alternative hypothesis if and only if the p-value is less than α.

Let's see an example of how these steps are carried out.

10.3 Does Air Conditioning Affect Sales Price?

Based on the Ames Dataset, you want to know if the presence of air conditioning can affect the price.

To explore the impact of air conditioning on sales prices, you'll set the hypotheses as:

- ▷ H_0: The average sales price of houses with air conditioning is the same as those without.
- ▷ H_1: The average sales price of houses with air conditioning is not the same as those without.

Before performing the hypothesis test, let's visualize your data to get a preliminary understanding.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
Ames = pd.read_csv('Ames.csv')

# Data separation
ac_prices = Ames[Ames['CentralAir'] == 'Y']['SalePrice']
no_ac_prices = Ames[Ames['CentralAir'] == 'N']['SalePrice']

# Setting up the visualization
plt.figure(figsize=(10, 6))

# Histograms for sale prices based on air conditioning
# Plotting 'With AC' first for the desired order in the legend
plt.hist(ac_prices, bins=30, alpha=0.7, color='blue', edgecolor='blue', lw=0.5,
         label='Sales Prices With AC')
mean_ac = np.mean(ac_prices)
plt.axvline(mean_ac, color='blue', linestyle='dashed', linewidth=1.5,
            label=f'Mean (With AC): ${mean_ac:.2f}')

plt.hist(no_ac_prices, bins=30, alpha=0.7, color='red', edgecolor='red', lw=0.5,
         label='Sales Prices Without AC')
mean_no_ac = np.mean(no_ac_prices)
plt.axvline(mean_no_ac, color='red', linestyle='dashed', linewidth=1.5,
            label=f'Mean (Without AC): ${mean_no_ac:.2f}')

plt.title('Distribution of Sales Prices based on Presence of Air Conditioning',
          fontsize=18)
plt.xlabel('Sales Price', fontsize=15)
plt.ylabel('Number of Houses', fontsize=15)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```

Listing 10.1: Create overlapping histograms of the sales price based on the presence of air conditioning

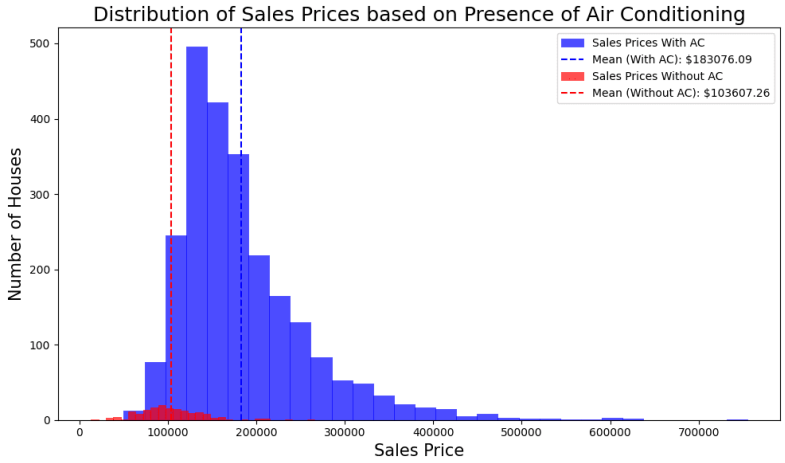

Figure 10.1: Overlapped histogram to compare the sales prices

The above code invoked `plt.hist()` twice with different data to show two overlapped histograms, one representing the distribution of sales price with air conditioning (AC) and the other without. Here are a few observations that can be made from the visual:

- ▷ *Distinct Peaks*: Both distributions exhibit a *distinct* peak, which indicates the most frequent sale prices in their respective categories.
- ▷ *Mean Sale Price*: The mean sale price of houses with AC is higher than that of houses without AC, as indicated by the vertical dashed lines.
- ▷ *Spread and Skewness*: The distribution of sale prices for houses with AC appears slightly right-skewed, indicating a broader range of prices with some properties sold at significantly higher prices. In contrast, the distribution for houses without AC is more compact, with a smaller range of prices.
- ▷ *Overlap*: Despite the differences in means, there's an overlap in the price range of houses with and without AC. This suggests that while AC may influence price, other factors are also at play in determining the value of a house.

Given these insights, the presence of AC seems to be associated with a higher sale price. The next step would be to perform the hypothesis test to numerically determine if this difference is significant.

```
import pandas as pd
import scipy.stats as stats

Ames = pd.read_csv('Ames.csv')
ac_prices = Ames[Ames['CentralAir'] == 'Y']['SalePrice']
no_ac_prices = Ames[Ames['CentralAir'] == 'N']['SalePrice']

# Performing a two-sample t-test
t_stat, p_value = stats.ttest_ind(ac_prices, no_ac_prices, equal_var=False)

# Printing the results
if p_value < 0.05:
    result = "reject the null hypothesis"
```

```
else:
    result = "fail to reject the null hypothesis"
print(f"With a p-value of {p_value:.5f}, we {result}.")
```

Listing 10.2: Running two-sample t-test using SciPy

This shows:

```
With a p-value of 0.00000, we reject the null hypothesis.
```

Output 10.1: The p-value of t-test

The p-value is less than $\alpha = 0.05$. The p-value says that it is very unlikely, under H_0, that the difference in the price is by chance. This indicates that the difference is statistically significant in the average sale prices of houses with air conditioning compared to those without. This aligns with your visual observations from the histogram. Thus, the presence of an air conditioner does seem to have a significant effect on the sale price of houses in the Ames dataset.

This p-value is computed using t-test. It is a statistic aimed at comparing the *means of two groups*. There are many statistics available, and t-test is a suitable one here because your hypotheses H_0, H_1 are about the average sales price.

If p-value is small but not less than α, you may still believe the alternative hypothesis H_1 is true but the evidence is *not strong* enough to conclude. This is how appropriately setting up H_0 and H_1 can affect the conclusion.

Note that the alternative hypothesis H_1 defined above can be changed. You can make it mean "the average sales price of houses with air conditioning is *less than* those without"; however, this is counter-intuitive to the reality. You can also make it mean "the average sales price of houses with air conditioning is *more than* those without"; which you should change the code to include the extra argument `alternative="greater"` in the t-test function:

```
import pandas as pd
import scipy.stats as stats

Ames = pd.read_csv('Ames.csv')
ac_prices = Ames[Ames['CentralAir'] == 'Y']['SalePrice']
no_ac_prices = Ames[Ames['CentralAir'] == 'N']['SalePrice']

# Performing a two-sample t-test
t_stat, p_value = stats.ttest_ind(ac_prices, no_ac_prices, equal_var=False,
                                  alternative="greater")

# Printing the results
if p_value < 0.05:
    result = "reject the null hypothesis"
else:
    result = "fail to reject the null hypothesis"
print(f"With a p-value of {p_value:.5f}, we {result}.")
```

Listing 10.3: Running t-test on a different alternative hypothesis

This changes the two-sided t-test to a one-sided t-test, but the resulting outcome is the same. Switching from a two-sided to a one-sided t-test but arriving at the same conclusion implies that you had a clear expectation of the direction of the difference from the start, or the data strongly supported one direction of difference making the outcome consistent across both test types.

The setup of the null hypothesis (H_0) and alternative hypothesis (H_1) is fundamental to the design of statistical tests, influencing the directionality of the test (one-sided vs. two-sided), the interpretation of results (how you understand p-values and evidence), and decision-making process (especially when the p-value is close to the significance level α). This framework determines not only what you are testing for but also how you interpret and act on the statistical evidence obtained.

10.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

"Hypothesis Testing Tutorial". In: *Statistics and Probability*. Unit 12. Khan Academy.
https://www.khanacademy.org/math/statistics-probability/significance-tests-one-sample

Student's t-test. Wikipedia.
https://en.wikipedia.org/wiki/Student%27s_t-test

`scipy.stats.ttest_ind` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html

10.5 Summary

In this exploration, you delved into the world of hypothesis testing using the Ames Housing dataset. You examined how the presence of an air conditioner might impact the sale price of a house. Through rigorous statistical testing, you found that houses with air conditioning tend to have a higher sale price than those without, a result that holds statistical significance. This not only underscores the importance of amenities like air conditioning in the real estate market but also showcases the power of hypothesis testing in making informed decisions based on data.

Specifically, you learned:

- ▷ The importance of hypothesis testing within inferential statistics.
- ▷ How to setup and evaluate null and alternative hypotheses using detailed methods of hypothesis testing.
- ▷ The practical implications of hypothesis testing in real-world scenarios, exemplified by the presence of air conditioning on property values in the Ames housing market.

There are more than t-test in statistics. In the next chapter, you will learn the chi-squared test, which is useful to disprove two features are associated.

Garage or Not? Housing Insights Through the Chi-Squared Test

11

The chi-squared test for independence is a statistical procedure employed to assess the relationship between two categorical variables—determining whether they are correlated or independent. The exploration of the visual appeal of a property and its impact on its valuation is intriguing. But how often do you associate the outlook of a house with functional features like a garage? With the chi-squared test, you can determine whether there exists a statistically significant correlation between features.

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ▷ Understanding the Chi-Squared Test
- ▷ How the Chi-Squared Test Works
- ▷ Unraveling the Correlation Between External Quality and Garage Presence
- ▷ Important Caveats

11.1 Understanding the Chi-Squared Test

The chi-squared (χ^2) test is useful because of its ability to test for relationships between *categorical variables*. It's particularly valuable when working with nominal or ordinal data, where the variables are divided into categories or groups.

- ▷ *Are two categorical variables independent of each other?*
  If the variables are independent, changes in one variable are not related to changes in the other. There is no correlation between them.
- ▷ *Is there a significant correlation between the two categorical variables?*
  Conversely, if the variables are correlated, changes in one variable usually change the other. The chi-squared test helps to quantify whether this correlation is statistically significant.

In this study, you will focus on the visual appeal of a house (categorized as "Great" or "Average") and its relation to the presence or absence of a garage. For the results of the chi-squared test to be valid, the following conditions must be satisfied:

- ▷ *Independence*: The observations must be independent, meaning the occurrence of one outcome shouldn't affect another. Your dataset satisfies this as each entry represents a distinct house.
- ▷ *Large Sample Size*: The dataset should be randomly sampled and sufficiently large to be representative. Your data, sourced from Ames, Iowa, meets this criterion.
- ▷ *Minimum Expected Frequency for Each Category*: Every cell in the contingency table should have an expected frequency of at least 5. This is vital for the reliability of the test, as the chi-squared test relies on a large sample approximation. You will demonstrate this condition below by creating and visualizing the expected frequencies.

11.2 How the Chi-Squared Test Works

The chi-squared test compares the observed frequencies from data to the expected frequencies from assumptions. It works by comparing the observed frequencies of the categories in a contingency table to the expected frequencies that would be expected under the assumption of independence. The contingency table is a cross-tabulation of the two categorical variables, showing how many observations fall into each combination of categories.

- ▷ *Null Hypothesis (H_0)*: The null hypothesis in the chi-squared test assumes independence between the two variables, i.e., the observed frequencies of Great and Average houses and that of with and without garages should match with each other.
- ▷ *Alternative Hypothesis (H_1)*: The alternative hypothesis suggests that there is a significant correlation between the two variables, i.e., the observed frequencies (with or without garage) should differ based on the value of another variable (quality of a house).

The test statistic in the chi-squared test is calculated by comparing the observed and expected frequencies in each cell of the contingency table. The larger the difference between observed and expected frequencies, the larger the chi-squared statistic becomes. The chi-squared test produces a p-value, which indicates the probability of observing the observed correlation (or a more extreme one) under the assumption of independence. If the p-value is below a chosen significance level α (commonly 0.05), the null hypothesis of independence is rejected, suggesting a significant correlation.

11.3 Unraveling the Correlation Between External Quality and Garage Presence

Using the Ames housing dataset, you set out to determine whether there's a correlation between the external quality of a house and the presence or absence of a garage. Let's delve into the specifics of your analysis, supported by the corresponding Python code.

```
import pandas as pd
from scipy.stats import chi2_contingency

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Extracting the relevant columns
data = Ames[['ExterQual', 'GarageFinish']].copy()

# Filling missing values in the 'GarageFinish' column with 'No Garage'
data['GarageFinish'] = data['GarageFinish'].fillna('No Garage')

# Grouping 'GarageFinish' into 'With Garage' and 'No Garage'
data['Garage Group'] \
    = data['GarageFinish'] \
      .apply(lambda x: 'With Garage' if x != 'No Garage' else 'No Garage')

# Grouping 'ExterQual' into 'Great' and 'Average'
data['Quality Group'] \
    = data['ExterQual'].apply(lambda x: 'Great' if x in ['Ex', 'Gd'] else 'Average')

# Constructing the simplified contingency table
simplified_contingency_table = pd.crosstab(data['Quality Group'], data['Garage Group'])

#Printing the Observed Frequency
print("Observed Frequencies:")
observed_df = pd.DataFrame(simplified_contingency_table,
                           index=["Average", "Great"],
                           columns=["No Garage", "With Garage"])
print(observed_df)
print()

# Performing the chi-squared test
chi2_stat, p_value, _, expected_freq = chi2_contingency(simplified_contingency_table)

# Printing the Expected Frequencies
print("Expected Frequencies:")
print(pd.DataFrame(expected_freq,
                   index=["Average", "Great"],
                   columns=["No Garage", "With Garage"]).round(1))
print()

# Printing the results of the test
print(f"Chi-squared Statistic: {chi2_stat:.4f}")
print(f"p-value: {p_value:.4e}")
```

Listing 11.1: Running chi-squared test using SciPy

The output should be:

```
Observed Frequencies:
         No Garage  With Garage
Average        121         1544
Great            8          906
```

```
Expected Frequencies:
         No Garage  With Garage
Average       83.3       1581.7
Great         45.7        868.3

Chi-squared Statistic: 49.4012
p-value: 2.0862e-12
```

Output 11.1: Frequencies and the result of chi-squared test

The code above performs three steps:

1. **Data Loading & Preparation:**
   - ▷ You began by loading the dataset and extracting the pertinent columns: `ExterQual` (Exterior Quality) and `GarageFinish` (Garage Finish).
   - ▷ Recognizing the missing values in `GarageFinish`, you sensibly imputed them with the label `"No Garage"`, indicating houses devoid of garages.

2. **Data Grouping for Simplification:**
   - ▷ You further categorized the `GarageFinish` data into two groups: "With Garage" (for houses with any kind of garage) and "No Garage".
   - ▷ Similarly, you grouped the `ExterQual` data into "Great" (houses with excellent or good exterior quality) and "Average" (houses with average or fair exterior quality).

3. **Chi-squared Test:**
   - ▷ With the data aptly prepared, you constructed a contingency table to depict the observed frequencies between the newly formed categories. They are the two tables printed in the output.
   - ▷ You then performed a chi-squared test on this contingency table using SciPy. The p-value is printed and found much less than α (0.05). The extremely low p-value suggests rejecting the null hypothesis, meaning there is a statistically significant relationship between the external quality of a house and the presence of a garage in this dataset.
   - ▷ A glance at the expected frequencies satisfies the third condition of a chi-squared test, which requires a minimum of 5 occurrences in each cell.

Through this analysis, you not only refined and simplified the data to make it more interpretable but also provided statistical evidence of a correlation between two categorical variables of interest.

11.4 Important Caveats

The chi-squared test, despite its utility, has its limitations:

- ▷ *No Causation:* While the test can determine correlation, it doesn't infer causation. So, even though there's a significant link between the external quality of a house and its garage presence, you can't conclude that one causes the other.
- ▷ *Directionality:* The test indicates an correlation but doesn't specify its direction. However, your data suggests that houses labeled as "Great" in terms of external quality are more likely to have garages than those labeled as "Average".
- ▷ *Magnitude:* The test doesn't provide insights into the strength of a relationship. Other metrics, like Cramér's V, would be more informative in this regard.
- ▷ *External Validity:* Your conclusions are specific to the Ames dataset. Caution is advised when generalizing these findings to other regions.

11.5 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

H. B. Berman. *Chi-square Test for Independence*. Stat Trek.
https://stattrek.com/chi-square-test/independence

Chi-square test. Wikipedia.
https://en.wikipedia.org/wiki/Chi-squared_test

`scipy.stats.chi2_contingency` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2_contingency.html

11.6 Summary

In this chapter, you delved into the chi-squared test and its application on the Ames housing dataset. You discovered a significant correlation between the external quality of a house and the presence of a garage.

Specifically, you learned:

- ▷ The fundamentals and practicality of the chi-squared test.
- ▷ The chi-squared test revealed a significant correlation between the external quality of a house and the presence of a garage in the Ames dataset. Houses with a "Great" external quality rating showed a higher likelihood of having a garage when compared to those with an "Average" rating, a trend that was statistically significant.
- ▷ The vital caveats and limitations of the chi-squared test.

In the next chapter, you will learn about ANOVA, which can be considered as an extension of the t-test.

Leveraging ANOVA and Kruskal-Wallis Tests to Analyze the Impact of the Great Recession on Housing Prices

12

In the world of real estate, numerous factors influence property prices. The economy, market demand, location, and even the year a property is sold can play significant roles. The years 2007 to 2009 marked a tumultuous time for the US housing market. This period, often referred to as the Great Recession, saw a drastic decline in home values, a surge in foreclosures, and widespread financial market turmoil. The impact of the recession on housing prices was profound, with many homeowners finding themselves in homes that were worth less than their mortgages. The ripple effect of this downturn was felt across the country, with some areas experiencing sharper declines and slower recoveries than others.

Given this backdrop, it's particularly intriguing to analyze housing data from Ames, Iowa, as the dataset spans from 2006 to 2010, encapsulating the height and aftermath of the Great Recession. Does the year of sale, amidst such economic volatility, influence the sales price in Ames? In this chapter, you'll delve deep into the Ames Housing dataset to explore this query using Exploratory Data Analysis (EDA) and two statistical tests: ANOVA and the Kruskal-Wallis Test.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ EDA: Visual Insights
- ▷ Assessing Variability in Sales Prices Across Years Using ANOVA
- ▷ Kruskal-Wallis Test: A Nonparametric Alternative

12.1 EDA: Visual Insights

To begin the exploratory data analysis, let's load the Ames Housing dataset and compare different years of sale against the dependent variable: the sales price.

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

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Convert 'YrSold' to a categorical variable
Ames['YrSold'] = Ames['YrSold'].astype('category')

plt.figure(figsize=(10, 6))
sns.boxplot(x=Ames['YrSold'], y=Ames['SalePrice'], hue=Ames['YrSold'])
plt.title('Boxplot of Sales Prices by Year', fontsize=18)
plt.xlabel('Year Sold', fontsize=15)
plt.ylabel('Sales Price (US$)', fontsize=15)
plt.legend('')
plt.show()
```

Listing 12.1: Creating a box plot for the sales price

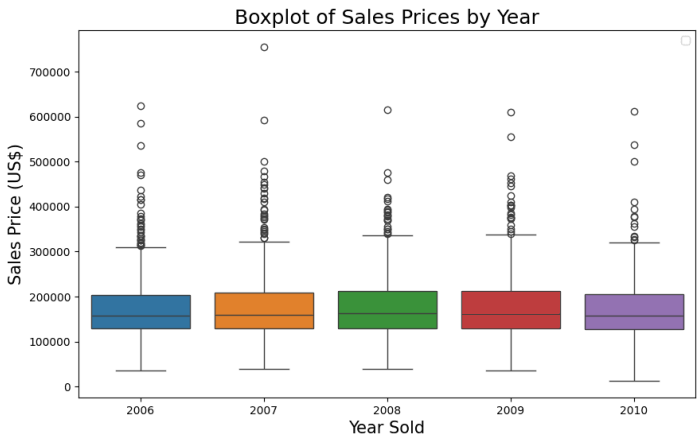

Figure 12.1: Comparing the trend of sales prices

From the box plot, you can observe that the sales prices were quite consistent across different years because each year looks alike. Let's take a closer look using the `groupby` function in pandas.

```
import pandas as pd
Ames = pd.read_csv('Ames.csv')

# Calculating mean and median sales price by year
summary_table = Ames.groupby('YrSold')['SalePrice'].agg(['mean', 'median'])

# Rounding the values for better presentation
summary_table = summary_table.round(2)
print(summary_table)
```

Listing 12.2: Finding the mean and median sales prices

This is what you should get:

```
              mean    median
YrSold
2006    176615.62  157000.0
2007    179045.08  159000.0
2008    178170.02  162700.0
2009    180387.64  162000.0
2010    173971.67  157900.0
```

Output 12.1: The mean and median of sales prices

From the table, you can make the following observations:

1. The *mean* sales price was the highest in 2009 at approximately $180,388, while it was the lowest in 2010 at around $173,972.
2. The *median* sales price was the highest in 2008 at $162,700 and the lowest in 2006 at $157,000.
3. Even though the mean and median sales prices are close in value for each year, there are slight variations. This suggests that while there might be some outliers influencing the mean, they are not extremely skewed.
4. Over the five years, there doesn't seem to be a consistent upward or downward trend in sales prices, which is interesting given the larger economic context (the Great Recession) during this period.

This table, combined with the box plot, gives a comprehensive view of the distribution and central tendency of sales prices across the years. It sets the stage for deeper statistical analysis to determine if the observed differences (or lack thereof) are statistically significant.

12.2 Assessing Variability in Sales Prices Across Years Using ANOVA

ANOVA (Analysis of Variance) helps you test if there are any statistically significant differences *between the means* of three or more independent groups. Its null hypothesis is that the means of all groups are equal. This can be considered as a version of t-test to support more than two groups. It makes use of the F-test statistic to check if the variance (σ^2) is different within each group compared to across all groups. In the following example, you compare the sales prices of all years and that of each year. If no year is special, all these sales prices should have the same distribution.

The hypothesis setup is:

- ▷ H_0: The means of sales price for all years are equal.
- ▷ H_1: At least one year has a different mean sales price.

You can run your test using the `scipy.stats` library as follows:

```
import pandas as pd
import scipy.stats as stats

Ames = pd.read_csv('Ames.csv')

# Perform the ANOVA
f_value, p_value = stats.f_oneway(*[Ames['SalePrice'][Ames['YrSold'] == year]
                                    for year in Ames['YrSold'].unique()])
print(f_value, p_value)
```

Listing 12.3: Performing ANOVA using SciPy

The two values are:

```
0.4478735462379817 0.774024927554816
```

Output 12.2: The F statistic and its p-value

The results of the ANOVA test are:

- ▷ *F-value:* 0.4479
- ▷ *p-value:* 0.7740

Given the high *p-value* (greater than a common significance level of 0.05), you cannot reject the null hypothesis (H_0). This suggests that there are no statistically significant differences between the means of sales price for the different years present in the dataset.

Is it? Everyone knows the crisis of 2008 had a significant impact on the housing market.

Before accepting your ANOVA results, it's essential to ensure that the assumptions underlying the test have been met. Let's delve into verifying the 3 assumptions of ANOVA tests to validate your findings.

Assumption 1: Independence of Observations. Since each observation (house sale) is independent of another, this assumption is met.

Assumption 2: Normality of the Residuals. For ANOVA to be valid, the residuals from the model should approximately follow a *normal distribution* since this is the model behind the F-test. You can check this both visually and statistically.

Visual assessment can be done using a QQ plot:

```
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

Ames = pd.read_csv('Ames.csv')

# Fit an ordinary least squares model and get residuals
model = sm.OLS(Ames['SalePrice'], Ames['YrSold'].astype('int')).fit()
residuals = model.resid

# Plot QQ plot
```

```
sm.qqplot(residuals, line='s')
plt.title('Normality Assessment of Residuals via QQ Plot', fontsize=18)
plt.xlabel('Theoretical Quantiles', fontsize=15)
plt.ylabel('Sample Residual Quantiles', fontsize=15)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

Listing 12.4: Creating a QQ plot

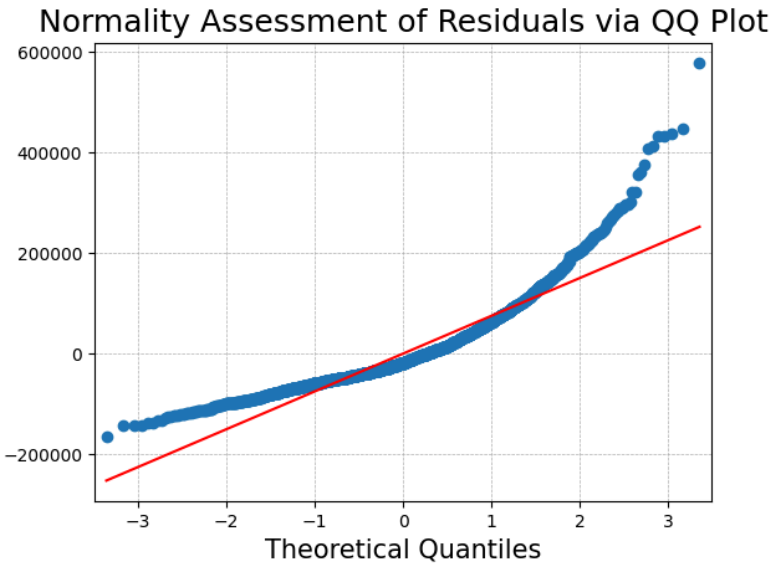

Figure 12.2: QQ plot to confirm normality

The QQ Plot presented above serves as a valuable visual tool to assess the normality of residuals of your dataset, offering insights into how well the observed data aligns with the theoretical expectations of a normal distribution. In this plot, each point represents a pair of quantiles: one from the residuals of your data and the other from the standard normal distribution. Ideally, if your data perfectly followed a normal distribution, all the points on the QQ Plot would fall precisely along the red 45-degree reference line. The plot illustrates deviations from the 45-degree reference line, suggesting potential deviations from normality.

Statistical assessment can be done using the Shapiro-Wilk Test, which provides a formal method to test for normality. The null hypothesis of the test is that the data follows a normal distribution. This test is also available in SciPy:

```
...
# Import shapiro from scipy.stats package
from scipy.stats import shapiro

# Shapiro-Wilk Test
shapiro_stat, shapiro_p = shapiro(residuals)
print(f"Shapiro-Wilk Test Statistic: {shapiro_stat}")
print(f"P-value: {shapiro_p}")
```

Listing 12.5: Running Shapiro-Wilk test in SciPy

The output is:

```
Shapiro-Wilk Test Statistic: 0.8774482011795044
P-value: 4.273399796804962e-41
```

Output 12.3: The statistic and the corresponding p-value

A low p-value (typically $p < 0.05$) suggests rejecting the null hypothesis, indicating that the residuals do not follow a normal distribution. This indicates a violation of the second assumption of ANOVA, which requires that the residuals be normally distributed. Both the QQ plot and the Shapiro-Wilk test converge on the same conclusion: the residuals do not strictly adhere to a normal distribution. Hence, the result of the ANOVA may not be valid.

Assumption 3: Homogeneity of Variances. The variances of the groups (years) should be approximately equal. This happens to be the null hypothesis of Levene's test. Hence you can use it to verify:

```
...
# Check for equal variances using Levene's test
levene_stat, levene_p = stats.levene(*[Ames['SalePrice'][Ames['YrSold'] == year]
                                       for year in Ames['YrSold'].unique()])

print(f"Levene's Test Statistic: {levene_stat}")
print(f"P-value: {levene_p}")
```

Listing 12.6: Running Levene's test in SciPy

The output is:

```
Levene's Test Statistic: 0.2514412478357097
P-value: 0.9088910499612235
```

Output 12.4: The statistic and the corresponding p-value

Given the high p-value of 0.909 from Levene's test, you cannot reject the null hypothesis, indicating that the variances of sales prices across different years are statistically homogeneous, satisfying the third key assumption for ANOVA.

Putting all together, the following code runs the ANOVA test and verifies the three assumptions. This is the suggested practice to verify the assumptions rather than unconditionally saying the hypothesis is accepted or rejected by ANOVA:

```
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import statsmodels.api as sm
from scipy.stats import shapiro

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Perform the ANOVA
```

```
f_value, p_value = stats.f_oneway(*[Ames['SalePrice'][Ames['YrSold'] == year]
                                    for year in Ames['YrSold'].unique()])
print("F-value:", f_value)
print("p-value:", p_value)

# Fit an ordinary least squares model and get residuals
model = sm.OLS(Ames['SalePrice'], Ames['YrSold'].astype('int')).fit()
residuals = model.resid

# Plot QQ plot
sm.qqplot(residuals, line='s')
plt.title('Normality Assessment of Residuals via QQ Plot', fontsize=18)
plt.xlabel('Theoretical Quantiles', fontsize=15)
plt.ylabel('Sample Residual Quantiles', fontsize=15)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()

# Shapiro-Wilk Test
shapiro_stat, shapiro_p = shapiro(residuals)
print(f"Shapiro-Wilk Test Statistic: {shapiro_stat}")
print(f"P-value: {shapiro_p}")

# Check for equal variances using Levene's test
levene_stat, levene_p = stats.levene(*[Ames['SalePrice'][Ames['YrSold'] == year]
                                       for year in Ames['YrSold'].unique()])
print(f"Levene's Test Statistic: {levene_stat}")
print(f"P-value: {levene_p}")
```

Listing 12.7: Performing multiple statistics

12.3 Kruskal-Wallis Test: A Nonparametric Alternative

The ANOVA test above is inconclusive because the normality assumption has not been met. You need an alternative.

The Kruskal-Wallis test is a nonparametric method used to compare the median values of three or more independent groups, making it a suitable alternative to the one-way ANOVA (especially when assumptions of ANOVA are not met).

Nonparametric statistics are a class of statistical methods that do not make explicit assumptions about the underlying distribution of the data. In contrast to *parametric* tests, which assume a specific distribution (e.g., normal distribution in assumption 2 above), *nonparametric* tests are more flexible and can be applied to data that may not meet the stringent assumptions of parametric methods. Nonparametric tests are particularly useful when dealing with ordinal or nominal data, as well as data that might exhibit skewness or heavy tails. These tests focus on the order or rank of values rather than the specific values themselves. Nonparametric tests, including the Kruskal-Wallis test, offer a flexible and distribution-free approach to statistical analysis, making them suitable for a wide range of data types and situations.

The hypothesis setup under the Kruskal-Wallis test is:

- ▷ H_0: The distributions of the sales price for all years are identical.
- ▷ H_1: At least one year has a different distribution of sales price.

You can run the Kruskal-Wallis test using SciPy, as follows:

```
...
# Perform the Kruskal-Wallis H-test
H_statistic, kruskal_p_value = stats.kruskal(*[Ames['SalePrice'][Ames['YrSold'] == year]
                                               for year in Ames['YrSold'].unique()])
print(H_statistic, kruskal_p_value)
```

Listing 12.8: Perform the Kruskal-Wallis test in SciPy

The output is:

```
2.1330989438609236 0.7112941815590765
```

Output 12.5: The H-statistic and its p-value

The results of the Kruskal-Wallis test are:

- ▷ *H-statistic:* 2.133
- ▷ *p-value:* 0.7113

The Kruskal-Wallis test doesn't specifically test for differences in means (like ANOVA does), but rather for differences in distributions. This can include differences in medians, shapes, and spreads.

Given the high *p-value* (greater than a common significance level of 0.05), you cannot reject the null hypothesis. This suggests that there are no statistically significant differences in the median sales prices for the different years present in the dataset when using the Kruskal-Wallis test. Let's delve into verifying the 3 assumptions of the Kruskal-Wallis test to validate your findings.

Assumption 1: Independence of Observations. This remains the same as for ANOVA; each observation is independent of another.

Assumption 2: The Response Variable Should be Ordinal, Interval, or Ratio. The sales price is a ratio variable, so this assumption is met.

Assumption 3: The Distributions of the Response Variable Should be the Same for All Groups. This can be validated using both visual and numerical methods.

```
...
# Plot histograms of Sales Price for each year
fig, axes = plt.subplots(nrows=5, ncols=1, figsize=(12, 8), sharex=True)

for idx, year in enumerate(sorted(Ames['YrSold'].unique())):
    sns.histplot(Ames[Ames['YrSold'] == year]['SalePrice'], kde=True, ax=axes[idx],
```

```
                color='skyblue')
    axes[idx].set_title(f'Distribution of Sales Prices for Year {year}', fontsize=16)
    axes[idx].set_ylabel('Frequency', fontsize=14)
    if idx == 4:
        axes[idx].set_xlabel('Sales Price', fontsize=15)
    else:
        axes[idx].set_xlabel('')

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

Listing 12.9: Creating a histogram showing the sales price distribution of different years

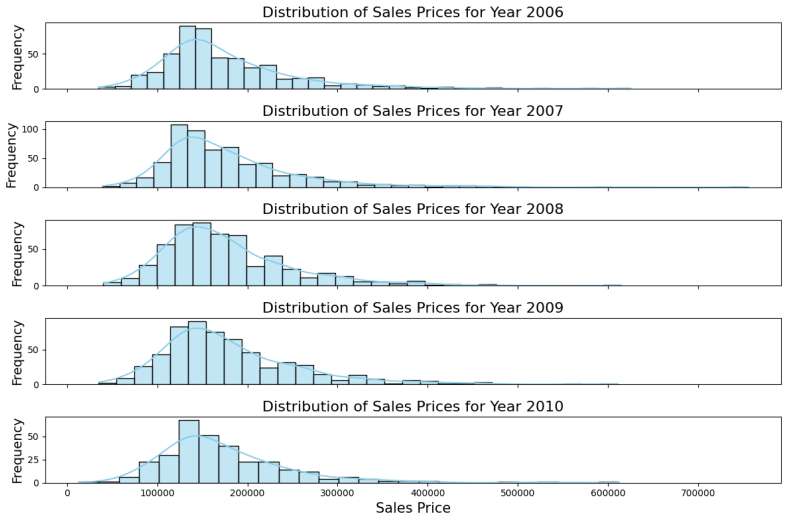

Figure 12.3: Distribution of sales prices of different years

The stacked histograms indicate consistent distributions of sales prices across the years, with each year displaying a similar range and peak despite slight variations in frequency.

Furthermore, you can conduct pairwise Kolmogorov-Smirnov tests, which is a nonparametric test to compare the similarity of two probability distributions. It is available in SciPy. You can use the version that the null hypothesis is the two distributions equal, and the alternative hypothesis is not equal:

```
...
# Run KS Test from scipy.stats
from scipy.stats import ks_2samp
results = {}
for i, year1 in enumerate(sorted(Ames['YrSold'].unique())):
    for j, year2 in enumerate(sorted(Ames['YrSold'].unique())):
        if i < j:
            ks_stat, ks_p = ks_2samp(Ames[Ames['YrSold'] == year1]['SalePrice'],
                                     Ames[Ames['YrSold'] == year2]['SalePrice'])
            results[f"{year1} vs {year2}"] = (ks_stat, ks_p)
```

```
# Convert the results into a DataFrame for tabular representation
ks_df = pd.DataFrame(results).transpose()
ks_df.columns = ['KS Statistic', 'P-value']
ks_df.reset_index(inplace=True)
ks_df.rename(columns={'index': 'Years Compared'}, inplace=True)
print(ks_df)
```

Listing 12.10: Running the Kolmogorov-Smirnov test from SciPy

This shows:

```
  Years Compared  KS Statistic   P-value
0   2006 vs 2007      0.038042  0.798028
1   2006 vs 2008      0.052802  0.421325
2   2006 vs 2009      0.062235  0.226623
3   2006 vs 2010      0.040006  0.896946
4   2007 vs 2008      0.039539  0.732841
5   2007 vs 2009      0.044231  0.586558
6   2007 vs 2010      0.051508  0.620135
7   2008 vs 2009      0.032488  0.908322
8   2008 vs 2010      0.052752  0.603031
9   2009 vs 2010      0.053236  0.586128
```

Output 12.6: Result of KS statstic to compare sales prices between years

While you satisfied only 2 out of the 3 assumptions for ANOVA, you have met all the necessary criteria for the Kruskal-Wallis test. The pairwise Kolmogorov-Smirnov tests indicate that the distributions of sales prices across different years are remarkably consistent. Specifically, the high p-values (all greater than the common significance level of 0.05) imply that there isn't enough evidence to reject the hypothesis that the sales prices for each year come from the same distribution. These findings satisfy the assumption for the Kruskal-Wallis Test that the distributions of the response variable should be the same for all groups. This underscores the stability in the sales price distributions from 2006 to 2010 in Ames, Iowa, despite the broader economic context.

Putting everything together, the following is the complete code:

```
import pandas as pd
import seaborn as sns
import scipy.stats as stats
import matplotlib.pyplot as plt
from scipy.stats import ks_2samp

Ames = pd.read_csv('Ames.csv')

# Perform the Kruskal-Wallis H-test
H_statistic, kruskal_p_value = stats.kruskal(*[Ames['SalePrice'][Ames['YrSold'] == year]
                                               for year in Ames['YrSold'].unique()])
print(H_statistic, kruskal_p_value)

# Plot histograms of Sales Price for each year
fig, axes = plt.subplots(nrows=5, ncols=1, figsize=(12, 8), sharex=True)
```

```
for idx, year in enumerate(sorted(Ames['YrSold'].unique())):
    sns.histplot(Ames[Ames['YrSold'] == year]['SalePrice'], kde=True, ax=axes[idx],
                 color='skyblue')
    axes[idx].set_title(f'Distribution of Sales Prices for Year {year}', fontsize=16)
    axes[idx].set_ylabel('Frequency', fontsize=14)
    if idx == 4:
        axes[idx].set_xlabel('Sales Price', fontsize=15)
    else:
        axes[idx].set_xlabel('')

plt.tight_layout()
plt.show()

# Run KS Test from scipy.stats
results = {}
for i, year1 in enumerate(sorted(Ames['YrSold'].unique())):
    for j, year2 in enumerate(sorted(Ames['YrSold'].unique())):
        if i < j:
            ks_stat, ks_p = ks_2samp(Ames[Ames['YrSold'] == year1]['SalePrice'],
                                     Ames[Ames['YrSold'] == year2]['SalePrice'])
            results[f"{year1} vs {year2}"] = (ks_stat, ks_p)

# Convert the results into a DataFrame for tabular representation
ks_df = pd.DataFrame(results).transpose()
ks_df.columns = ['KS Statistic', 'P-value']
ks_df.reset_index(inplace=True)
ks_df.rename(columns={'index': 'Years Compared'}, inplace=True)
print(ks_df)
```

Listing 12.11: Nonparametric tests

12.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

ANOVA. Statistics Solutions.
https://www.statisticssolutions.com/anova-analysis-of-variance/

Analysis of variance. Wikipedia.
https://en.wikipedia.org/wiki/Analysis_of_variance

`scipy.stats.f_oneway` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html

`scipy.stats.shapiro` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html

`scipy.stats.levene` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.levene.html

`scipy.stats.kruskal` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kruskal.html

`scipy.stats.ks_2samp` *API*. SciPy.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ks_2samp.html

12.5 Summary

In the multi-dimensional world of real estate, several factors, including the year of sale, can potentially influence property prices. The US housing market experienced considerable turbulence during the Great Recession between 2007 and 2009. The study focuses on housing data from Ames, Iowa, spanning 2006 to 2010, aiming to determine if the year of sale affected the sales price, particularly during this tumultuous period.

The analysis employed both the ANOVA and Kruskal-Wallis tests to gauge variations in sales prices across different years. While the findings of ANOVA were instructive, not all its underlying assumptions were satisfied, notably the normality of residuals. Conversely, the Kruskal-Wallis test met all its criteria, suggesting more reliable insights. Therefore, relying solely on the ANOVA could have been misleading without the corroborative perspective of the Kruskal-Wallis test.

Both the one-way ANOVA and the Kruskal-Wallis test yielded consistent results, indicating no statistically significant differences in sales prices across the different years. This outcome is particularly fascinating given the turbulent economic backdrop from 2006 to 2010. The findings demonstrate that property prices in Ames seemed to be very stable and influenced more by local conditions than the economy.

Specifically, you learned:

- The importance of validating the assumptions of statistical tests, as seen with the residuals normality challenge of ANOVA.
- The significance and application of both parametric (ANOVA) and nonparametric (Kruskal-Wallis) tests in comparing data distributions.
- How local factors can insulate property markets, like that of Ames, Iowa, from broader economic downturns, emphasizing the nuanced nature of real estate pricing.

In the next chapter, you will learn how to find outliers in the dataset.

Spotting the Exception: Classical Methods for Outlier Detection in Data Science

13

Outliers are unique in that they often don't play by the rules. These data points, which significantly differ from the rest, can skew your analyses and make your predictive models less accurate. Although detecting outliers is critical, there is no universally agreed-upon method for doing so. While some advanced techniques like machine learning offer solutions, in this chapter, you will focus on the foundational Data Science methods that have been in use for decades.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ Understanding Outliers and Their Impact
- ▷ Traditional Methods for Outlier Detection
- ▷ Detecting Outliers in the Ames Dataset

13.1 Understanding Outliers and Their Impact

Outliers can emerge for a variety of reasons, from data entry errors to genuine anomalies. Their presence can be attributed to factors like:

- ▷ Measurement errors
- ▷ Data processing errors
- ▷ Genuine extreme observations

Understanding the source of an outlier is crucial for determining whether to keep, modify, or discard it. The impact of outliers on statistical analyses can be profound. They can change the results of data visualizations, central tendency measurements, and other statistical tests.

Outliers can also influence the assumptions of normality, linearity, and homoscedasticity[1] in a dataset, leading to unreliable and spurious conclusions.

13.2 Traditional Methods for Outlier Detection

In the realm of Data Science, several classical methods exist for detecting outliers. These can be broadly categorized into:

- ▷ *Visual methods*: Plots and graphs, such as scatter plots, box plots, and histograms, provide an intuitive feel of the data distribution and any extreme values.
- ▷ *Statistical methods*: Techniques like the z-score, IQR (interquartile range), and the modified z-score are mathematical methods used to define outliers based on data distribution.
- ▷ *Probabilistic and statistical models*: These leverage the probability distribution of data, such as the Gaussian distribution, to detect unlikely observations.

It's essential to understand that the choice of method often depends on the nature of your dataset and the specific problem at hand.

13.3 Detecting Outliers in the Ames Dataset

In this section, you'll dive into the practical application of detecting outliers using the Ames Housing Dataset. Specifically, you'll explore three features: Lot Area, Sales Price, and Total Rooms Above Ground.

Visual Inspection

Visual methods are a quick and intuitive way to identify outliers. Let's start with box plots for your chosen features.

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

Ames = pd.read_csv('Ames.csv')

# Define feature names in full form for titles and axis
feature_names_full = {
    'LotArea': 'Lot Area (sq ft)',
    'SalePrice': 'Sales Price (US$)',
    'TotRmsAbvGrd': 'Total Rooms Above Ground'
}

plt.figure(figsize=(18, 6))
features = ['LotArea', 'SalePrice', 'TotRmsAbvGrd']
```

[1]Homoscedasticity is a term to mean the variance is uniform. For example, in the housing dataset with sales price and year of sales, you expect the variance of price to be identical in each year if homoscedastic.

```
for i, feature in enumerate(features, 1):
    plt.subplot(1, 3, i)
    sns.boxplot(y=Ames[feature], color="lightblue")
    plt.title(feature_names_full[feature], fontsize=16)
    plt.ylabel(feature_names_full[feature], fontsize=14)
    plt.xlabel('')  # Removing the x-axis label as it's not needed

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

Listing 13.1: Creating box plot on three features

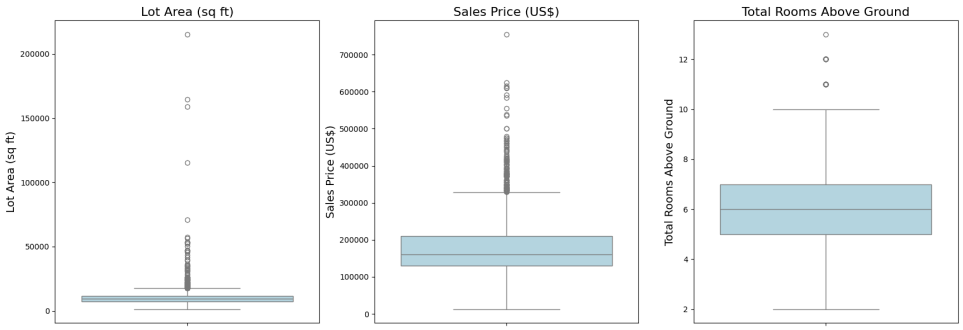

Figure 13.1: Three box plots for three features

These plots provide immediate insights into potential outliers in your data. The dots you see beyond the whiskers represent data points that are considered outliers, lying outside 1.5 times the interquartile range (IQR) from the first and third quartiles. For instance, you might notice properties with exceptionally large lot areas or homes with a large number of rooms above ground.

Statistical Methods: IQR

The dots in the box plots above are outliers. They are those greater than 1.5 times the interquartile range (IQR) from the third quartiles, which is a robust method to define outliers quantitatively. You can precisely find and count these dots from the pandas DataFrame without the box plot:

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')
features = ['LotArea', 'SalePrice', 'TotRmsAbvGrd']

def detect_outliers_iqr_summary(dataframe, features):
    outliers_summary = {}

    for feature in features:
        data = dataframe[feature]
        Q1 = data.quantile(0.25)
        Q3 = data.quantile(0.75)
```

```
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        outliers = data[(data < lower_bound) | (data > upper_bound)]
        outliers_summary[feature] = len(outliers)

    return outliers_summary

outliers_summary = detect_outliers_iqr_summary(Ames, features)
print(outliers_summary)
```

Listing 13.2: Finding outliers using IQR

This prints:

```
{'LotArea': 113, 'SalePrice': 116, 'TotRmsAbvGrd': 35}
```

Output 13.1: Number of outliers in each of the three columns

In your analysis of the Ames Housing Dataset using the interquartile range (IQR) method, you identified 113 outliers in the "Lot Area" feature, 116 outliers in the "Sales Price" feature, and 35 outliers for the "Total Rooms Above Ground" feature. These outliers are visually represented as dots beyond the whiskers in the box plots. The data points beyond the whiskers of the box plots are considered outliers. This is just one definition of outliers. Such values should be further investigated or treated appropriately in subsequent analyses.

Probabilistic and Statistical Models

The distribution of data can sometimes help you identify outliers. One of the most common assumptions about data distribution is that it follows a Gaussian (or normal) distribution. In a perfectly Gaussian distribution, about 68% of the data lies within one standard deviation from the mean, 95% within two standard deviations, and 99.7% within three standard deviations. Data points that fall far away from the mean (typically beyond three standard deviations) can be considered outliers.

This method is particularly effective when the dataset is large and believed to be normally distributed. Let's apply this technique to your Ames Housing Dataset and see what you find.

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

Ames = pd.read_csv('Ames.csv')
feature_names_full = {
    'LotArea': 'Lot Area (sq ft)',
    'SalePrice': 'Sales Price (US$)',
    'TotRmsAbvGrd': 'Total Rooms Above Ground'
}
features = ['LotArea', 'SalePrice', 'TotRmsAbvGrd']

# Define a function to detect outliers using the Gaussian model
def detect_outliers_gaussian(dataframe, features, threshold=3):
```

```
    outliers_summary = {}

    for feature in features:
        data = dataframe[feature]
        mean = data.mean()
        std_dev = data.std()
        outliers = data[(data < mean - threshold * std_dev) |
                        (data > mean + threshold * std_dev)]
        outliers_summary[feature] = len(outliers)

        # Visualization
        plt.figure(figsize=(12, 6))
        sns.histplot(data, color="lightblue")
        plt.axvline(mean, color='r', linestyle='-', label=f'Mean: {mean:.2f}')
        plt.axvline(mean - threshold * std_dev, color='y', linestyle='--',
                    label=f'-{threshold} std devs')
        plt.axvline(mean + threshold * std_dev, color='g', linestyle='--',
                    label=f'+{threshold} std devs')

        # Annotate upper 3rd std dev value
        annotate_text = f'{mean + threshold * std_dev:.2f}'
        plt.annotate(annotate_text, xy=(mean + threshold * std_dev, 0),
                     xytext=(mean + (threshold + 1.45) * std_dev, 50),
                     arrowprops={'facecolor': 'black',
                                 'arrowstyle': 'wedge,tail_width=0.7'},
                     fontsize=12, ha='center')

        plt.title(f'Distribution of {feature_names_full[feature]} with Outliers',
                  fontsize=16)
        plt.xlabel(feature_names_full[feature], fontsize=14)
        plt.ylabel('Frequency', fontsize=14)
        plt.legend()
        plt.show()

    return outliers_summary

outliers_gaussian_summary = detect_outliers_gaussian(Ames, features)
print(outliers_gaussian_summary)
```

Listing 13.3: Finding outliers using the Gaussian model

This shows these charts of distribution:

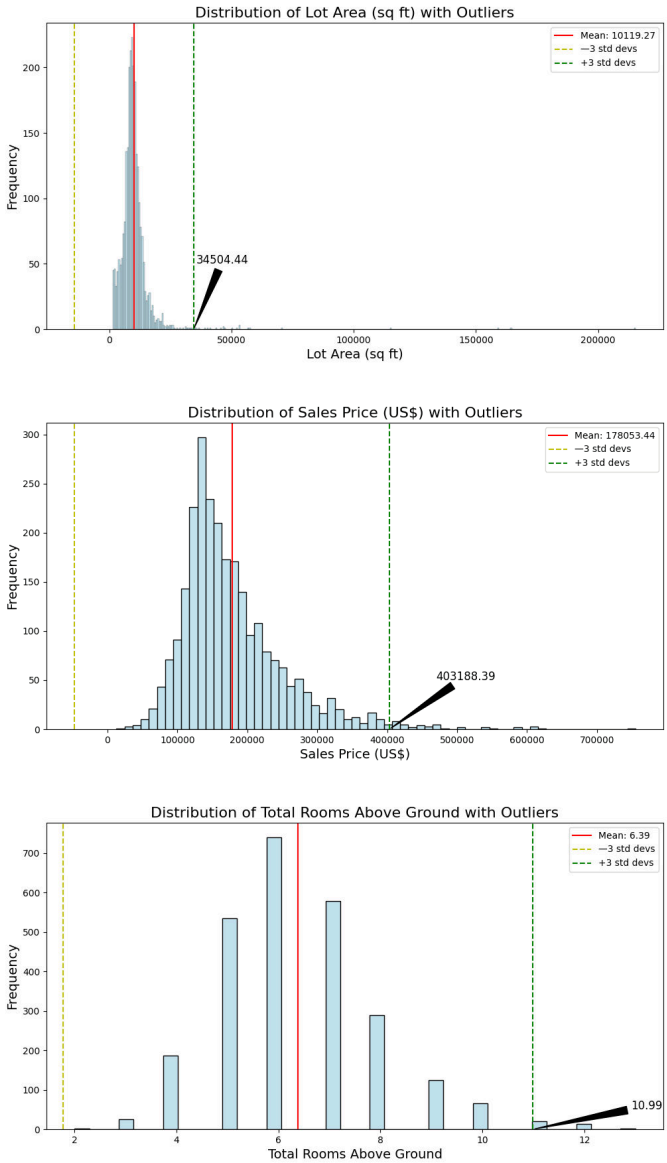

Figure 13.2: Histograms and the outliers as found by the Gaussian model

Then it prints the following:

```
{'LotArea': 24, 'SalePrice': 42, 'TotRmsAbvGrd': 35}
```

Output 13.2: Number of outliers found by the Gaussian model

Upon applying the Gaussian model for outlier detection, you observed that there are outliers in the "Lot Area," "Sales Price," and "Total Rooms Above Ground" features. These outliers are identified based on the upper threshold of three standard deviations from the mean:

- ▷ *Lot Area*: Any observation with a lot area larger than 34,505.44 square feet is considered an outlier. You found 24 such outliers in the dataset.
- ▷ *Sales Price*: Any observation above $403,188.39 is considered an outlier. Your analysis revealed 42 outliers in the "Sales Price" feature.

▷ *Total Rooms Above Ground*: Observations with more than 10.99 rooms above ground are considered outliers. You identified 35 outliers using this criterion.

The number of outliers is different because the definition of outliers is different. These figures differ from your earlier IQR method, emphasizing the importance of utilizing multiple techniques for a more comprehensive understanding. The visualizations accentuate these outliers, allowing for a clear distinction from the main distribution of the data. Such discrepancies underscore the necessity of domain expertise and context when deciding on the best approach for outlier management.

To enhance your understanding and facilitate further analysis, it's valuable to compile a comprehensive list of identified outliers. This list provides a clear overview of the specific data points that deviate significantly from the norm. In the following section, you'll illustrate how to systematically organize and list these outliers into a DataFrame for each feature: "Lot Area," "Sales Price," and "Total Rooms Above Ground." This tabulated format allows for easy inspection and potential actions, such as further investigation or targeted data treatment.

Let's explore the approach that accomplishes this task.

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')
features = ['LotArea', 'SalePrice', 'TotRmsAbvGrd']

# Define a function to tabulate outliers into a DataFrame
def outliers_dataframes_gaussian(dataframe, features, threshold=3, num_rows=None):
    outliers_dataframes = {}

    for feature in features:
        data = dataframe[feature]
        mean = data.mean()
        std_dev = data.std()
        outliers = data[(data < mean - threshold * std_dev) |
                        (data > mean + threshold * std_dev)]

        # Create a new DataFrame for outliers of the current feature
        outliers_df = dataframe.loc[outliers.index, [feature]].copy()
        outliers_df.rename(columns={feature: 'Outlier Value'}, inplace=True)
        outliers_df['Feature'] = feature
        outliers_df.reset_index(inplace=True)

        # Display specified number of rows (default: full dataframe)
        if num_rows:
            outliers_df = outliers_df.head(num_rows)

        outliers_dataframes[feature] = outliers_df

    return outliers_dataframes

# Example usage with user-defined number of rows = 7
outliers_gaussian_dataframes = outliers_dataframes_gaussian(Ames, features, num_rows=7)

# Print each DataFrame with the original format and capitalized 'index'
```

```
for feature, df in outliers_gaussian_dataframes.items():
    df_reset = df.reset_index().rename(columns={'index': 'Index'})
    print(f"Outliers for {feature}:\n", df_reset[['Index', 'Feature', 'Outlier Value']])
    print()
```

Listing 13.4: Enumerating outlier values based in Gaussian model

Now, before revealing the results, it's essential to note that the code snippet allows user customization. By adjusting the parameter `num_rows`, you have the flexibility to define the number of rows you want to see in each DataFrame. In the example shared earlier, you used `num_rows=7` for a concise display, but the default setting is `num_rows=None`, which prints the entire DataFrame. Feel free to adjust this parameter to suit your preferences and the specific requirements of your analysis.

```
Outliers for LotArea:
    Index  Feature  Outlier Value
0     104  LotArea          53107
1     195  LotArea          53227
2     249  LotArea         159000
3     309  LotArea          40094
4     329  LotArea          45600
5     347  LotArea          50271
6     355  LotArea         215245

Outliers for SalePrice:
    Index    Feature  Outlier Value
0      29  SalePrice         450000
1      65  SalePrice         615000
2     103  SalePrice         468000
3     108  SalePrice         500067
4     124  SalePrice         475000
5     173  SalePrice         423000
6     214  SalePrice         500000

Outliers for TotRmsAbvGrd:
    Index       Feature  Outlier Value
0      50  TotRmsAbvGrd             12
1     165  TotRmsAbvGrd             11
2     244  TotRmsAbvGrd             11
3     309  TotRmsAbvGrd             11
4     407  TotRmsAbvGrd             11
5     424  TotRmsAbvGrd             13
6     524  TotRmsAbvGrd             11
```

Output 13.3: Outlier values in three feature columns

In this exploration of probabilistic and statistical models for outlier detection, you focused on the Gaussian model applied to the Ames Housing Dataset, specifically utilizing a threshold of three standard deviations. By leveraging the insights provided by visualizations and statistical methods, you identified outliers and demonstrated their listing in a customizable DataFrame.

13.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Online

Outlier. Wikipedia.
https://en.wikipedia.org/wiki/Outlier

"What are outliers in the data?" In: *Engineering Statistics Handbook*. Section 7.1.6. NIST, 2012.
https://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm

13.5 Summary

Outliers, stemming from diverse causes, significantly impact statistical analyses. Recognizing their origins is crucial as they can distort visualizations, central tendency measures, and statistical tests. Classical Data Science methods for outlier detection encompass visual, statistical, and probabilistic approaches, with the choice dependent on the nature of the dataset and specific problems.

Application of these methods on the Ames Housing Dataset, focusing on Lot Area, Sales Price, and Total Rooms Above Ground, revealed insights. Visual methods like box plots provided quick outlier identification. The interquartile range (IQR) method quantified outliers, revealing 113, 116, and 35 outliers for Lot Area, Sales Price, and Total Rooms Above Ground. Probabilistic models, particularly the Gaussian model with three standard deviations, found 24, 42, and 35 outliers in the respective features.

These results underscore the need for a multifaceted approach to outlier detection. Beyond identification, systematically organizing and listing outliers in tabulated DataFrames facilitates in-depth inspection. Customizability, demonstrated by the `num_rows` parameter, ensures flexibility in presenting tailored results. In conclusion, this exploration enhances understanding and provides practical guidance for managing outliers in real-world datasets.

Specifically, you learned:

- The significance of outliers and their potential impact on data analyses.
- Various traditional methods used in Data Science for outlier detection.
- How to apply these methods in a real-world dataset, using the Ames Housing Dataset as an example.
- Systematic organization and listing of identified outliers into customizable DataFrames for detailed inspection and further analysis.

Careful readers should wonder in Figure 13.2, the distributions do not look like Gaussian while you assumed they are. If that's a concern, the next chapter tells you how to transform features into Gaussian (as new features) so that analyses can be applied.

Skewness Be Gone: Transformative Tricks for Data Scientists

14

Data transformations enable data scientists to refine, normalize, and standardize raw data into a format ripe for analysis. These transformations are not merely procedural steps; they are essential in mitigating biases, handling skewed distributions, and enhancing the robustness of statistical models. This chapter will primarily focus on how to address skewed data. By focusing on the "SalePrice" and "YearBuilt" attributes from the Ames housing dataset, you will see examples of positive and negative skewed data and ways to normalize their distributions using transformations.

Let's get started.

Overview

This chapter is divided into five parts; they are:

- ▷ Understanding Skewness and the Need for Transformation
- ▷ Strategies for Taming Positive Skewness
- ▷ Strategies for Taming Negative Skewness
- ▷ Statistical Evaluation of Transformations
- ▷ Choosing the Right Transformation

14.1 Understanding Skewness and the Need for Transformation

Skewness is a statistical measure that describes the asymmetry of a data distribution around its mean. In simpler terms, it indicates whether the bulk of the data is bunched up on one side of the scale, leaving a long tail stretching out in the opposite direction. There are two types of skewness you encounter in data analysis:

- ▷ *Positive Skewness*: This occurs when the tail of the distribution extends towards higher values, on the right side of the peak. The majority of data points are clustered at the lower end of the scale, indicating that while most values are relatively low, there are a few exceptionally high values. The "SalePrice" attribute in the Ames dataset

exemplifies positive skewness, as most homes sell at lower prices, but a small number sell at significantly higher prices.

- ▷ *Negative Skewness*: Conversely, negative skewness happens when the tail of the distribution stretches towards lower values, on the left side of the peak. In this scenario, the data is concentrated towards the higher end of the scale, with fewer values trailing off into lower numbers. The "YearBuilt" feature of the Ames dataset is a perfect illustration of negative skewness, suggesting that while a majority of houses were built in more recent years, a smaller portion dates back to earlier times.

As a reference, the normal distribution has zero skewness. If you want to use a model that depends on normal distribution, such as defining outliers as those data points located beyond the second standard deviation from the mean, you need to transform the data by "fixing" the skewness before determining the boundary of outlier, then transform it back to the original distribution.

To better grasp these concepts, let's visualize the skewness.

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

# Load the dataset
Ames = pd.read_csv('Ames.csv')

# Calculate skewness
sale_price_skew = Ames['SalePrice'].skew()
year_built_skew = Ames['YearBuilt'].skew()

# Set the style of seaborn
sns.set(style='whitegrid')

# Create a figure for 2 subplots (1 row, 2 columns)
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Plot for SalePrice (positively skewed)
sns.histplot(Ames['SalePrice'], kde=True, ax=ax[0], color='skyblue')
ax[0].set_title('Distribution of SalePrice (Positive Skew)', fontsize=16)
ax[0].set_xlabel('SalePrice')
ax[0].set_ylabel('Frequency')

# Annotate Skewness
ax[0].text(0.5, 0.5, f'Skew: {sale_price_skew:.2f}', transform=ax[0].transAxes,
           horizontalalignment='right', color='black', weight='bold',
           fontsize=14)

# Plot for YearBuilt (negatively skewed)
sns.histplot(Ames['YearBuilt'], kde=True, ax=ax[1], color='salmon')
ax[1].set_title('Distribution of YearBuilt (Negative Skew)', fontsize=16)
ax[1].set_xlabel('YearBuilt')
ax[1].set_ylabel('Frequency')

# Annotate Skewness
```

```
ax[1].text(0.5, 0.5, f'Skew: {year_built_skew:.2f}', transform=ax[1].transAxes,
           horizontalalignment='right', color='black', weight='bold',
           fontsize=14)

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

Listing 14.1: Visualizing the skewness

For "SalePrice," the graph shows a pronounced right-skewed distribution, highlighting the challenge of skewness in data analysis. Such distributions can complicate predictive modeling and obscure insights, making it difficult to draw accurate conclusions. In contrast, "YearBuilt" demonstrates negative skewness, where the distribution reveals that newer homes predominate, with older homes forming the long tail to the left.

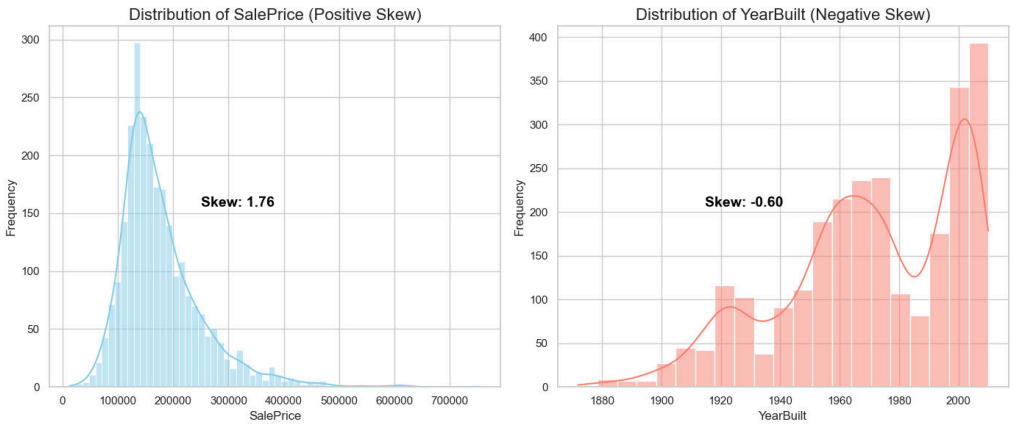

Figure 14.1: Histograms of sales price and built year

Addressing skewness through data transformation is not merely a statistical adjustment; it is a crucial step toward uncovering precise, actionable insights. By applying transformations, you aim to mitigate the effects of skewness, facilitating more reliable and interpretable analyses. This normalization process enhances your ability to conduct meaningful data science, beyond just meeting statistical prerequisites. It underscores your commitment to improving the clarity and utility of your data, setting the stage for insightful, impactful findings in your subsequent explorations of data transformation.

14.2 Strategies for Taming Positive Skewness

To combat positive skewness, you can use five key transformations: Log, square root, Box-Cox, Yeo-Johnson, and quantile transformations. Each method aims to mitigate skewness, enhancing the suitability of data for further analysis.

Log Transformation This method is particularly suited for right-skewed data, effectively minimizing large-scale differences by taking the natural log of all data points. This compression of the data range makes it more amenable to further statistical analysis.

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

Ames = pd.read_csv('Ames.csv')

# Applying Log Transformation
Ames['Log_SalePrice'] = np.log(Ames['SalePrice'])
print(f"Skewness after Log Transformation: {Ames['Log_SalePrice'].skew():.5f}")
```

Listing 14.2: Applying log transform

You can see that the skewness is reduced:

```
Skewness after Log Transformation: 0.04172
```

Output 14.1: Skewness after log transform

Square Root Transformation A softer approach than the log transformation, ideal for moderately skewed data. Applying the square root to each data point reduces skewness and diminishes the impact of outliers, making the distribution more symmetric.

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

Ames = pd.read_csv('Ames.csv')

# Applying Square Root Transformation
Ames['Sqrt_SalePrice'] = np.sqrt(Ames['SalePrice'])
print(f"Skewness after Square Root Transformation: {Ames['Sqrt_SalePrice'].skew():.5f}")
```

Listing 14.3: Applying square root transform

This prints:

```
Skewness after Square Root Transformation: 0.90148
```

Output 14.2: Skewness after square root transform

Box-Cox Transformation Offers flexibility by optimizing the transformation parameter lambda (λ), but applicable only to positive data. Mathematically, it transforms value y into

$$y' = \frac{y^\lambda - 1}{\lambda}$$

except when $\lambda = 0$, that will be $y' = \log(y)$. The Box-Cox method in SciPy systematically finds the best value of λ to reduce skewness and stabilize variance, enhancing the normality of data.

```python
import pandas as pd
import scipy.stats

Ames = pd.read_csv('Ames.csv')
```

```
# Applying Box-Cox Transformation after checking all values are positive
if (Ames['SalePrice'] > 0).all():
    Ames['BoxCox_SalePrice'], lmbda = scipy.stats.boxcox(Ames['SalePrice'])
else:
    # Consider alternative transformations or handling strategies
    print("Not all SalePrice values are positive.")
    print("Consider using Yeo-Johnson or handling negative values.")
print(f"Skewness after Box-Cox Transformation: {Ames['BoxCox_SalePrice'].skew():.5f}")
```

Listing 14.4: Applying Box-Cox transform

This is the best transformation so far because the skewness is very close to zero:

```
Skewness after Box-Cox Transformation: -0.00436
```

Output 14.3: Skewness after Box-Cox transform

Yeo-Johnson Transformation The above transformations only work with positive data. Yeo-Johnson is similar to Box-Cox but adaptable to both positive and non-positive data. It modifies the data through an optimal transformation parameter. This adaptability allows it to manage skewness across a wider range of data values, improving its fit for statistical models.

```
import pandas as pd
import scipy.stats

Ames = pd.read_csv('Ames.csv')

# Applying Yeo-Johnson Transformation
Ames['YeoJohnson_SalePrice'], _ = scipy.stats.yeojohnson(Ames['SalePrice'])
print("Skewness after Yeo-Johnson Transformation: "
      f"{Ames['YeoJohnson_SalePrice'].skew():.5f}")
```

Listing 14.5: Applying Yeo-Johnson transform

Similar to Box-Cox, the skewness after transformation is very close to zero:

```
Skewness after Yeo-Johnson Transformation: -0.00437
```

Output 14.4: Skewness after Yeo-Johnson transform

Quantile Transformation Quantile transformation maps data to a specified distribution, such as normal, effectively addressing skewness by redistributing the data points evenly across the chosen distribution. This transformation normalizes the shape of the data, focusing on making the distribution more uniform or Gaussian-like without assuming it will directly benefit linear models due to its nonlinear nature and the challenge of reverting the data to its original form.

```
import pandas as pd
from sklearn.preprocessing import QuantileTransformer

Ames = pd.read_csv('Ames.csv')

# Applying Quantile Transformation to follow a normal distribution
quantile_transformer = QuantileTransformer(output_distribution='normal', random_state=0)
Ames['Quantile_SalePrice'] = \
    quantile_transformer.fit_transform(Ames['SalePrice'].values.reshape(-1, 1)).flatten()
print(f"Skewness after Quantile Transformation: {Ames['Quantile_SalePrice'].skew():.5f}")
```

Listing 14.6: Applying quantile transform

Because this transformation fits the data into the Gaussian distribution by brute force, the skewness is closest to zero:

```
Skewness after Quantile Transformation: 0.00286
```

Output 14.5: Skewness after quantile transform

To illustrate the effects of these transformations, let's take a look at the visual representation of the "SalePrice" distribution before and after each method is applied.

```
import pandas as pd
import numpy as np
import scipy.stats
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import QuantileTransformer

Ames = pd.read_csv('Ames.csv')
Ames['Log_SalePrice'] = np.log(Ames['SalePrice'])
Ames['Sqrt_SalePrice'] = np.sqrt(Ames['SalePrice'])
Ames['BoxCox_SalePrice'], _ = scipy.stats.boxcox(Ames['SalePrice'])
Ames['YeoJohnson_SalePrice'], _ = scipy.stats.yeojohnson(Ames['SalePrice'])
quantile_transformer = QuantileTransformer(output_distribution='normal', random_state=0)
Ames['Quantile_SalePrice'] = \
    quantile_transformer.fit_transform(Ames['SalePrice'].values.reshape(-1, 1)).flatten()

# Plotting the distributions
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Flatten the axes array for easier indexing
axes = axes.flatten()

# Hide unused subplot axes
for ax in axes[6:]:
    ax.axis('off')

# Original SalePrice Distribution
sns.histplot(Ames['SalePrice'], kde=True, bins=30, color='skyblue', ax=axes[0])
axes[0].set_title('Original SalePrice Distribution (Skew: 1.76)')
axes[0].set_xlabel('SalePrice')
axes[0].set_ylabel('Frequency')
```

```
# Log Transformed SalePrice
sns.histplot(Ames['Log_SalePrice'], kde=True, bins=30, color='blue', ax=axes[1])
axes[1].set_title('Log Transformed SalePrice (Skew: 0.04172)')
axes[1].set_xlabel('Log of SalePrice')
axes[1].set_ylabel('Frequency')

# Square Root Transformed SalePrice
sns.histplot(Ames['Sqrt_SalePrice'], kde=True, bins=30, color='orange', ax=axes[2])
axes[2].set_title('Square Root Transformed (Skew: 0.90148)')
axes[2].set_xlabel('Square Root of SalePrice')
axes[2].set_ylabel('Frequency')

# Box-Cox Transformed SalePrice
sns.histplot(Ames['BoxCox_SalePrice'], kde=True, bins=30, color='red', ax=axes[3])
axes[3].set_title('Box-Cox Transformed SalePrice (Skew: -0.00436)')
axes[3].set_xlabel('Box-Cox of SalePrice')
axes[3].set_ylabel('Frequency')

# Yeo-Johnson Transformed SalePrice
sns.histplot(Ames['YeoJohnson_SalePrice'], kde=True, bins=30, color='purple', ax=axes[4])
axes[4].set_title('Yeo-Johnson Transformed (Skew: -0.00437)')
axes[4].set_xlabel('Yeo-Johnson of SalePrice')
axes[4].set_ylabel('Frequency')

# Quantile Transformed SalePrice (Normal Distribution)
sns.histplot(Ames['Quantile_SalePrice'], kde=True, bins=30, color='green', ax=axes[5])
axes[5].set_title('Quantile Transformed (Normal Distn, Skew: 0.00286)')
axes[5].set_xlabel('Quantile Transformed SalePrice')
axes[5].set_ylabel('Frequency')

plt.tight_layout(pad=4.0)
plt.show()
```

Listing 14.7: Histogram to show the skewness after transforms

The following visual provides a side-by-side comparison, helping you to understand better the influence of each transformation on the distribution of housing prices.

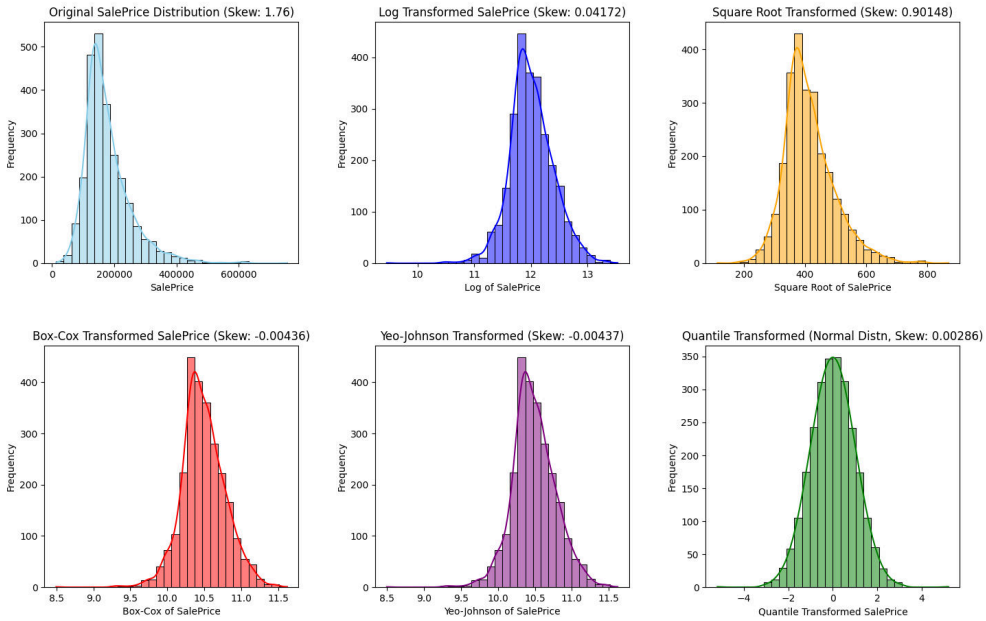

Figure 14.2: Distribution of data after transformation

This visual serves as a clear reference for how each transformation method alters the distribution of "SalePrice," demonstrating the resulting effect towards achieving a more normal distribution.

14.3 Strategies for Taming Negative Skewness

To combat negative skewness, you can use the five key transformations: Squared, cubed, Box-Cox, Yeo-Johnson, and quantile transformations. Each method aims to mitigate skewness, enhancing the suitability of data for further analysis.

Squared Transformation This involves taking each data point in the dataset and squaring it (i.e., raising it to the power of 2). The squared transformation is useful for reducing negative skewness because it tends to spread out the lower values more than the higher values. However, it's more effective when *all data points are positive* and the degree of negative skewness is not extreme.

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

# Applying Squared Transformation
Ames['Squared_YearBuilt'] = Ames['YearBuilt'] ** 2
print(f"Skewness after Squared Transformation: {Ames['Squared_YearBuilt'].skew():.5f}")
```

Listing 14.8: Applying squared transform

It prints:

```
Skewness after Squared Transformation: -0.57207
```

Output 14.6: Skewness after squared transform

Cubed Transformation Similar to the squared transformation but involves raising each data point to the power of 3. The cubed transformation can further reduce negative skewness, especially in cases where the squared transformation is insufficient. It's *more aggressive* in spreading out values, which can benefit more negatively skewed distributions.

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

# Applying Cubed Transformation
Ames['Cubed_YearBuilt'] = Ames['YearBuilt'] ** 3
print(f"Skewness after Cubed Transformation: {Ames['Cubed_YearBuilt'].skew():.5f}")
```

Listing 14.9: Applying cubed transform

It prints:

```
Skewness after Cubed Transformation: -0.54539
```

Output 14.7: Skewness after cubed transform

Box-Cox Transformation Same as the case of transforming positively skewed data, you can use Box-Cox for negatively skewed data, and only for positive-valued data. For negatively skewed data, a positive lambda is often found, applying a transformation that effectively reduces skewness.

```
import pandas as pd
import scipy.stats

Ames = pd.read_csv('Ames.csv')

# Applying Box-Cox Transformation after checking all values are positive
if (Ames['YearBuilt'] > 0).all():
    Ames['BoxCox_YearBuilt'], _ = scipy.stats.boxcox(Ames['YearBuilt'])
else:
    # Consider alternative transformations or handling strategies
    print("Not all YearBuilt values are positive.")
    print("Consider using Yeo-Johnson or handling negative values.")
print(f"Skewness after Box-Cox Transformation: {Ames['BoxCox_YearBuilt'].skew():.5f}")
```

Listing 14.10: Applying Box-Cox transform

You can see the skewness is moved closer to zero than before:

```
Skewness after Box-Cox Transformation: -0.12435
```

Output 14.8: Skewness after Box-Cox transform

Yeo-Johnson Transformation As mentioned earlier, Yeo-Johnson is designed to handle both positive and negative data. It adjusts the data in a way that reduces skewness, making it particularly versatile for datasets with a mix of positive and negative values.

```
import pandas as pd
import scipy.stats

Ames = pd.read_csv('Ames.csv')

# Applying Yeo-Johnson Transformation
Ames['YeoJohnson_YearBuilt'], _ = scipy.stats.yeojohnson(Ames['YearBuilt'])
print("Skewness after Yeo-Johnson Transformation: "
      f"{Ames['YeoJohnson_YearBuilt'].skew():.5f}")
```

Listing 14.11: Applying Yeo-Johnson transform

Similar to Box-Cox, you get a skewness moved closer to zero:

```
Skewness after Yeo-Johnson Transformation: -0.12435
```

Output 14.9: Skewness after Yeo-Johnson transform

Quantile Transformation When applied to negatively skewed data, the quantile transformation can effectively normalize the distribution. It's particularly useful for dealing with outliers and making the distribution of the data uniform or normal, regardless of the original skewness.

```
import pandas as pd
from sklearn.preprocessing import QuantileTransformer

Ames = pd.read_csv('Ames.csv')

# Applying Quantile Transformation to follow a normal distribution
quantile_transformer = QuantileTransformer(output_distribution='normal', random_state=0)
Ames['Quantile_YearBuilt'] = \
    quantile_transformer.fit_transform(Ames['YearBuilt'].values.reshape(-1, 1)).flatten()
print(f"Skewness after Quantile Transformation: {Ames['Quantile_YearBuilt'].skew():.5f}")
```

Listing 14.12: Applying quantile transform

As you saw before in the case of positive skewness, quantile transformation provides the best result in the sense that the resulting skewness is closest to zero:

```
Skewness after Quantile Transformation: 0.02713
```

Output 14.10: Skewness after quantile transform

To illustrate the effects of these transformations, let's take a look at the visual representation of the "YearBuilt" distribution before and after each method is applied.

```
import pandas as pd
import scipy.stats
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import QuantileTransformer

Ames = pd.read_csv('Ames.csv')
Ames['Squared_YearBuilt'] = Ames['YearBuilt'] ** 2
Ames['Cubed_YearBuilt'] = Ames['YearBuilt'] ** 3
Ames['BoxCox_YearBuilt'], _ = scipy.stats.boxcox(Ames['YearBuilt'])
Ames['YeoJohnson_YearBuilt'], _ = scipy.stats.yeojohnson(Ames['YearBuilt'])
quantile_transformer = QuantileTransformer(output_distribution='normal', random_state=0)
Ames['Quantile_YearBuilt'] = \
    quantile_transformer.fit_transform(Ames['YearBuilt'].values.reshape(-1, 1)).flatten()

# Plotting the distributions
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Flatten the axes array for easier indexing
axes = axes.flatten()

# Original YearBuilt Distribution
sns.histplot(Ames['YearBuilt'], kde=True, bins=30, color='skyblue', ax=axes[0])
axes[0].set_title(f'Original YearBuilt Distr. (Skew: {Ames["YearBuilt"].skew():.5f})')
axes[0].set_xlabel('YearBuilt')
axes[0].set_ylabel('Frequency')

# Squared YearBuilt
sns.histplot(Ames['Squared_YearBuilt'], kde=True, bins=30, color='blue', ax=axes[1])
axes[1].set_title(f'Squared YearBuilt (Skew: {Ames["Squared_YearBuilt"].skew():.5f})')
axes[1].set_xlabel('Squared YearBuilt')
axes[1].set_ylabel('Frequency')

# Cubed YearBuilt
sns.histplot(Ames['Cubed_YearBuilt'], kde=True, bins=30, color='orange', ax=axes[2])
axes[2].set_title(f'Cubed YearBuilt (Skew: {Ames["Cubed_YearBuilt"].skew():.5f})')
axes[2].set_xlabel('Cubed YearBuilt')
axes[2].set_ylabel('Frequency')

# Box-Cox Transformed YearBuilt
sns.histplot(Ames['BoxCox_YearBuilt'], kde=True, bins=30, color='red', ax=axes[3])
axes[3].set_title(f'Box-Cox Transformed (Skew: {Ames["BoxCox_YearBuilt"].skew():.5f})')
axes[3].set_xlabel('Box-Cox YearBuilt')
axes[3].set_ylabel('Frequency')

# Yeo-Johnson Transformed YearBuilt
sns.histplot(Ames['YeoJohnson_YearBuilt'], kde=True, bins=30, color='purple', ax=axes[4])
axes[4].set_title('Yeo-Johnson Transformed (Skew: '
                  f'{Ames["YeoJohnson_YearBuilt"].skew():.5f})')
axes[4].set_xlabel('Yeo-Johnson YearBuilt')
axes[4].set_ylabel('Frequency')

# Quantile Transformed YearBuilt (Normal Distribution)
sns.histplot(Ames['Quantile_YearBuilt'], kde=True, bins=30, color='green', ax=axes[5])
```

```
axes[5].set_title('Quantile Transformed (Normal Dist., '
                  f'Skew: {Ames["Quantile_YearBuilt"].skew():.5f})')
axes[5].set_xlabel('Quantile Transformed YearBuilt')
axes[5].set_ylabel('Frequency')

plt.tight_layout(pad=4.0)
plt.show()
```

Listing 14.13: Histogram to show the skewness after transforms

The following visual provides a side-by-side comparison, helping you to better understand the influence of each transformation on this feature.

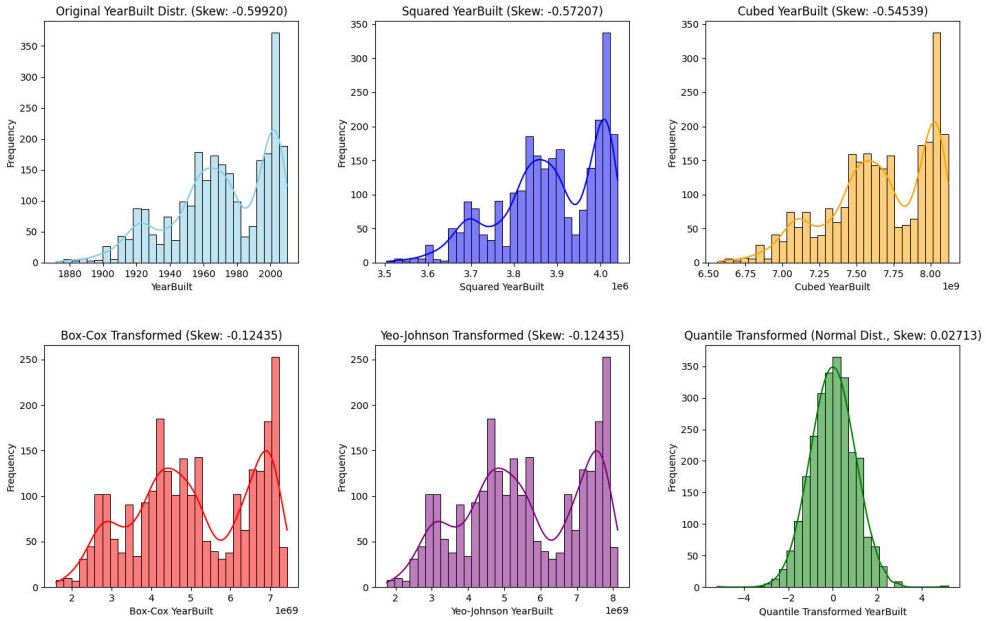

Figure 14.3: Distribution of data after transformation

This visual provides a clear reference for how each transformation method alters the distribution of "YearBuilt," demonstrating the resulting effect towards achieving a more normal distribution.

14.4 Statistical Evaluation of Transformations

How do you know the transformed data matches the normal distribution?

The Kolmogorov-Smirnov (KS) test is a *nonparametric test* used to determine if a sample comes from a population with a specific distribution. Unlike parametric tests, which assume a specific distribution form for the data (usually normal distribution), nonparametric tests make no such assumptions. This quality makes them highly useful in the context of data transformations because it helps to assess how closely a transformed dataset approximates a normal distribution. The KS test compares the cumulative distribution function (CDF) of the

sample data against the CDF of a known distribution (in this case, the normal distribution), providing a test statistic that quantifies the distance between the two.

Null and Alternate Hypothesis:

- ▷ *Null Hypothesis (H_0)*: The data follows the specified distribution (normal distribution, in this case).
- ▷ *Alternate Hypothesis (H_1)*: The data does not follow the specified distribution.

In this context, the KS test is used to evaluate the goodness of fit between the empirical distribution of the transformed data and the normal distribution. The test statistic is a measure of the largest discrepancy between the empirical (transformed data) and theoretical CDFs (normal distribution). A small test statistic suggests that the distributions are similar.

```
...
import scipy.stats

# Run the KS tests for the 10 cases
transformations = ["Log_SalePrice", "Sqrt_SalePrice", "BoxCox_SalePrice",
                   "YeoJohnson_SalePrice", "Quantile_SalePrice",
                   "Squared_YearBuilt", "Cubed_YearBuilt", "BoxCox_YearBuilt",
                   "YeoJohnson_YearBuilt", "Quantile_YearBuilt"]

# Standardizing the transformations before performing KS test
ks_test_results = {}
for transformation in transformations:
    standardized_data = \
        (Ames[transformation] - Ames[transformation].mean()) / Ames[transformation].std()
    ks_stat, ks_p_value = scipy.stats.kstest(standardized_data, 'norm')
    ks_test_results[transformation] = (ks_stat, ks_p_value)

# Convert results to DataFrame for easier comparison
ks_test_results_df = pd.DataFrame.from_dict(ks_test_results, orient='index',
                                            columns=['KS Statistic', 'P-Value'])
print(ks_test_results_df.round(5))
```

Listing 14.14: Running KS test to verify goodness-of-fit to normal distribution

The code above prints a table as follows:

```
                     KS Statistic  P-Value
Log_SalePrice             0.04261  0.00017
Sqrt_SalePrice            0.07689  0.00000
BoxCox_SalePrice          0.04294  0.00014
YeoJohnson_SalePrice      0.04294  0.00014
Quantile_SalePrice        0.00719  0.99924
Squared_YearBuilt         0.11661  0.00000
Cubed_YearBuilt           0.11666  0.00000
BoxCox_YearBuilt          0.11144  0.00000
YeoJohnson_YearBuilt      0.11144  0.00000
Quantile_YearBuilt        0.02243  0.14717
```

Output 14.11: KS statistic and the corresponding p-value

You can see that the higher the KS statistic, the lower the p-value. Respectively,

- ▷ *KS Statistic*: This represents the maximum difference between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. Smaller values indicate a closer fit to the normal distribution.
- ▷ *P-Value*: Provides the probability of observing the test results under the null hypothesis. A low p-value (typically <0.05) rejects the null hypothesis, indicating the sample distribution significantly differs from the normal distribution.

The quantile transformation of "SalePrice" yielded the most promising results, with a KS statistic of 0.00719 and a p-value of 0.99924, indicating that after this transformation, the distribution closely aligns with the normal distribution. It is not surprising because quantile transformation is designed to produce a good fit for normal distribution. The p-value is significant because a higher p-value (close to 1) suggests that the null hypothesis (that the sample comes from a specified distribution) cannot be rejected, implying good normality.

Other transformations like log, Box-Cox, and Yeo-Johnson also improved the distribution of "SalePrice" but to a lesser extent, as reflected by their lower p-values (ranging from 0.00014 to 0.00017), indicating less conformity to the normal distribution compared to the quantile transformation. The transformations applied to "YearBuilt" showed generally less effectiveness in achieving normality compared to "SalePrice." The BoxCox and YeoJohnson transformations offered slight improvements over Squaring and Cubing, as seen in their slightly lower KS statistics and p-values, but still indicated significant deviations from normality. The quantile transformation for "YearBuilt" showed a more favorable outcome with a KS statistic of 0.02243 and a p-value of 0.14717, suggesting a moderate improvement towards normality, although not as pronounced as the effect seen with "SalePrice."

14.5 Choosing the Right Transformation

There is no one-size-fits-all transformation for addressing skewness in data; it requires careful consideration of the context and characteristics of the data at hand. The importance of context in selecting the appropriate transformation method cannot be overstated. Here are key factors to consider:

- ▷ *Data Characteristics*: The nature of the data (e.g., the presence of zeros or negative values) can limit the applicability of certain transformations. For instance, log transformations cannot be directly applied to zero or negative values without adjustments.
- ▷ *Degree of Skewness*: The extent of skewness in the data influences the choice of transformation. More severe skewness might require more potent transformations (e.g., Box-Cox or Yeo-Johnson) compared to milder skewness, which might be adequately addressed with log or square root transformations.
- ▷ *Statistical Properties*: The transformation chosen should ideally improve the statistical properties of the dataset, such as normalizing the distribution and stabilizing variance, which are essential for many statistical tests and models.
- ▷ *Interpretability*: The ease of interpreting results after transformation is crucial. Some transformations, like log or square root, allow for relatively straightforward

interpretation, whereas others, like the quantile transformation, might complicate the interpretation of the original scale.

- ▷ *Objective of Analysis*: The ultimate goal of the analysis—whether it's predictive modeling, hypothesis testing, or exploratory analysis—plays a critical role in selecting the transformation method. The transformation should align with the analytical techniques and models to be employed later.

In summary, the choice of the right transformation depends on multiple factors, including a solid understanding of the dataset, the specific goals of the analysis, and the practical implications for model interpretability and performance. No method is universally superior; each has its trade-offs and applicability depending on the scenario.

It's important to highlight a cautionary note regarding the quantile transformation, which your visual and statistical tests identified as highly effective in achieving a normal distribution. While potent, the quantile transformation is not a linear transformation like the others. This means it can significantly alter the structure of data in ways that are not easily reversible, potentially complicating the interpretation of results and preventing back-transformation of values to the original scale. Therefore, despite its effectiveness in normalization, its use should be considered carefully, especially in cases where maintaining a connection to the original data scale is important or where the interpretability of the model is a priority. In most scenarios, the preference might lean towards transformations that balance normalization effectiveness with simplicity and reversibility, ensuring that the data remains as interpretable and manageable as possible.

14.6 Further Reading

This section provides more resources on the topic if you want to go deeper.

Papers

In-Kwon Yeo and Richard A. Johnson. "A New Family of Power Transformations to Improve Normality or Symmetry". *Biometrika*, 87(4), Dec. 2000, pp. 954–959.

Online

James Chen. *Right Skewed vs. Left Skewed Distribution*. Investopedia. Mar. 2024.
https://www.investopedia.com/terms/s/skewness.asp

`scipy.stats.boxcox` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html

`scipy.stats.yeojohnson` *API*. SciPy.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.yeojohnson.html

`sklearn.preprocessing.QuantileTransformer` *API*. SciPy.
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.QuantileTransformer.html

`scipy.stats.kstest` *API*. SciPy.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kstest.html

14.7 Summary

In this chapter, you've embarked on a detailed exploration of data transformations, focusing on their critical role in addressing skewed data within the field of data science. Through practical examples using the "SalePrice" and "YearBuilt" features from the Ames housing dataset, you demonstrated various transformation techniques—log, square root, Box-Cox, Yeo-Johnson, and quantile transformations—and their impact on normalizing data distributions. Your analysis underscores the necessity of selecting appropriate transformations based on data characteristics, the degree of skewness, statistical goals, interpretability, and the specific objectives of the analysis.

Specifically, you learned:

- The significance of data transformations and how they can handle skewed distributions.
- How to compare the effectiveness of different transformations through visual and statistical assessments.
- The importance of evaluating data characteristics, the severity of skewness, and analytical objectives to choose the most suitable transformation technique.

One reason to study the housing market dataset is to invest in it. In the next chapter, you will see how you can find an investment that you should consider or some that you should avoid.

Finding Value with Data: The Cohesive Force Behind Luxury Real Estate Decisions

15

The real estate industry is a vast network of stakeholders including agents, homeowners, investors, developers, municipal planners, and tech innovators, each bringing unique perspectives and objectives to the table. Within this intricate ecosystem, data emerges as the critical element that binds these diverse interests together, facilitating collaboration and innovation. PropTech, or Property Technology, illustrates this synergy by applying information technology to real estate, transforming how properties are researched, bought, sold, and managed through the power of data science.

From its nascent stages in the digitization of property listings to the current landscape that includes virtual reality, IoT-enabled smart homes, and blockchain-enhanced transactions, the journey of PropTech reflects an industry increasingly driven by data. This evolution is marked not by the technologies but by how data science harnesses information to streamline operations, enhance user experiences, and introduce efficiencies at every turn.

At the core of the transformative impact of PropTech is data science, which excels in making sense of vast datasets to uncover actionable insights. It equips all players in the real estate domain—from developers optimizing project locations to municipal planners shaping sustainable urban environments—with the tools to make decisions rooted in solid data analysis. Data science lays the groundwork for strategic planning and operational improvements across the real estate sector through sophisticated data management and descriptive analytics.

Exploring the application of data science for luxury homebuyers, you see a specific example of the wider influence of PropTech. This narrative not only showcases the nuanced benefits of data-driven strategies in evaluating luxury properties but also underlines the fundamental role of data in fostering a more integrated, efficient, and consumer-responsive real estate industry.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ Folium: A Guide to Interactive Mapping
- ▷ Empowering Luxury Homebuyers with Data Science: Finding Value in the Market
- ▷ Visualizing Opportunities: Finding Accessible Luxury Real Estate

15.1 Folium: A Guide to Interactive Mapping

Folium is a powerful Python library that brings the world of geospatial data to life through interactive maps. Leveraging Leaflet.js, Folium allows users to create rich, dynamic visualizations with just a few lines of Python code, making it an invaluable tool for real estate professionals and data scientists alike. Its key strengths lie in its interactivity, allowing for zooming and clicking for details, and its seamless integration with the data science stack in Python, notably pandas, enabling sophisticated data-driven map visualizations with minimal effort.

For the real estate sector, Folium enables the visualization of property data against geographical contexts, offering insights into market trends, property valuations, and demographic overlays with unparalleled clarity. Folium maps can be easily saved as HTML files, facilitating effortless sharing with clients or stakeholders. This feature democratizes access to complex geospatial analyses, enhancing presentations, reports, and listings with interactive maps that provide a comprehensive view of the real estate landscape.

Folium is a package in PyPI. To install, you can use the following commands on your terminal or command-line interface:

```
pip install folium
```

Listing 15.1: Installing folium package for Python

Once you've successfully installed the required packages, you're ready to follow through with the examples below.

15.2 Empowering Luxury Homebuyers with Data Science: Finding Value in the Market

In the real estate market today, potential homeowners, especially those interested in the luxury segment, encounter the significant challenge of finding properties that not only satisfy their aesthetic and comfort preferences but also offer substantial value. This is where data science plays a pivotal role, transforming the daunting task of locating the perfect home into a journey rich with insights and discoveries.

The power of data science in real estate lies in its ability to analyze vast amounts of information, uncovering patterns, trends, and opportunities that might not be visible at first glance. For luxury homebuyers, this means the ability to identify properties that represent both opulence and value, ensuring their investment is as sound as it is beautiful.

Your first step is to identify the top 10 most expensive homes in Ames. This initial filter serves as your starting point, showcasing properties considered most valuable by the market. To do this, you'll use the pandas library in Python to load your dataset and perform the necessary analysis.

Here's the code that marks the beginning of your data-driven journey:

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

# Identify the 10 most expensive homes based on SalePrice with key features
top_10_expensive_homes = Ames.nlargest(10, 'SalePrice')
features = ['SalePrice', 'GrLivArea', 'OverallQual', 'KitchenQual', 'TotRmsAbvGrd',
            'Fireplaces']
top_10_df = top_10_expensive_homes[features]
print(top_10_df)
```

Listing 15.2: Find the most expensive homes

This block of code efficiently sifts through the Ames dataset to extract the 10 most expensive homes, focusing on key attributes that define luxury living: "SalePrice", "GrLivArea" (above grade living area), "OverallQual" (overall material and finish quality), "KitchenQual" (kitchen quality), "TotRmsAbvGrd" (total rooms above grade), and "Fireplaces." These features set the stage for differentiating between mere aesthetic appeal and true quality.

```
      SalePrice  GrLivArea  OverallQual KitchenQual  TotRmsAbvGrd  Fireplaces
1007     755000       4316           10          Ex            10           2
1325     625000       3627           10          Gd            10           1
65       615000       2470           10          Ex             7           2
584      611657       2364            9          Ex            11           2
1639     610000       2674           10          Ex             8           2
615      591587       2338            9          Gd             8           2
2087     584500       3500            9          Ex            11           1
1112     555000       2402           10          Ex            10           2
802      538000       3279            8          Ex            12           1
407      535000       2698           10          Ex            11           1
```

Output 15.1: The most expensive homes

To refine your search further, you apply specific criteria that embody the essence of luxury living. You focus on homes that boast an Overall Quality rating of 10, indicating the highest level of craftsmanship and materials. Additionally, you look for properties with exceptional kitchen quality (KitchenQual of "Ex") and a significant feature of luxury comfort: the presence of two or more fireplaces. These criteria help you sift through the top contenders to highlight properties that truly represent the pinnacle of luxury in Ames.

Let's execute the next block of code to filter the list of the top 10 most expensive homes down to those that meet these elite standards:

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')
top_10_expensive_homes = Ames.nlargest(10, 'SalePrice')
features = ['SalePrice', 'GrLivArea', 'OverallQual', 'KitchenQual', 'TotRmsAbvGrd',
            'Fireplaces']
top_10_df = top_10_expensive_homes[features]

# Refine the search with highest quality, excellent kitchen, and 2 fireplaces
```

```
elite = top_10_df.query('OverallQual == 10 & KitchenQual == "Ex" & Fireplaces >= 2')
print(elite)
```

Listing 15.3: Refining for high quality homes

```
      SalePrice  GrLivArea  OverallQual KitchenQual  TotRmsAbvGrd  Fireplaces
1007     755000       4316           10          Ex            10           2
65       615000       2470           10          Ex             7           2
1639     610000       2674           10          Ex             8           2
1112     555000       2402           10          Ex            10           2
```

Output 15.2: Short-listed DataFrame to match the query

This refined search narrows your focus to four elite properties that not only rank as some of the most expensive in Ames but also meet the stringent criteria for luxury.

As you delve deeper, it becomes imperative to introduce *feature engineering*. Feature engineering is the process of creating new variables or features from existing data that better represent the underlying problem. In your case, to enhance your understanding of real estate value. One such innovative feature is the *Price Per Square Foot (PSF)*. The PSF metric emerges as a critical tool in your analytical arsenal, offering a perspective on value beyond mere listing prices. By calculating the PSF for each property, you can compare properties more equitably, irrespective of their size or the absolute price. This measure illuminates the investment value per square foot of living space, providing a standardized scale for assessing the true worth of luxury properties.

Let's apply this calculation to your elite selection of homes:

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')
top_10_expensive_homes = Ames.nlargest(10, 'SalePrice')
features = ['SalePrice', 'GrLivArea', 'OverallQual', 'KitchenQual', 'TotRmsAbvGrd',
            'Fireplaces']
top_10_df = top_10_expensive_homes[features]
elite = top_10_df.query('OverallQual == 10 & KitchenQual == "Ex" & Fireplaces >= 2') \
                 .copy()

# Introduce PSF to rank the options
elite['PSF'] = elite['SalePrice']/elite['GrLivArea']
print(elite.sort_values(by='PSF'))
```

Listing 15.4: Adding PSF column

This action yields the following insights, allowing you to rank the properties by their relative value:

```
      SalePrice  GrLivArea  OverallQual KitchenQual  TotRmsAbvGrd  Fireplaces         PSF
1007     755000       4316           10          Ex            10           2  174.930491
1639     610000       2674           10          Ex             8           2  228.122663
1112     555000       2402           10          Ex            10           2  231.057452
65       615000       2470           10          Ex             7           2  248.987854
```

Output 15.3: Calculated PSF of the selected homes

Upon analyzing the PSF, it becomes evident that not all luxury homes are created equal. Despite being the most expensive on your list, the property with the lowest PSF offers the best value, underscoring the importance of this metric in evaluating luxury properties.

Now you can identify additional properties that embody the luxury standards and present exceptional value as defined by the PSF metric. By broadening your criteria to include all homes in the dataset with an Overall Quality rating of 10, excellent kitchen quality, and at least two fireplaces, but with a PSF under $175. The aim is to uncover homes that offer luxury at a more accessible price point.

Here's how you proceed with this expanded analysis:

```
import pandas as pd

Ames = pd.read_csv('Ames.csv')

# Cross check entire homes to search for better value
Ames['PSF'] = Ames['SalePrice']/Ames['GrLivArea']
value = Ames.query('PSF < 175 & OverallQual == 10 & KitchenQual == "Ex" & Fireplaces >=2')
print(value[['SalePrice', 'GrLivArea', 'OverallQual', 'KitchenQual', 'TotRmsAbvGrd',
             'Fireplaces', 'PSF']])
```

Listing 15.5: Finding the best value homes

This refined search yields intriguing results:

```
      SalePrice  GrLivArea  OverallQual KitchenQual  TotRmsAbvGrd  Fireplaces         PSF
1007     755000       4316           10          Ex            10           2  174.930491
2003     475000       3608           10          Ex            12           2  131.651885
```

Output 15.4: Result of home search

In this comprehensive search across the Ames dataset, you have uncovered two properties that not only embody the pinnacle of luxury with their superior amenities and craftsmanship but also stand as paragons of value within the luxury market. Remarkably, one of these homes presents a Price Per Square Foot (PSF) that significantly undercuts your established threshold, offering an exceptional opportunity for luxury homebuyers. This discovery underscores the potency of data science in real estate, enabling buyers to find homes that offer an extraordinary living experience and exceptional financial value.

Transitioning from numerical analysis to spatial visualization, let's turn to Folium to map these standout properties within the geographical tapestry of Ames. This next step will provide a visual context to your findings and illustrate the practical application of data science in enhancing the real estate selection process, making the journey towards finding the perfect luxury home both informed and visually engaging.

15.3 Visualizing Opportunities: Finding Accessible Luxury Real Estate

With the two standout properties identified, your next step leverages the power of Folium to bring these findings to life on an interactive map. This visualization not only situates each property within the geographical context of Ames but also enriches your analysis by allowing you to embed detailed information directly on the map.

Using Folium, you can create markers for each of these exceptional properties, providing potential buyers and stakeholders with a comprehensive overview at a glance. Each marker contains key data points about the property, including sale price, gross living area, quality ratings, and the calculated Price Per Square Foot (PSF), offering an intuitive and engaging way to explore these luxury homes. Below, you detail the process of adding this rich information to your map, ensuring each property is not just a point on a map, but a gateway to its unique story and value proposition.

```
import pandas as pd
import folium

Ames = pd.read_csv('Ames.csv')
Ames['PSF'] = Ames['SalePrice']/Ames['GrLivArea']
value = Ames.query('PSF < 175 & OverallQual == 10 & KitchenQual == "Ex" & Fireplaces >=2')

final_observation_indexes = value.index.tolist()

# Filter the dataset for these observations to get their latitude and longitude
final_locations = Ames.loc[final_observation_indexes, ['Latitude', 'Longitude']]

# Create a Folium map centered around the average location of the final observations
map_center = [final_locations['Latitude'].mean(), final_locations['Longitude'].mean()]
value_map = folium.Map(location=map_center, zoom_start=12)

# Add information to markers
for idx, row in final_locations.iterrows():
    # Extract additional information for the popup
    info = value.loc[idx, ['SalePrice', 'GrLivArea', 'OverallQual', 'KitchenQual',
                           'TotRmsAbvGrd', 'Fireplaces', 'PSF']]
    popup_text = f"""<b>Index:</b> {idx}<br>
                     <b>SalePrice:</b> {info['SalePrice']}<br>
                     <b>GrLivArea:</b> {info['GrLivArea']} sqft<br>
                     <b>OverallQual:</b> {info['OverallQual']}<br>
                     <b>KitchenQual:</b> {info['KitchenQual']}<br>
                     <b>TotRmsAbvGrd:</b> {info['TotRmsAbvGrd']}<br>
                     <b>Fireplaces:</b> {info['Fireplaces']}<br>
                     <b>PSF:</b> ${info['PSF']:.2f} /sqft"""
    folium.Marker([row['Latitude'], row['Longitude']],
                  popup=folium.Popup(popup_text, max_width=250)).add_to(value_map)

# Save the map to an HTML file on working directory
value_map.save('value_map.html')
```

Listing 15.6: Visualizing with Folium

Some HTML markups have been used in the string `popup_text` above to provide a floating text box over the map displayed. Executing this code will not only populate the Folium map with interactive markers but will also encapsulate the culmination of your data-driven quest within an HTML file, effortlessly saved to your working directory as `value_map.html`. This file serves as a tangible artifact, ready to be shared and explored further, inviting stakeholders to engage with your findings in an intuitive and dynamic format. Here is a static output of the file:

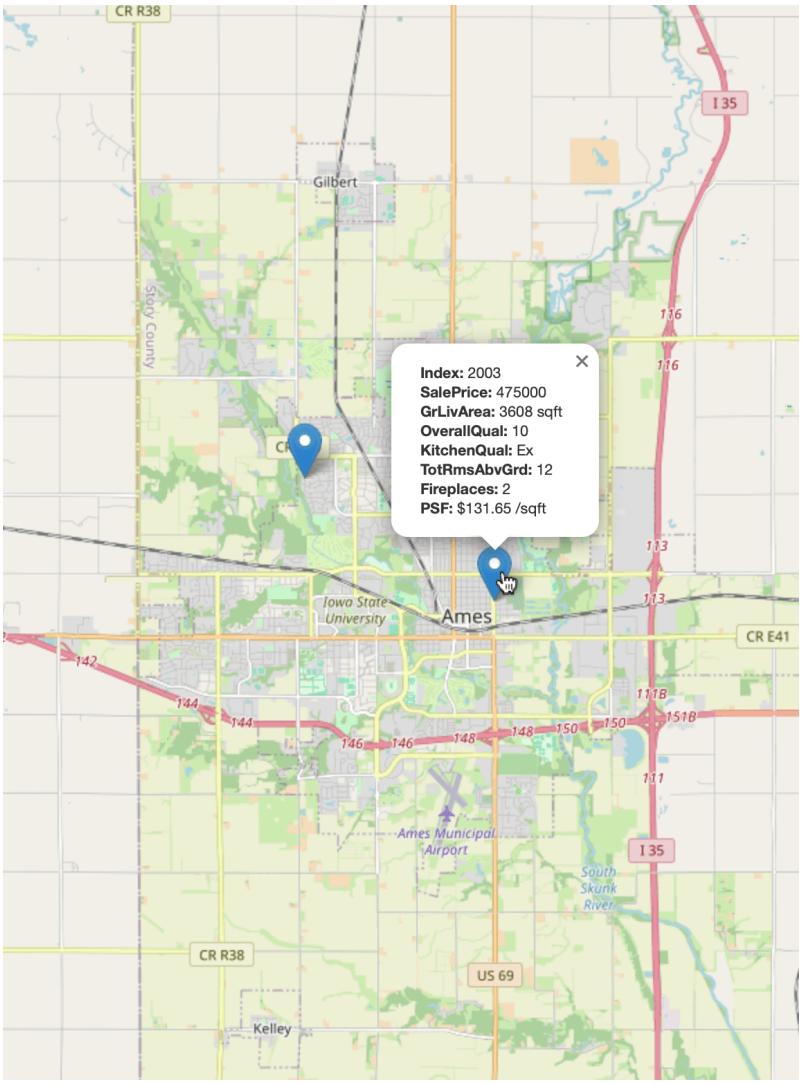

Figure 15.1: Screenshot of Folium output

As you encapsulate your analytical journey, this visualization emerges as a pivotal resource, guiding potential homeowners through a market replete with hidden gems, illuminated by the meticulous application of data science. This approach not only empowers buyers with a deeper understanding of luxury real estate values but also fosters a more informed, strategic, and ultimately fulfilling decision-making process.

Heat map is another powerful visualization technique offered by Folium. It provides a vivid representation of data density or intensity in a given geographical area, allowing you to identify hotspots of activity or interest within the Ames real estate market.

```
import pandas as pd
import folium
from folium.plugins import HeatMap

Ames = pd.read_csv('Ames.csv')

# Filter out rows with NaN values in 'Latitude' or 'Longitude'
Ames_Heat = Ames.dropna(subset=['Latitude', 'Longitude'])

# Group by 'Neighborhood' and calculate mean 'Latitude' and 'Longitude'
neighborhood_locs = Ames_Heat.groupby('Neighborhood') \
                             .agg({'Latitude':'mean', 'Longitude':'mean'}) \
                             .reset_index()

# Create a map centered around Ames, Iowa
ames_map_center = [Ames_Heat['Latitude'].mean(), Ames_Heat['Longitude'].mean()]
ames_heatmap = folium.Map(location=ames_map_center, zoom_start=12)

# Extract latitude and longitude data for the heatmap
heat_data = [(lat,lon) for lat, lon in zip(Ames_Heat['Latitude'], Ames_Heat['Longitude'])]

# Create and add a HeatMap layer to the map
HeatMap(heat_data, radius=12).add_to(ames_heatmap)

# Add one black flag per neighborhood to the map
for index, row in neighborhood_locs.iterrows():
    folium.Marker(
        location=[row['Latitude'], row['Longitude']],
        popup=row['Neighborhood'],
        icon=folium.Icon(color='black', icon='flag')
    ).add_to(ames_heatmap)

# Save the map to an HTML file in the working directory
ames_heatmap.save('ames_heatmap.html')
```

Listing 15.7: Showing heatmap on Folium

In the output, you've strategically placed flags to mark each neighborhood within Ames, providing immediate visual cues to their locations. These flags, distinguished by their black color, bear the full name of each neighborhood, serving as a navigational guide through your exploration. Additionally, the heat map utilizes a color gradient to indicate the density of properties, with warmer colors representing higher concentrations of homes. This color coding not only enhances the visual appeal of your map but also offers an intuitive understanding of market activity and potential areas of interest for developers and buyers alike.

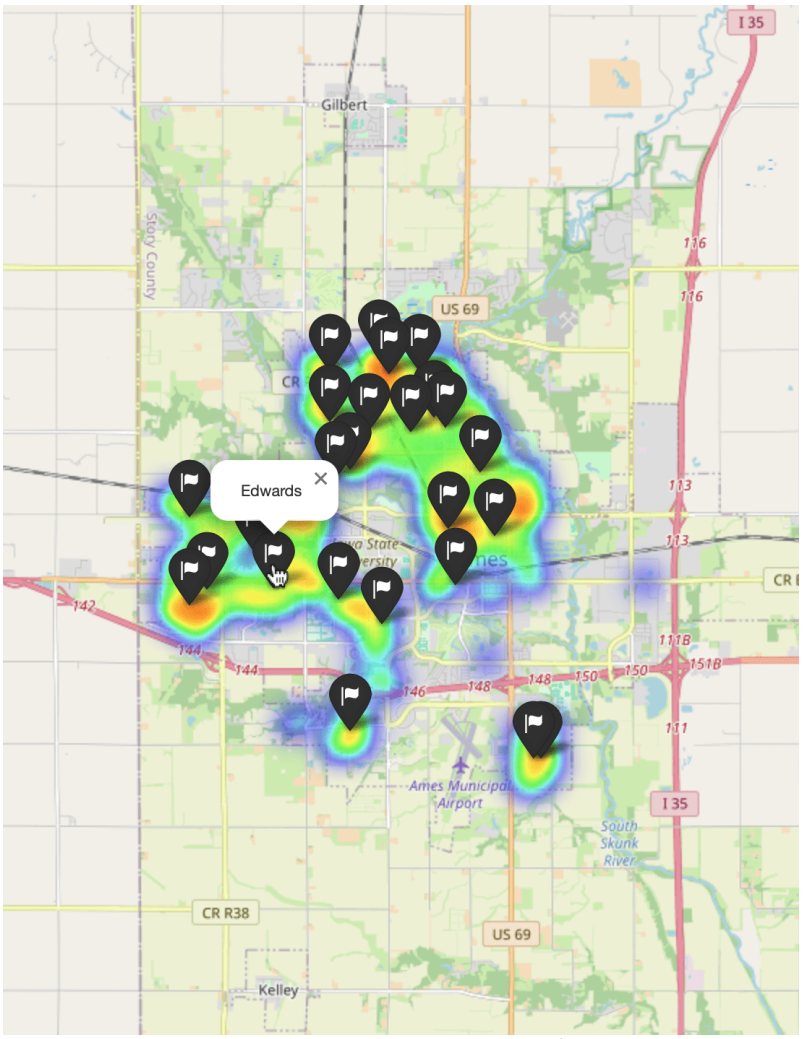

Figure 15.2: Heatmap on Folium

Leveraging insights from your heat map visualization enables developers to strategically target lower-density neighborhoods. By focusing on emerging neighborhoods and prioritizing innovative design and construction efficiency, developers can introduce a new dimension of affordable luxury homes. This strategy caters to the demand for high-quality living spaces and expands the market by making luxury homes more accessible to a broader audience. This combined strategy, informed by data science, geospatial analysis, and domain experts, underscores the transformative potential of these disciplines in shaping the future of accessible luxury real estate, ensuring that high-quality living is within reach for more people.

15.4 Further Reading

This section provides more resources on the topic if you want to go deeper.

Resources

Folium: Interactive Mapping Library. PyPI.
https://pypi.org/project/folium/

Svetlana Lavrinenko. *Leveraging Data Science for Real Estate Excellence*. DataForest, Sept. 2023.
https://dataforest.ai/blog/leveraging-data-science-for-real-estate-excellence

15.5 Summary

In this comprehensive exploration, you delved into the transformative role of data science and interactive mapping in the real estate market, particularly focusing on the luxury segment in Ames, Iowa. Through a meticulous application of data analysis and visualization techniques, you uncovered invaluable insights that not only empower luxury homebuyers but also open new avenues for developers looking to innovate within the realm of accessible luxury real estate.

Specifically, you learned:

- The application of data science in identifying luxury properties.
- The introduction of feature engineering and the calculation of Price Per Square Foot (PSF) as innovative methods to assess relative value.
- How to utilize Folium, a powerful Python library, for creating dynamic visuals to enhance real estate decisions.

This marks the end of this book.

III Appendix

How to Install Python

If not using Anaconda, you may also install Python by downloading the standard CPython environment from the official Python.org site. You may not have a full-featured environment in one shot as you can get from Anaconda, but it is no less powerful. In this tutorial, you will discover how to setup a Python environment using CPython. After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing data science projects.

Overview

In this tutorial, we will cover the following steps:

1. Download CPython
2. Install CPython
3. Install Visual Studio Code environment

Note: The specific versions may differ as the software and libraries are updated frequently.

A.1 Download CPython

In this step, you will download the CPython installer for your platform. CPython is the standard and most common Python environment. It is available for Windows, macOS, and Linux platforms. Windows are used below for demonstration, so you may see some Windows dialogs and file extensions.

1. Visit the Anaconda homepage `https://www.anaconda.com/`

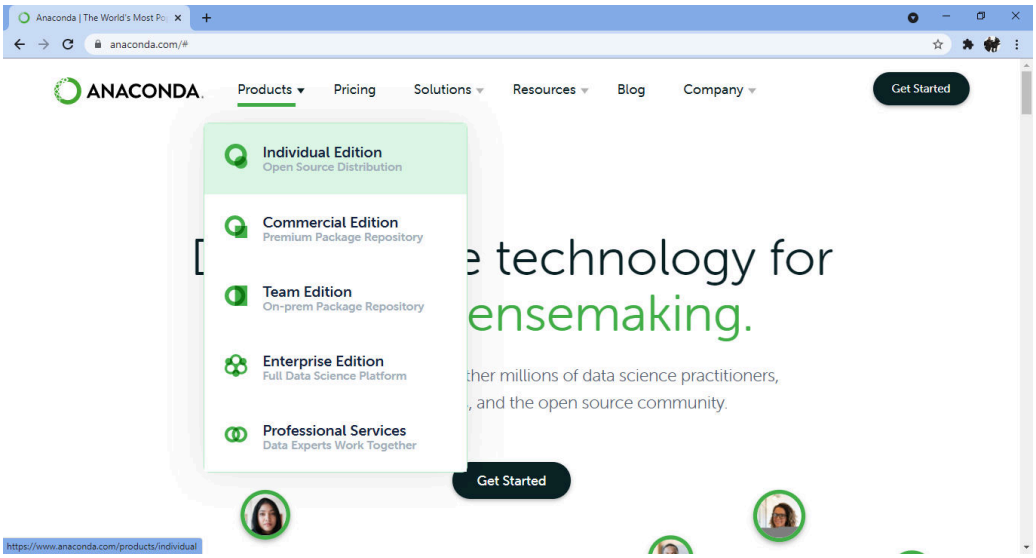

Figure A.1: Click "Products" and "Individual Edition"

2. Click "Products" from the menu and click "Individual Edition" to go to the download page `https://www.anaconda.com/products/individual-d/`.

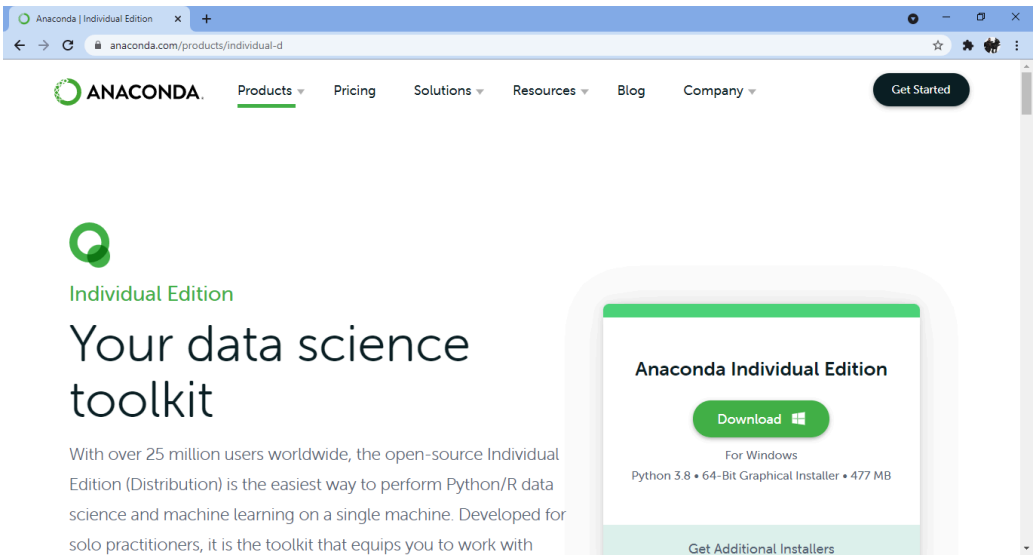

Figure A.2: Click Download

This will download the Anaconda Python package to your workstation. It will automatically give you the installer according to your OS (Windows, Linux, or MacOS). The file is about 480 MB. You should have a file with a name like:

```
Anaconda3-2021.05-Windows-x86_64.exe
```

A.2 Install Anaconda

In this step, you will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

1. Double-click the downloaded file.
2. Follow the installation wizard.

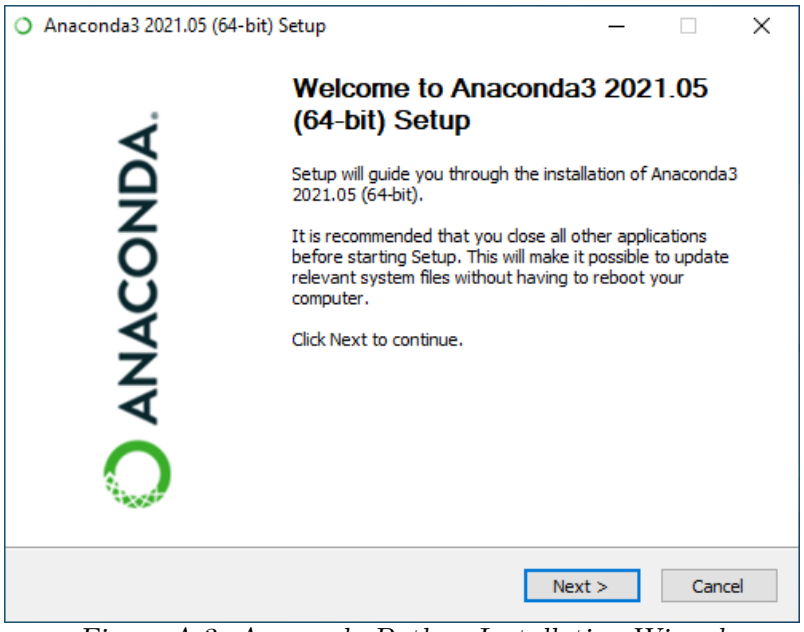

Figure A.3: Anaconda Python Installation Wizard

Installation is quick and painless. There should be no tricky questions or sticking points.

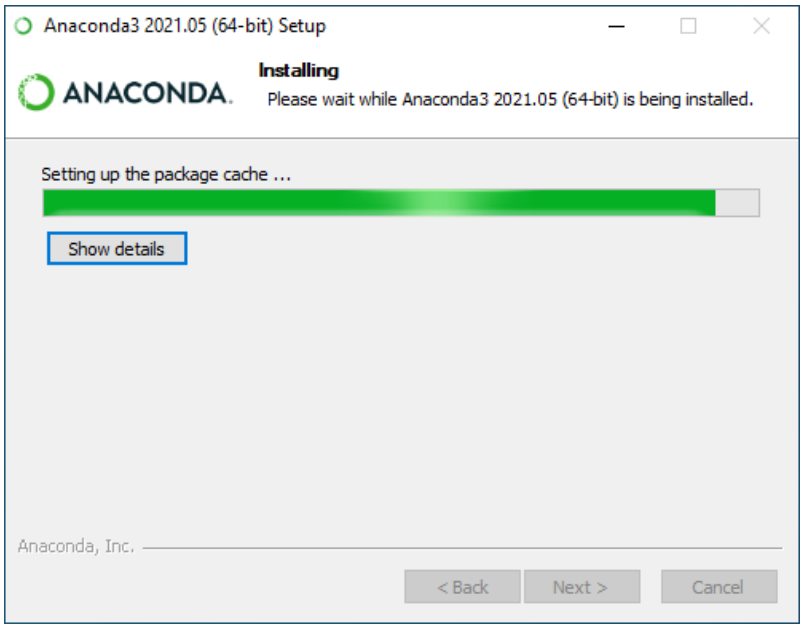

Figure A.4: Anaconda Python Installation Wizard Writing Files

The installation should take less than 10 minutes and take a bit more than 5 GB of space on your hard drive.

A.3 Start and Update Anaconda

In this step, you will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

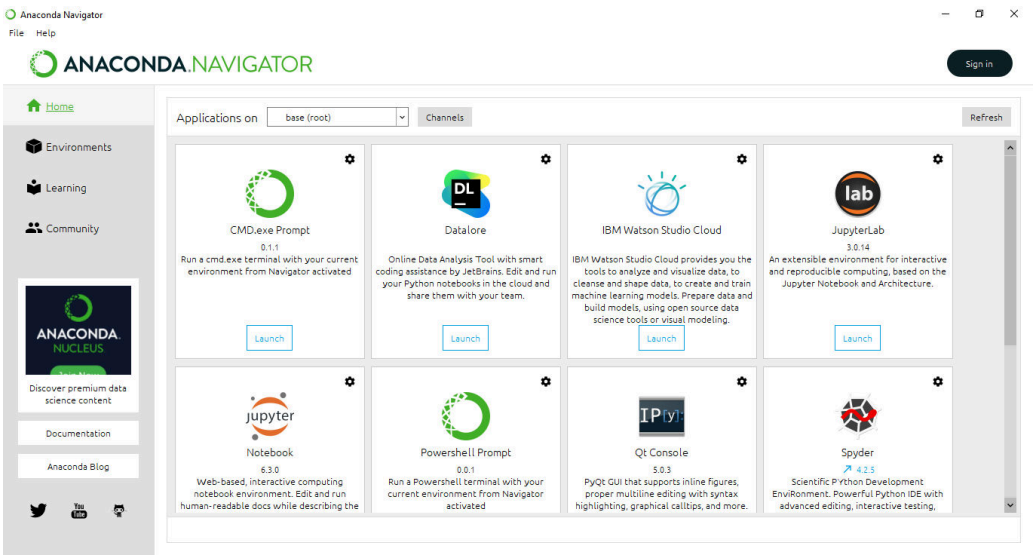

Figure A.5: Anaconda Navigator GUI

You can use the Anaconda Navigator and graphical development environments later; for now, it is recommended to start with the Anaconda command line environment called conda[1]. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

1. Open a terminal or CMD.exe prompt (command line window).
2. Confirm conda is installed correctly, by typing:

```
conda -V
```

   You should see the following (or something similar):

```
conda 4.10.1
```

3. Confirm Python is installed correctly by typing:

```
python -V
```

[1] https://conda.pydata.org/docs/index.html

You should see the following (or something similar):

```
Python 3.8.8
```

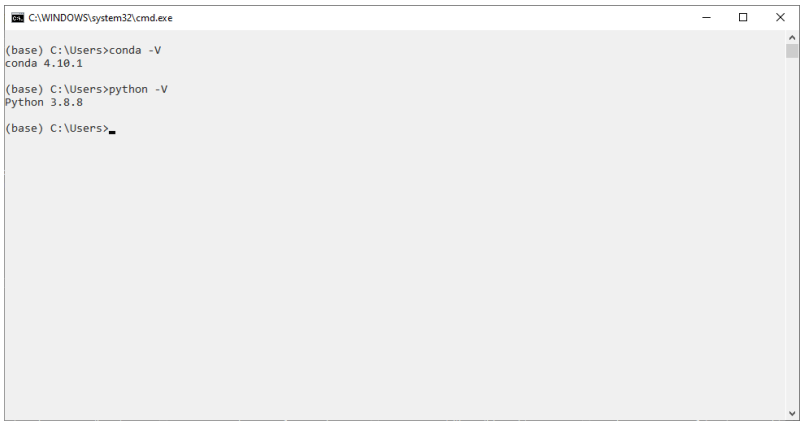

Figure A.6: Confirm Conda and Python are Installed

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the "Further Reading" section.

4. This step is optional. You can make community-supported packages available in conda by adding a "conda-forge" channel:

```
conda config --add channels conda-forge
```

5. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

You may need to install some packages and confirm the updates.

6. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for this book, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type "`python`" and type the commands in directly. Alternatively, it recommended to open a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
```

```
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing A.1: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

You should see output like the following:

```
scipy: 1.8.1
numpy: 1.23.1
matplotlib: 3.5.1
pandas: 1.4.3
statsmodels: 0.13.2
sklearn: 1.1.1
```

Output A.1: Sample output of thhe versions script

Figure A.7: Confirm Anaconda SciPy environment

A.4 Install Visual Studio Code for Python

If you have installed Anaconda, you will have an IDE called Spyder installed. You can develop your Python project with Spyder. The other popular way of writing Python code nowadays

is to use Visual Studio Code. It is a free programming environment from Microsoft. Visual Studio Code can do a lot of things via "extensions". Python extension is what you should get.

These instructions are suitable for Windows, macOS, and Linux platforms. Below, macOS is used to demonstrate, so you may see some Windows dialogs and file extensions.

1. Visit the Visual Studio Code homepage `https://code.visualstudio.com`

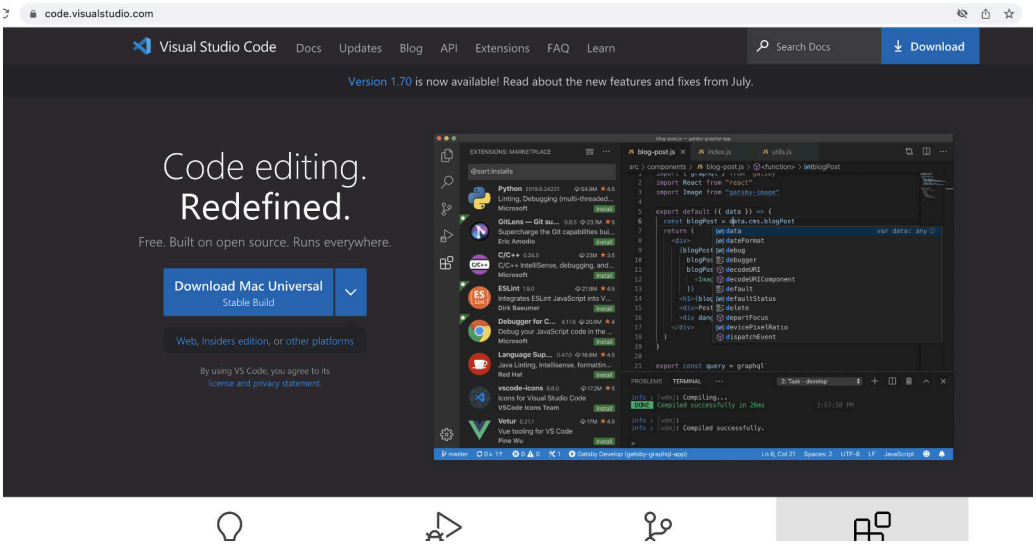

Figure A.8: Click "Products" and "Individual Edition"

2. Click the download button at the top toolbar or at the center of the screen to download a ZIP file (such as `VSCode-darwin-universal.zip`). It is around 200MB in size
3. Expand the ZIP you will find the application program. In macOS, you should move it into your `/Applications` folder

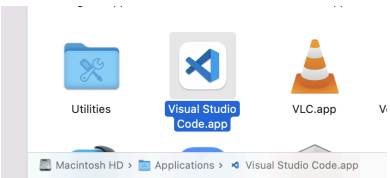

Figure A.9: Visual Studio Code in `/Applications` in macOS

4. Running Visual Studio Code will show you the main screen like the following:

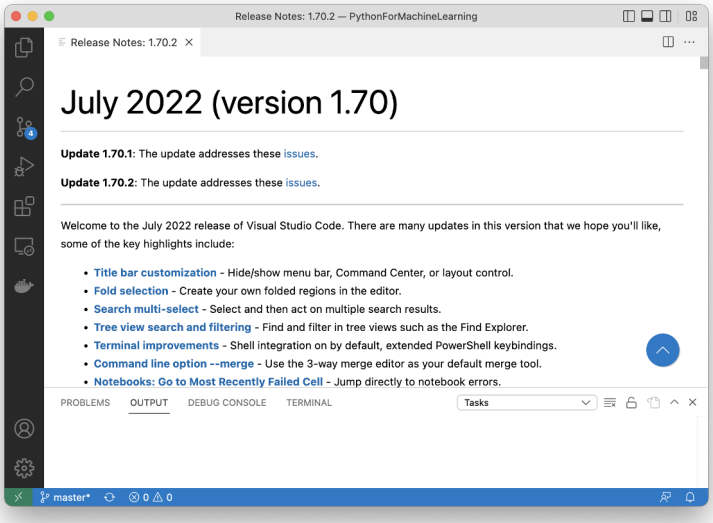

Figure A.10: Visual Studio Code main screen

You can click on the building block icon on the left toolbar to open the extensions marketplace. Typing "`python`" on the search box will usually show the Microsoft-developed Python extension at the top.

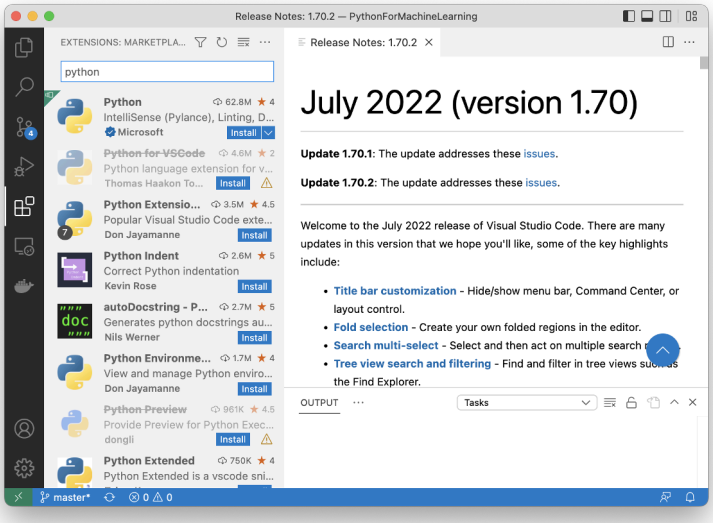

Figure A.11: Searching for Python extension in the extensions marketplace

5. Click on the "Install' button on the extension marketplace will get the extension installed. It should be very quick.

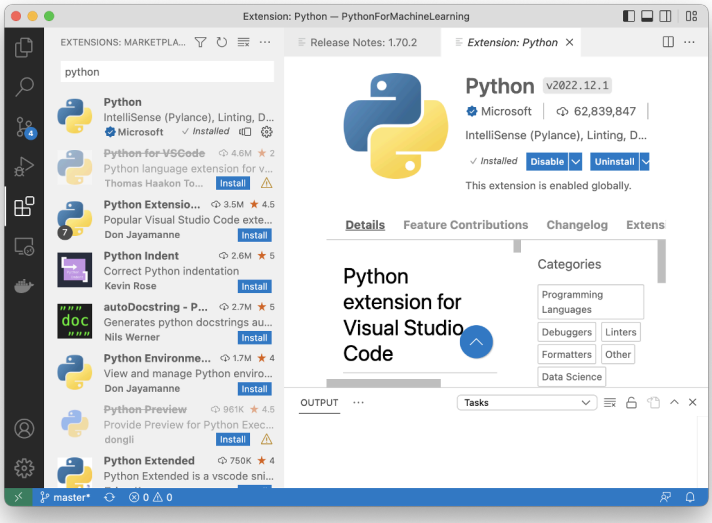

Figure A.12: Python extension installed in Visual Studio Code

Afterward, you will find your Visual Studio Code can launch a Python interpreter or Python debugger to run your program.

A.5 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda Documentation
  https://docs.continuum.io/
- Anaconda Documentation: Installation
  https://docs.continuum.io/anaconda/install
- Anaconda Navigator
  https://docs.continuum.io/anaconda/navigator.html
- The conda command line tool
  https://conda.pydata.org/docs/index.html
- Using conda
  https://conda.pydata.org/docs/using/
- conda-forge
  https://conda-forge.org//docs/using/

A.6 Summary

Congratulations, you now have a working Python development environment for your projects on your workstation.

The Da Vinci Code of Data: Mastering The Data Science Mind Map

B

Data Science embodies a delicate balance between the art of visual storytelling, the precision of statistical analysis, and the foundational bedrock of data preparation, transformation, and analysis. The intersection of these domains is where true data alchemy happens - transforming and interpreting data to tell compelling stories that drive decision-making and knowledge discovery. Just as Leonardo da Vinci masterfully blended scientific observation with artistic genius, we will explore how the art of storytelling in data science can illuminate insights with the same precision and beauty. In this appendix, we will navigate through our Data Science Mind Map to unpack and simplify this process while providing links that showcase concrete examples.

B.1 Mastering The Data Science Mind Map

In our quest to master the Data Science Mind Map, we emphasize the critical importance of the foundational Python packages that every data scientist should be familiar with. These packages form the pillars of our Mind Map, representing the triad of essential skills: data preparation, visualization, and statistical analysis. They are tools and building blocks that enable us to transform raw data into a compelling narrative. As we proceed, we will delve into the role of each package and its dual or singular functions within the data science workflow, exploring their synergy and individual strengths in crafting data stories.

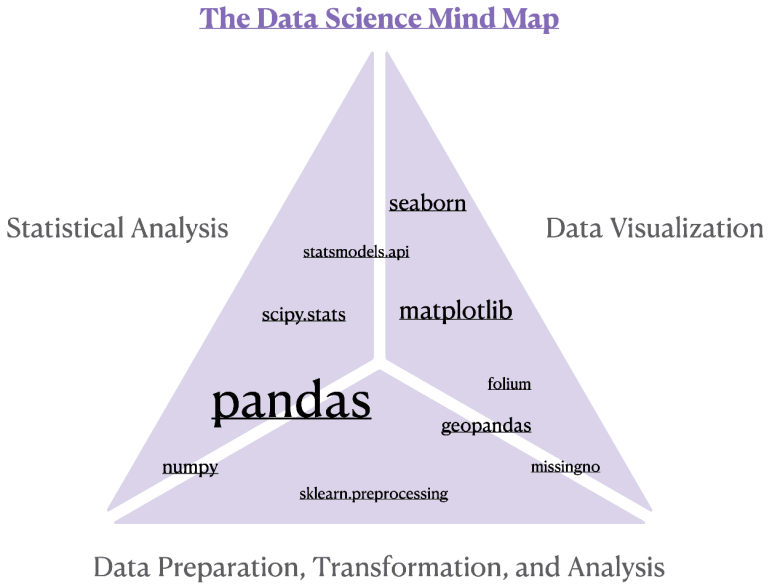

Figure B.1: Data Science Mind Map

Pandas: The brainchild of Wes McKinney, `pandas` stands out as a foundation for data wrangling and as a bridge to statistical analysis. In `pandas`, the `DataFrame` is not just a data structure; it is the cornerstone upon which data manipulation, transformation, and analysis are built. This two-dimensional, size-mutable, and potentially heterogeneous tabular data structure is akin to a spreadsheet loaded right into Python. With its rows and columns neatly organized, the `DataFrame` makes data operations both intuitive and efficient. Each method, whether it's `DataFrame.describe()` for statistical summaries, `DataFrame.groupby()` for aggregation, or `DataFrame.pivot_tables()` for advanced reshaping, is applied to a `DataFrame`, unlocking the full potential of your data. As shown in Chapter 5, `pandas` allows you to distill complex datasets into meaningful statistics efficiently, a vital step before any further analysis. Moreover, understanding data types is pivotal, as it determines the type of analysis you can perform. Chapter 2 guides you through this critical stage, where methods like `DataFrame.dtypes` and `DataFrame.select_dtypes()` in `pandas` facilitate the discernment and manipulation of different data categories. The `DataFrame.query()` function lets you filter with ease, making complex SQL-like querying in Python a breeze, and offering a more dynamic approach to data manipulation. For more in-depth examples and applications of these methods, consider exploring the insights on transforming real estate data and harmonizing data techniques in Chapter 4.

numpy: A foundational library for numerical computation in Python, enabling data scientists to perform complex mathematical calculations and data manipulation with ease and speed. In Chapter 10, we leveraged `numpy` to efficiently calculate key statistical measures such as the mean, serving as a pivotal step in setting the groundwork for conducting hypothesis tests. While `pandas` excels at handling tabular data, `numpy` follows suit by providing support for arrays, forming a formidable duo in the data science toolkit.

`sklearn.preprocessing`: Despite the depth of this series not extending into machine learning, it's worthwhile to highlight `sklearn.preprocessing` for its role in data transformations,

specifically with the `QuantileTransformer()`. This technique was demonstrated in Chapter 14 that discussed how to combat skew.

`missingno`: The `missingno` package uniquely bridges the gap between preprocessing, analysis, and visualization in the data science workflow. It specializes in providing a graphical representation of missing data within a dataset, thus serving a dual function: it aids in the early stages of data cleaning and preparation by visually identifying patterns of missingness, and it also facilitates exploratory data analysis by revealing underlying structures or anomalies that could influence subsequent statistical analyses. In Chapter 1, we delve into how `missingno` can be used to efficiently detect and handle missing data, demonstrating its critical role in ensuring the integrity and robustness of your data science projects. Through intuitive visualizations, `missingno` not only enhances data preprocessing but also enriches the analytical narrative by shedding light on aspects of the data that often remain obscured.

`geopandas`: This package extends the functionalities of `pandas` into the realm of geospatial data, making it an indispensable tool for both data preprocessing and visualization in geographical contexts. It allows data scientists to easily manipulate and analyze spatial data, integrating seamlessly with other Python libraries for geospatial analysis. With `Geopandas`, you can perform sophisticated spatial operations, merge spatial datasets, and conduct spatial joins, all while maintaining the familiar `pandas` DataFrame structure. This capability ensures that handling geospatial data is as intuitive as working with tabular data. Moreover, `Geopandas` excels in visualizing geospatial data, enabling the creation of maps that can reveal compelling insights into geographical patterns and relationships. In Chapter 6, we showcase how `Geopandas` can transform raw geospatial data into informative maps, highlighting its dual role in both preprocessing and visually interpreting spatial data within the data science workflow.

`Folium`: Specializing in the singular role of creating interactive maps, `folium` leverages the mapping strengths of the Leaflet.js library within the Python environment. It excels at building rich, interactive geospatial visualizations, allowing for the dynamic representation of data overlaid on maps. This capability is invaluable for projects requiring intuitive spatial data exploration and presentation, making `folium` a go-to library for geospatial mapping in Chapter 15.

`matplotlib` and *`seaborn`*: These two packages emerge as pivotal threads, interwoven to enhance the fabric of analytical storytelling. `Matplotlib`, the foundational library, offers extensive flexibility and control, laying the groundwork for creating a wide array of static, animated, and interactive visualizations. It serves as the bedrock upon which `seaborn` is built, with the latter extending capabilities of `matplotlib` by offering a high-level interface for drawing attractive and informative statistical graphics. `seaborn` specializes in making complex visualizations accessible, integrating tightly with `pandas` DataFrames to streamline the process from data manipulation to representation. This synergy is particularly evident when exploring feature relationships and uncovering patterns within datasets, as advanced plotting functions in `seaborn`, like pair plots, build upon the foundational structions in `matplotlib` to provide richer, more insightful visual narratives. Chapters 7 and 8 delve into how `seaborn` and `matplotlib`, in conjunction with `pandas`, form a cohesive visualization suite. Together, these libraries offer an unparalleled toolkit for data scientists aiming to translate complex data insights into compelling visual stories, highlighting the interconnectedness and distinct strengths of each package in the visualization ecosystem.

statsmodels.api: This tool is particularly useful in the realm of statistical visualization through its QQ plots (Quantile-Quantile plots) functionality, aiding in the assessment of whether data distributions match theoretical expectations, typically the normal distribution. We demonstrate this technique in Chapter 12. The generation of a QQ plot involves comparing the sorted values of the sample data against the expected values of the chosen theoretical distribution, providing a graphical method to evaluate the assumption of normality critical to many parametric statistical tests.

scipy.stats: As the data science journey progresses from descriptive to inferential statistics, `scipy.stats` emerges as a pivotal toolkit. This package is fundamental for conducting a wide range of statistical tests and analyses that form the backbone of inferential statistics, enabling data scientists to draw meaningful conclusions from their data. Within `scipy.stats`, you find an extensive array of functions designed for hypothesis testing, confidence interval estimation, and much more, making it indispensable for rigorous statistical investigation.

Our exploration of statistical techniques through various chapters demonstrates the versatility and power of `scipy.stats`:

- ▷ In Chapter 9, we delve into how confidence intervals can provide a range of plausible values for an unknown parameter, showcasing the `t.interval` function for calculating intervals based on sample data.
- ▷ Chapter 10 illustrates the core of inferential statistics, employing tests like the t-test to evaluate hypotheses about our data.
- ▷ Our examination of categorical data analysis in Chapter 11 leverages the `chi2_contingency()` function to test for independence between variables.
- ▷ Lastly, Chapter 12 highlights how `scipy.stats` supports both parametric (ANOVA) and nonparametric (Kruskal-Wallis) tests to assess the effects of categorical variables on continuous outcomes.

`scipy.stats` thus plays a crucial role in transitioning data science efforts from understanding what is in the data (descriptive statistics) to infering the implications of that data (inferential statistics), providing a comprehensive suite for statistical testing and analysis.

The Data Science Mind Map introduces you to a collection of Python libraries, each playing a distinct yet interconnected role in the broader data science landscape. From the data structuring prowess of `pandas` and the numerical might of `numpy`, to the cleaning insights provided by `missingno` and the geographical intelligence of `geopandas`; from the captivating visualizations afforded by `folium`, `matplotlib`, and `seaborn`, to the analytical depth and statistical rigor of `statsmodels.api` and `scipy.stats`— each library contributes a unique thread to the interdisciplinary nature of Data Science.

B.2 The Art of Storytelling in Data Science

Imagine the process of storytelling in data science as Leonardo da Vinci embarks on the creation of a masterpiece. Each brushstroke, choice of color, and play of light and shadow serve a purpose, much like the elements of our data narrative. Let's explore this artistic journey.

Sketching the Outline: Before touching brush to canvas, Leonardo spent countless hours in preparation. He dissected human bodies to understand anatomy, studied the properties of light and shadow, and sketched detailed drawings. Similarly, our first step in data storytelling involves deep diving into the dataset, understanding its variables, and planning our analysis. This stage sets the foundation for a narrative that is both accurate and compelling.

Choosing the Palette: Just as Leonardo mixed his paints to achieve the perfect hues, a data storyteller selects tools and techniques from the Data Science Mind Map. The choice of Python packages, such as pandas for data manipulation, Matplotlib and seaborn for visualization, or `scipy.stats` for statistical analysis, becomes our palette, allowing us to illuminate insights from the data.

Creating Depth with Perspective: Leonardo's use of perspective gave his paintings depth, making them more lifelike and engaging. In data storytelling, we create depth through analysis, examining the data from multiple angles to uncover underlying patterns and relationships. This perspective helps us build a narrative that resonates with the audience, providing them with insights beyond the surface.

Highlighting with Light and Shadow: Leonardo was a master of chiaroscuro, the technique of using light and shadow to bring drama and focus to his paintings. In our data story, visualizations serve as our light and shadow, highlighting key findings and drawing the audience's attention to the most important insights. Through effective visualization, we can make complex data understandable and memorable.

The Final Masterpiece: When Leonardo presented his finished work, it was not just a painting; it was a story captured in time, evoking emotion and provoking thought. Our data story, culminating in the presentation of our findings, aims to do the same. It's where our preparation, analysis, and visualization come together to inform, persuade, and inspire our audience to action.

Just as viewers stand in front of a da Vinci painting, absorbing its beauty and depth, we invite your audience to reflect on the data-driven stories you will tell. This reflection is where understanding deepens, and the true impact of your work is felt, echoing the enduring legacy of da Vinci's art.

Further Reading

Brent Dykes. *Data Storytelling: The Essential Data Science Skill Everyone Needs*. Forbes. Mar. 2016.
https://www.forbes.com/sites/brentdykes/2016/03/31/data-storytelling-the-essential-data-science-skill-everyone-needs/

Unfolding Data Stories: From First Glance to In-Depth Analysis

C

The path to uncovering meaningful insights often starts with a single step: looking at the data before asking questions. This journey through the Ames Housing dataset is more than an exploration; it's a narrative about the hidden stories within numbers, waiting to be told. Through a "Data First Approach," we invite you to dive deep into the process of data-driven storytelling, where every visualization, every statistical test, and every hypothesis forms a part of a larger narrative. This appendix is designed to guide you through a step-by-step process of understanding and presenting data, from the initial broad view of the dataset to the focused lens of hypothesis testing, unraveling the intricate tales woven into the Ames Housing Market.

C.1 The Data First Approach

What comes first, the question or the data?

Starting our data science journey often involves a counterintuitive first step: beginning with the data itself, before posing any specific questions. This perspective is at the heart of the "Data First Approach," a philosophy that champions the power of discovery by allowing the data to lead the way. Advocating for an open-minded exploration, this approach turns the dataset at hand—such as the detailed and rich Ames Housing dataset—into a guiding light, revealing stories, secrets, and the potential for insightful analysis. This philosophy urges us to set aside our preconceived notions, enabling the inherent trends of data, patterns, and insights to surface naturally. A concise three-step guide to embracing this approach includes:

1. *Sizing Up The Data*: The initial step, emphasizing our "Data First Approach," involves understanding the size and shape of your data, as highlighted in Chapter 1. This stage is crucial for grasping the scope of dataset and addressing any missing values, setting the groundwork for comprehensive analysis.
2. *Understanding The Spectrum of Data Types*: Delving deeper into our dataset, we explore the variety of data types it contains, a crucial step for informing our choice of visuals and framing our analytical questions. This exploration, akin to navigating through Chapter 2, is vital for tailoring our analysis and visualization strategies to the inherent characteristics of data, ensuring our methods are both relevant and effective.

3. *Descriptive Statistics*: Outlined in Chapter 5, this step provides tools for quantitatively summarizing and understanding the dataset, preparing us for deeper analysis and interpretation.

Integrating these steps into our preliminary exploration underscores the "Data First Approach," systematically unveiling the stories embedded within the Ames Housing dataset. Each step acts as a cornerstone in revealing a fuller narrative. By allowing the data to speak first, we unlock the most compelling stories hidden within the numbers.

C.2 Anchored in Data, Revealed Through Visuals

Following our "Data First Approach," where we prioritize a thorough understanding of the dataset and its variables, we naturally progress to the next crucial step: visualization. This stage is where our initial engagement with the data informs the selection of the most appropriate visual tools to illuminate the insights we've uncovered. Visualization is not just about making data look appealing; it's an integral part of the storytelling process, enabling us to "Show, Don't Tell" the stories hidden within the data. The art lies in choosing the right type of visualization that resonates with the narrative of data, a decision deeply rooted in our initial exploration. Here are several key visualizations and their optimal use cases:

- ▷ *Histograms*: Ideal for showcasing the distribution of a single numerical variable (Chapter 9). Histograms help identify skewness, peaks, and the spread of the data, making them perfect for analyzing variables like income levels or ages within a population.
- ▷ *Bar Charts*: Effective for comparing quantities across different categories (Chapter 3). Use bar charts to highlight differences between groups, such as sales figures across different regions or customer counts by product category.
- ▷ *Line Charts*: Best suited for displaying data trends over time. Line charts are the go-to choice for visualizing stock price changes, temperature fluctuations over a year, or sales growth across quarters.
- ▷ *Scatter Plots*: Excellent for exploring relationships between two numerical variables. Scatter plots can help identify correlations (Chapter 7), such as the relationship between advertising spend and sales revenue, or height and weight correlations.
- ▷ *Box Plots (Box-and-Whisker Plots)*: Useful for summarizing the distribution of a dataset and comparing distributions between groups (Chapter 12). Box plots provide insights into the median, quartiles, and potential outliers within data, making them valuable for statistical analyses like comparing test scores across different classrooms.
- ▷ *Heat Maps*: Ideal for visualizing complex data matrices (Chapter 3), showing patterns of similarity or variation. Heat maps are effective in areas like displaying website traffic sources across different times of the day or understanding geographical data distributions.
- ▷ *Geospatial Maps*: Ideal for showcasing data with a geographical component, allowing for a visual representation of patterns and trends across different regions. Geospatial maps are perfect for visualizing population density, sales distribution by location

(Chapter 6), or any data that has a spatial element. They help in identifying regional trends, making them invaluable for analyses that require a geographical context, such as market penetration in different cities or climate change effects in various parts of the world.

- ▷ *Stacked Bar Charts*: Great for displaying part-to-whole relationships and comparisons across categories, with each bar segment representing a value of a sub-category. Use stacked bar charts to illustrate sales data divided by product type over multiple periods.
- ▷ *Area Charts*: Similar to line charts but filled beneath the line, area charts are useful for emphasizing the magnitude of change over time. They work well for visualizing cumulative totals, such as website traffic sources or population growth.
- ▷ *Pair Plots*: Ideal for exploring correlations and distributions among multiple variables simultaneously. Pair plots, or scatter plot matrices, provide a comprehensive view of how every variable in a dataset relates to each other, highlighting potential relationships and trends that merit further investigation (Chapter 8). They are particularly useful in the early stages of analysis to quickly assess potential variables of interest.

Visualization is an iterative process. Initial visuals often lead to new questions, prompting further analysis and refined visuals. This cycle enhances our understanding, gradually revealing the fuller narrative woven into our data. To delve deeper into the iterative visualization process using the Ames Housing dataset, let's explore potential questions and the types of visuals that could help answer them. Here are some questions, along with the suggested types of visuals:

- ▷ *What patterns can be observed in the sale prices across different months and seasons?*
  Line Charts or Bar Charts to analyze seasonal trends in sale prices.
- ▷ *How does the lot size compare to the sale price across different zoning classifications?*
  Scatter Plots with different colors for each zoning classification to explore the relationship between lot size and sale price.
- ▷ *How it affect the sales price if the property has a pool?*
  Box Plots comparing the sale prices of homes with and without pools.
- ▷ *How do year built and year remodeled affect the overall condition of the property and sale price?*
  Pair Plots to simultaneously explore the relationships between year built, year remodeled, overall condition, and sale price.
- ▷ *Is there a correlation between the proximity to various amenities (parks, schools, etc.) and sale prices?*
  Geospatial Maps with overlays indicating amenities proximity and Scatter Plots to correlate these distances with sale prices.

These questions encourage exploring the dataset from various angles, leading to a richer understanding through iterative visualization. Each visualization not only answers the initial question but may also spark further inquiry, demonstrating the dynamic process of data exploration and storytelling.

C.3 From Patterns to Proof: Hypothesis Testing in the Ames Housing Market

After immersing ourselves in the "Data First Approach" and harnessing the power of visuals to uncover hidden patterns and relationships within the Ames Housing dataset, our journey takes us to the crucial phase of hypothesis formation and testing (Chapter 10 and 11). This iterative process of questioning, exploring, and deducing represents the essence of data-driven storytelling, transforming observations into actionable insights.

We are now ready to ask deeper questions, inspired by the patterns and anomalies our visuals have uncovered. Here are few possible directions one can take that have not yet been demonstrated in our previous chapters:

- ▷ *Does the sale price depend on the neighborhood?*
  One-way ANOVA to compare sale prices across multiple neighborhoods, assuming equal variances; otherwise, Kruskal-Wallis test.
- ▷ *Is there a significant difference in sale prices between the different types of dwellings (e.g., 1-story vs. 2-story homes)?*
  ANOVA for multiple groups, or t-test for comparing two specific dwelling types.
- ▷ *Are there significant differences in sale prices among houses with different exterior materials?*
  Chi-square test for independence after categorizing sale prices into bands (low, medium, high) and comparing against types of exterior materials.
- ▷ *Is the sale price influenced by the season in which the house is sold?*
  Kruskal-Wallis test or ANOVA, depending on the distribution, to compare median sale prices across different seasons, identifying if certain times of the year yield higher sale prices.
- ▷ *Does having a finished vs. unfinished basement significantly impact the sale price?*
  t-test or Mann-Whitney U-test (based on the data distribution) to compare the sale prices between homes with finished basements versus those with unfinished basements.

The transition from visualization to hypothesis testing is not merely analytical; it's a creative process that involves synthesizing data insights into compelling narratives. Each hypothesis tested sheds light on the dynamics at play within the housing market, contributing chapters to the broader story of the Ames dataset. As we validate or refute our hypotheses, we're not just gathering evidence; we're constructing a story grounded in data. This narrative might reveal how the Ames housing market pulsates with the rhythms of the seasons, or how modernity commands a premium, reflecting contemporary buyers' preferences.

By marrying the "Data First Approach" with the iterative exploration of visuals and the rigor of hypothesis testing, we unlock a deeper understanding of our data. This approach not only enhances our comprehension but also equips us with the tools to share our findings compellingly and convincingly, turning data exploration into an engaging narrative that resonates with audiences. In embracing this threefold path—anchored in data, revealed through visuals, and narrated through hypotheses—we craft stories that not only inform but inspire, showcasing the transformative power of data-driven storytelling.

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

- ▷ You learned how to manipulate a tabular dataset as a DataFrame in pandas.
- ▷ You learned how to write code to extract a basic summary from a dataset, such as the mean subject to a filtering condition.
- ▷ You learned how to visualize data as a probability distribution and interpret its properties such as skewness.
- ▷ You learned how to use different techniques to figure out the relationship between features.
- ▷ You also learned how to formulate your belief, convert it into a statistical hypothesis, then prove or disprove it quantitatively.

This is a long way but you made it in a short amount of time. These are important and valuable skills. In the domain of data science, they give you the ability to reason mathematically and rigorously. Not only you can use them to confirm your guess but also help you to explore the data and learn what you haven't thought of. In the context of this book, you can now:

- ▷ Among the many properties to describe a house, you can tell which are more influential.
- ▷ In a town, you can tell how many segments in the housing market and what are the distinctive features.
- ▷ Instead of verbally arguing, you can statistically prove whether a feature affects the house price.
- ▷ Identify the houses that are priced unreasonably.

We the editorial team in Machine Learning Mastery want to thank you for letting us help you start your journey of data science. We hope you feel enlightened and see image processing differently afterward.