Data Analysis with Python

Exercise Guide

Version 1.0.0-v0.0.0

P.I.L.

Public Information Limited, LLC.

Table of Contents

Data Analytics with Python Course Overview 1

Course Structure 1

Target Audience 2

Learning Outcomes 2

Prerequisites 2

Module 1: Data Visualization 3

Exercise 1A - Create an interactive Plotly visualization displaying trends in a dataset. 4

Exercise 1B - Generate a static Matplotlib plot illustrating the distribution of a numerical variable. 5

Exercise 1C - Use Seaborn to visualize the relationship between multiple variables in a dataset. 6

Exercise 1D - Combine Plotly, Matplotlib, and Seaborn to create a comprehensive dashboard showcasing various aspects of a dataset. 7

Module 2: Data Manipulation 9

Exercise 2A - Create NumPy arrays and perform basic array manipulations such as reshaping and indexing. 10

Exercise 2B - Use Pandas to load, clean, and analyze a dataset, performing tasks such as data filtering and aggregation. 11

Exercise 2C - Explore the features of the Polars library by performing data manipulation tasks and comparing their efficiency with NumPy and Pandas. 12

Module 3: Statistical Analysis 15

Exercise 3A - Perform exploratory data analysis on a given dataset, including calculating descriptive statistics and visualizing distributions. 16

Exercise 3B - Use Statsmodels to perform linear regression analysis and hypothesis testing on a dataset. 18

Exercise 3C - Explore advanced statistical analysis techniques using Pingouin, including conducting correlation analysis and non-parametric tests. 20

Exercise 3D - Conduct hypothesis tests and statistical computations using SciPy. 22

Module 4: Web Scraping 23

Exercise 4A - Create a Scrapy spider to scrape product information from an e-commerce website. 24

Exercise 4B - Use Selenium to scrape data from a dynamically loaded webpage with JavaScript content. 26

Exercise 4C - Implement HTML parsing and scraping with BeautifulSoup to extract information from a news article webpage. 27

Exercise 4D - Combine Scrapy, Selenium, and BeautifulSoup to scrape data from a website with both structured and dynamic content. 29

Module 5: Natual Language Processing 31

Exercise 5A - Perform basic text processing tasks such as tokenization and sentiment analysis using TextBlob. 32

Exercise 5B - Explore NLTK's text corpora and implement advanced text analysis techniques such as sentiment analysis and text classification. 34

Exercise 5C - Fine-tune a pre-trained BERT model for sentiment analysis on a dataset of customer reviews. 36

Exercise 5D - Evaluate the performance of the fine-tuned BERT model and compare it with traditional NLP techniques. 39

Module 6: Time Series Analysis 42

Exercise 6A - Explore and visualize the characteristics of time series data, including trend, seasonality, and noise. 43

Exercise 6B - Use Darts to build time series forecasting models and evaluate their performance on historical data. 45

Exercise 6C - Perform time series decomposition and anomaly detection using Kats on real-world datasets. 47

Exercise 6D - Extract relevant features from time series data using Tsfresh and apply them to machine learning tasks. 49

Data Analytics with Python Course Overview

This comprehensive course equips learners with the essential skills and knowledge to leverage Python for data analytics tasks. From data visualization and manipulation to statistical analysis, web scraping, natural language processing, and time series analysis, participants will gain proficiency in utilizing Python libraries and frameworks for extracting insights from data.

Course Structure

The course is divided into six modules, each focusing on a distinct aspect of data analytics:

1. Data Visualization: Understand the importance of data visualization and explore tools like Plotly, Matplotlib, and Seaborn for creating interactive and static visualizations.
2. Data Manipulation: Master data manipulation techniques using NumPy, Pandas, and Polars to clean, transform, and analyze datasets effectively.
3. Statistical Analysis: Learn fundamental principles of statistical analysis and apply tools like Statsmodels, Pingouin, and SciPy for regression analysis, hypothesis testing, and advanced statistical computations.
4. Web Scraping: Explore the basics of web scraping and dive into structured and dynamic web scraping techniques using Scrapy, Selenium, and BeautifulSoup.
5. Natural Language Processing: Gain insights into natural language processing fundamentals and implement text processing tasks using TextBlob, NLTK, and advanced NLP models like BERT.
6. Time Series Analysis: Understand the characteristics of time series data and utilize libraries like Darts, Kats, and Tsfresh for time series forecasting, analysis, and feature extraction.

Target Audience

This course is ideal for aspiring data analysts, data scientists, researchers, and professionals seeking to enhance their Python skills for data analysis tasks. Basic knowledge of Python programming is recommended but not required.

Learning Outcomes

Upon completion of this course, participants will:

- Possess a strong foundation in data visualization, manipulation, statistical analysis, web scraping, natural language processing, and time series analysis using Python.
- Be proficient in using popular Python libraries and frameworks such as Pandas, NumPy, Plotly, SciPy, and NLTK for various data analytics tasks.
- Gain practical experience through hands-on exercises, case studies, and projects, enabling them to apply learned concepts in real-world scenarios.

Prerequisites

- Basic understanding of Python programming language
- Familiarity with fundamental concepts of data analysis and statistics

Module 1: Data Visualization

P.I.L.
Public Information Limited, LLC.

Learning Objectives

Upon completion of these exercises, students should be able to:

- Create an interactive Plotly visualization displaying trends in a dataset. *(Exercise 1A)*
- Generate a static Matplotlib plot illustrating the distribution of a numerical variable. *(Exercise 1B)*
- Use Seaborn to visualize the relationship between multiple variables in a dataset. *(Exercise 1C)*
- Combine Plotly, Matplotlib, and Seaborn to create a comprehensive dashboard showcasing various aspects of a dataset. *(Exercise 1D)*

Exercise 1A - Create an interactive Plotly visualization displaying trends in a dataset.

```python
import pandas as pd
import plotly.express as px

# Load your dataset
df = pd.read_csv('your_dataset.csv')

# Create the Plotly visualization
fig = px.line(df, x='date_column', y='value_column', title='Trends Over
Time')

# Customize the visualization
fig.update_layout(xaxis_title='Date', yaxis_title='Value')

# Show the visualization
fig.show()
```

- Again, replace `'your_dataset.csv'` with the path to your dataset file.
- Ensure that your dataset has columns with appropriate names (like `'date_column'` and `'value_column'`) that you want to visualize.
- You can modify the code according to your dataset structure and the type of visualization you want to create.

Exercise 1B - Generate a static Matplotlib plot illustrating the distribution of a numerical variable.

Below is an example of generating a static Matplotlib plot illustrating the distribution of a numerical variable:

```python
# Step 1: Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np

# Step 2: Generate some sample data
# For this example, let's generate 1000 random numbers from a normal
distribution
data = np.random.normal(loc=0, scale=1, size=1000)

# Step 3: Create the Matplotlib histogram plot
plt.figure(figsize=(8, 6))
plt.hist(data, bins=30, color='skyblue', edgecolor='black')

# Step 4: Customize the plot (optional)
plt.title('Distribution of a Numerical Variable')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Step 5: Show the plot
plt.show()
```

This code will generate a static histogram plot illustrating the distribution of a numerical variable using Matplotlib. You can adjust parameters such as the number of bins, colors, and labels to customize the plot according to your preferences.

Exercise 1C - Use Seaborn to visualize the relationship between multiple variables in a dataset.

Seaborn is a high-level interface for drawing attractive statistical graphics built on top of Matplotlib. Below is an example of how you can use Seaborn to visualize the distribution of a numerical variable:

```python
# Step 1: Import necessary libraries
import seaborn as sns
import numpy as np

# Step 2: Generate some sample data
# For this example, let's generate 1000 random numbers from a normal
distribution
data = np.random.normal(loc=0, scale=1, size=1000)

# Step 3: Create the Seaborn histogram plot
sns.histplot(data, bins=30, color='skyblue', edgecolor='black')

# Step 4: Customize the plot (optional)
plt.title('Distribution of a Numerical Variable')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Step 5: Show the plot
plt.show()
```

This code will generate a histogram plot illustrating the distribution of a numerical variable using Seaborn. Seaborn provides additional functionalities and a more visually appealing default style compared to Matplotlib, making it a preferred choice for many data visualization tasks. You can further customize the plot using various Seaborn functions and parameters to suit your needs.

Exercise 1D - Combine Plotly, Matplotlib, and Seaborn to create a comprehensive dashboard showcasing various aspects of a dataset.

Combining Plotly, Matplotlib, and Seaborn in a single dashboard can provide a comprehensive and visually appealing way to explore various aspects of a dataset. Below is an example of how you can create such a dashboard:

```
# Step 1: Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

# Step 2: Load your dataset
# For this example, let's use a sample dataset
# Replace 'your_dataset.csv' with the path to your dataset
df = pd.read_csv('your_dataset.csv')

# Step 3: Preprocess your data if necessary

# Step 4: Create visualizations using different libraries
# Plotly scatter plot
fig1 = px.scatter(df, x='feature1', y='feature2', color='target', title
='Plotly Scatter Plot')

# Matplotlib histogram
plt.figure(figsize=(8, 6))
plt.hist(df['numerical_feature'], bins=30, color='skyblue', edgecolor=
'black')
plt.title('Matplotlib Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Seaborn pairplot
plt.figure(figsize=(8, 6))
sns.pairplot(df, hue='target')
plt.suptitle('Seaborn Pairplot')

# Step 5: Show the visualizations
# Plotly scatter plot
fig1.show()
```

```
# Matplotlib histogram
plt.show()

# Seaborn pairplot doesn't need explicit show as it automatically shows
the plot

# Step 6: Combine the visualizations into a single dashboard (optional)
# You can combine them using HTML layout or use specific dashboard
frameworks like Dash for Plotly

# Step 7: Save the dashboard or display it in a web application
(optional)
# You can save the dashboard as HTML or display it using a web framework
like Flask or Django

# For more interactivity and customization, consider using Dash
framework for Plotly
```

This example demonstrates how to create visualizations using Plotly, Matplotlib, and Seaborn separately and then display them together. You can further enhance this dashboard by adding more visualizations, customizing the appearance, and combining them into a single layout using HTML or a dedicated dashboarding framework like Dash for Plotly.

Module 2: Data Manipulation

Upon completion of these exercises, students should be able to:

- Create NumPy arrays and perform basic array manipulations such as reshaping and indexing. *(Exercise 2A)*
- Use Pandas to load, clean, and analyze a dataset, performing tasks such as data filtering and aggregation. *(Exercise 2B)*
- Explore the features of the Polars library by performing data manipulation tasks and comparing their efficiency with NumPy and Pandas. *(Exercise 2C)*

Exercise 2A - Create NumPy arrays and perform basic array manipulations such as reshaping and indexing.

Below is an example of creating NumPy arrays and performing basic array manipulations such as reshaping and indexing:

```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Print the array
print("Original array:")
print(arr)

# Reshape the array to a 2x3 matrix
reshaped_arr = arr.reshape(2, 3)

# Print the reshaped array
print("\nReshaped array:")
print(reshaped_arr)

# Indexing: Accessing elements of the array
print("\nAccessing elements of the array:")
print("Element at index 0:", arr[0])  # Accessing the first element
print("Element at index 2:", arr[2])  # Accessing the third element

# Slicing: Accessing a subset of the array
print("\nAccessing a subset of the array:")
print("Elements from index 1 to 3:", arr[1:4])  # Accessing elements
from index 1 to 3
```

This code demonstrates the creation of a NumPy array, reshaping it into a matrix, and performing indexing and slicing operations to access elements and subsets of the array. You can run this code to see the output and understand how basic array manipulations work in NumPy.

Exercise 2B - Use Pandas to load, clean, and analyze a dataset, performing tasks such as data filtering and aggregation.

Below is an example of how you can use Pandas to load, clean, and analyze a dataset, performing tasks such as data filtering and aggregation:

```
import pandas as pd

# Step 1: Load the dataset
# Let's assume we have a CSV file named 'sales_data.csv' containing
sales data
# Replace 'sales_data.csv' with the actual path to your dataset
df = pd.read_csv('module_2_lesson_2_sampledata.csv')

# Step 2: Display the first few rows of the dataset to understand its
structure
print("First few rows of the dataset:")
print(df.head())

# Step 3: Data cleaning (if necessary)
# For example, handling missing values or converting data types

# Step 4: Data filtering
# Let's filter the dataset to include only sales data from a specific
region
region_filter = df['Region'] == 'North'
sales_data_north = df[region_filter]

# Step 5: Data aggregation
# Let's calculate the total sales amount for each product category in
the North region
sales_by_category = sales_data_north.groupby('Product Category')['Sales
Amount'].sum()

# Step 6: Display the aggregated data
print("\nTotal sales amount by product category in the North region:")
print(sales_by_category)
```

In this example, we first load the dataset using `pd.read_csv()`, then display the first few rows to understand its structure. We apply data filtering to extract sales data from a specific region ('North') and then perform data aggregation to calculate the total sales amount for each product category in that region using the `groupby()` and `sum()` functions. Finally, we display the aggregated data. You can adapt this example to your dataset and perform various other analysis tasks using Pandas.

Exercise 2C - Explore the features of the Polars library by performing data manipulation tasks and comparing their efficiency with NumPy and Pandas.

Exploring the features of the Polars library and comparing its efficiency with NumPy and Pandas involves performing various data manipulation tasks such as data loading, filtering, aggregation, and computation. Below, I'll demonstrate some basic data manipulation tasks using Polars, NumPy, and Pandas, and then compare their efficiency:

1. Data loading: Load a CSV file into each library's data structure.
2. Data filtering: Filter rows based on a condition.
3. Data aggregation: Group data by a column and calculate the sum of another column.
4. Efficiency comparison: Measure the execution time for each task.

First, you'll need to install the Polars library (`pip install polars`) if you haven't already. Then, let's proceed with the comparison:

```
import time
import pandas as pd
import numpy as np
import polars as pl

# Data loading
start_time = time.time()
# Pandas
pandas_df = pd.read_csv('module_2_lesson_3_sampledata.csv')
pandas_loading_time = time.time() - start_time

start_time = time.time()
# NumPy
numpy_arr = np.genfromtxt('module_2_lesson_3_sampledata.csv', delimiter
=',', names=True, dtype=None, encoding=None)
numpy_loading_time = time.time() - start_time

start_time = time.time()
# Polars
polars_df = pl.read_csv('module_2_lesson_3_sampledata.csv')
polars_loading_time = time.time() - start_time
```

```
# Data filtering
# Assuming we want to filter rows where a column 'value' is greater than
100
start_time = time.time()
# Pandas
pandas_filtered_df = pandas_df[pandas_df['value'] > 100]
pandas_filtering_time = time.time() - start_time

start_time = time.time()
# NumPy
numpy_filtered_arr = numpy_arr[numpy_arr['value'] > 100]
numpy_filtering_time = time.time() - start_time

start_time = time.time()
# Polars
polars_filtered_df = polars_df.filter(pl.col('value') > 100)
polars_filtering_time = time.time() - start_time

# Data aggregation
# Assuming we want to group by 'category' and sum the 'value' column
start_time = time.time()
# Pandas
pandas_aggregated_df = pandas_df.groupby('category')['value'].sum()
pandas_aggregation_time = time.time() - start_time

start_time = time.time()
# NumPy
numpy_aggregated_arr = np.sum(numpy_arr['value'][numpy_arr['category']
== b'category'])
numpy_aggregation_time = time.time() - start_time

start_time = time.time()
# Polars
polars_aggregated_df = polars_df.groupby('category').agg(pl.sum('value
'))
polars_aggregation_time = time.time() - start_time

# Print loading, filtering, and aggregation times
print("Loading time (Pandas):", pandas_loading_time)
print("Loading time (NumPy):", numpy_loading_time)
print("Loading time (Polars):", polars_loading_time)

print("\nFiltering time (Pandas):", pandas_filtering_time)
print("Filtering time (NumPy):", numpy_filtering_time)
print("Filtering time (Polars):", polars_filtering_time)

print("\nAggregation time (Pandas):", pandas_aggregation_time)
print("Aggregation time (NumPy):", numpy_aggregation_time)
print("Aggregation time (Polars):", polars_aggregation_time)
```

In this comparison, we load a CSV file, filter rows based on a condition, and perform data aggregation using Pandas, NumPy, and Polars. We measure the execution time for each task and compare their efficiency. Note that the actual execution time may vary depending on the size of the dataset and the complexity of the operations performed.

Module 3: Statistical Analysis

Upon completion of these exercises, students should be able to:

- Perform exploratory data analysis on a given dataset, including calculating descriptive statistics and visualizing distributions. *(Exercise 3A)*
- Use Statsmodels to perform linear regression analysis and hypothesis testing on a dataset. *(Exercise 3B)*
- Explore advanced statistical analysis techniques using Pingouin, including conducting correlation analysis and non-parametric tests. *(Exercise 3C)*
- Conduct hypothesis tests and statistical computations using SciPy. *(Exercise 3D)*

Exercise 3A - Perform exploratory data analysis on a given dataset, including calculating descriptive statistics and visualizing distributions.

Exploratory Data Analysis (EDA) involves understanding the data, identifying patterns, and extracting insights. Here's a step-by-step guide to performing EDA on a given dataset:

1. **Load the dataset:** Use pandas to load the dataset into a DataFrame.
2. **Inspect the dataset:** Display the first few rows (`df.head()`) to understand the structure of the data. Check for missing values (`df.info()`) and handle them if necessary.
3. **Descriptive statistics:** Calculate basic statistics such as mean, median, standard deviation, minimum, maximum, and quartiles using `df.describe()`.
4. **Visualize distributions:** Create visualizations to explore the distributions of numerical variables using histograms, box plots, and density plots. For categorical variables, use bar plots or pie charts.
5. **Explore relationships:** Use scatter plots, pair plots, or correlation matrices to understand the relationships between variables.
6. **Identify outliers:** Visualize and/or detect outliers using box plots or scatter plots.
7. **Feature engineering:** Create new features if necessary, based on domain knowledge or insights gained during exploration.

Let's illustrate these steps using a sample dataset:

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

# Step 1: Load the dataset
df = pd.read_csv('your_dataset.csv')

# Step 2: Inspect the dataset
print("First few rows of the dataset:")
print(df.head())
print("\nDataset information:")
print(df.info())

# Step 3: Descriptive statistics
print("\nDescriptive statistics:")
print(df.describe())

# Step 4: Visualize distributions
# Histograms for numerical variables
df.hist(figsize=(10, 8))
plt.suptitle("Histograms of Numerical Variables")
plt.show()

# Box plots for numerical variables
plt.figure(figsize=(10, 6))
sns.boxplot(data=df)
plt.title("Box Plot of Numerical Variables")
plt.show()

# Bar plot for categorical variables
plt.figure(figsize=(8, 6))
sns.countplot(x='category', data=df)
plt.title("Count of Observations by Category")
plt.show()

# Step 5: Explore relationships
# Pair plot for numerical variables
sns.pairplot(df)
plt.title("Pair Plot of Numerical Variables")
plt.show()

# Heatmap of correlation matrix
plt.figure(figsize=(8, 6))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Matrix")
plt.show()

# Step 6: Identify outliers
# Box plot of numerical variables
plt.figure(figsize=(10, 6))
sns.boxplot(data=df[['numeric_column1', 'numeric_column2']])
plt.title("Box Plot of Numerical Variables")
plt.show()

# Step 7: Feature engineering (if necessary)

# Further analysis based on insights gained

# Save or display visualizations
```

Replace `'your_dataset.csv'` with the path to your dataset. This code provides a basic framework for performing exploratory data analysis. You can further customize the visualizations and analysis based on the specific characteristics of your dataset.

Exercise 3B - Use Statsmodels to perform linear regression analysis and hypothesis testing on a dataset.

Below is an example code demonstrating how to perform linear regression analysis and hypothesis testing using Statsmodels on a dataset:

```python
import pandas as pd
import numpy as np
import statsmodels.api as sm

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

# Prepare the data
X = data[['feature1', 'feature2']]  # Independent variables
y = data['target']  # Dependent variable

# Add a constant term to the independent variables (intercept term)
X = sm.add_constant(X)

# Fit the linear regression model
model = sm.OLS(y, X).fit()

# Print the summary statistics
print(model.summary())

# Perform hypothesis testing
# For example, test the significance of the coefficients
print("Hypothesis Testing:")
print("Test for feature1 coefficient:")
print("Null Hypothesis: Coefficient for feature1 is zero")
print("Alternative Hypothesis: Coefficient for feature1 is not zero")
print("p-value:", model.pvalues['feature1'])
print("Conclusion: Reject the null hypothesis if p-value < significance
level (e.g., 0.05)")
```

Replace `'your_dataset.csv'` with the actual path to your dataset. This code performs the following steps:

1. Load the dataset into a DataFrame.
2. Prepare the data by separating the independent variables (features) and the dependent variable (target).

3. Add a constant term to the independent variables to fit the intercept term in the linear regression model.
4. Fit the Ordinary Least Squares (OLS) linear regression model using `sm.OLS()` and `.fit()`.
5. Print the summary statistics of the model, including coefficients, standard errors, t-values, p-values, and more.
6. Perform hypothesis testing on the coefficients to test their significance.

You can adjust the code according to your specific dataset and hypothesis testing requirements.

Exercise 3C - Explore advanced statistical analysis techniques using Pingouin, including conducting correlation analysis and non-parametric tests.

Pingouin is a Python package that provides an extensive collection of statistical methods for scientific research. It offers various advanced statistical analysis techniques, including correlation analysis and non-parametric tests. Below is an example code demonstrating how to perform correlation analysis and non-parametric tests using Pingouin:

```
import pandas as pd
import pingouin as pg

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

# Correlation Analysis
# Pearson correlation coefficient and p-value
pearson_corr = pg.corr(data['variable1'], data['variable2'], method=
'pearson')
print("Pearson correlation coefficient:")
print(pearson_corr)

# Spearman correlation coefficient and p-value
spearman_corr = pg.corr(data['variable1'], data['variable2'], method=
'spearman')
print("\nSpearman correlation coefficient:")
print(spearman_corr)

# Kendall correlation coefficient and p-value
kendall_corr = pg.corr(data['variable1'], data['variable2'], method=
'kendall')
print("\nKendall correlation coefficient:")
print(kendall_corr)

# Non-parametric tests
# Wilcoxon signed-rank test
wilcoxon_test = pg.wilcoxon(data['variable1'], data['variable2'])
print("\nWilcoxon signed-rank test:")
print(wilcoxon_test)

# Mann-Whitney U test
mannwhitneyu_test = pg.mwu(data['variable1'], data['variable2'])
print("\nMann-Whitney U test:")
print(mannwhitneyu_test)
```

Replace `'your_dataset.csv'` with the actual path to your dataset. This code performs the following tasks:

1. Loads the dataset into a DataFrame.
2. Performs correlation analysis using Pearson, Spearman, and Kendall correlation coefficients.
3. Performs non-parametric tests including the Wilcoxon signed-rank test and Mann-Whitney U test.
4. Prints the results of each analysis.

You can adjust the code according to your specific dataset and analysis requirements. Additionally, Pingouin offers many more statistical methods and tests, so feel free to explore its documentation for more options and functionalities.

Exercise 3D - Conduct hypothesis tests and statistical computations using SciPy.

```
import numpy as np
from scipy import stats

# Generate some sample data
np.random.seed(0)
sample1 = np.random.normal(loc=5, scale=2, size=100)  # Sample 1 with
mean 5 and standard deviation 2
sample2 = np.random.normal(loc=4, scale=2, size=100)  # Sample 2 with
mean 4 and standard deviation 2

# Conduct a t-test to compare means of two independent samples
t_statistic, p_value = stats.ttest_ind(sample1, sample2)
print("T-statistic:", t_statistic)
print("P-value:", p_value)

# Perform a chi-square test for independence
# Create a contingency table
observed = np.array([[20, 30], [15, 35]])  # Example contingency table
chi2_statistic, p_value, degrees_of_freedom, expected = stats
.chi2_contingency(observed)
print("\nChi-square statistic:", chi2_statistic)
print("P-value:", p_value)
print("Degrees of freedom:", degrees_of_freedom)
print("Expected frequencies:\n", expected)

# Conduct a one-way ANOVA test
# Generate some additional sample data
group1 = np.random.normal(loc=10, scale=2, size=50)  # Group 1 with mean
10 and standard deviation 2
group2 = np.random.normal(loc=12, scale=2, size=50)  # Group 2 with mean
12 and standard deviation 2
group3 = np.random.normal(loc=14, scale=2, size=50)  # Group 3 with mean
14 and standard deviation 2
# Combine all groups
all_groups = [group1, group2, group3]
f_statistic, p_value = stats.f_oneway(*all_groups)
print("\nF-statistic:", f_statistic)
print("P-value:", p_value)
```

Module 4: Web Scraping

Upon completion of these exercises, students should be able to:

- Create a Scrapy spider to scrape product information from an e-commerce website. *(Exercise 4A)*
- Use Selenium to scrape data from a dynamically loaded webpage with JavaScript content. *(Exercise 4B)*
- Implement HTML parsing and scraping with BeautifulSoup to extract information from a news article webpage. *(Exercise 4C)*
- Combine Scrapy, Selenium, and BeautifulSoup to scrape data from a website with both structured and dynamic content. *(Exercise 4D)*

Exercise 4A - Create a Scrapy spider to scrape product information from an e-commerce website.

Below is an example of a Scrapy spider that can be used to scrape product information from an e-commerce website:

```
import scrapy

class ECommerceSpider(scrapy.Spider):
    name = 'ecommerce'
    start_urls = ['https://amazon.com/products']

    def parse(self, response):
        # Extract product URLs
        product_urls = response.css('a.product-link::attr(href)'
).extract()

        # Follow each product URL and parse product details
        for product_url in product_urls:
            yield scrapy.Request(url=product_url, callback=self
.parse_product)

        # Follow pagination links
        next_page = response.css('a.next-page::attr(href)'
).extract_first()
        if next_page:
            yield scrapy.Request(url=next_page, callback=self.parse)

    def parse_product(self, response):
        # Extract product details
        product = {
            'name': response.css('h1.product-name::text').
extract_first(),
            'price': response.css('span.product-price::text'
).extract_first(),
            'description': response.css('div.product-description::text
').extract_first(),
            # Add more fields as needed
        }

        yield product
```

In this example: - The spider starts by visiting the URL specified in the `start_urls` list. - It extracts product URLs from the initial page and follows each URL to parse the details of individual products. - It also follows pagination links to scrape products from multiple pages.

- The `parse_product` method is responsible for extracting details such as product name, price, description, etc., from each product page.

You can customize this spider according to the specific structure and behavior of the e-commerce website you want to scrape. Make sure to inspect the HTML structure of the website to determine the appropriate CSS selectors or XPaths for extracting data. Additionally, be respectful of the website's terms of service and robots.txt file when scraping data.

Exercise 4B - Use Selenium to scrape data from a dynamically loaded webpage with JavaScript content.

Here's an example of how you can use Selenium to scrape data from a dynamically loaded webpage with JavaScript content:

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Start a Selenium webdriver
driver = webdriver.Chrome()

# Navigate to the webpage with dynamically loaded content
driver.get("https://example.com")

# Wait for the dynamically loaded content to appear
wait = WebDriverWait(driver, 10)
element = wait.until(EC.presence_of_element_located((By.CLASS_NAME,
"dynamic-content")))

# Extract data from the dynamically loaded content
data = element.text

# Print or process the extracted data as needed
print(data)

# Close the webdriver
driver.quit()
```

In this example: - We start a Selenium webdriver, in this case, using the Chrome webdriver. - We navigate to the webpage with dynamically loaded content using `driver.get()`. - We use WebDriverWait to wait for the dynamically loaded content to appear. Here, we wait for an element with class name "dynamic-content" to be present. - Once the dynamically loaded content appears, we extract its text using the `text` attribute of the WebElement object. - Finally, we print or process the extracted data as needed, and then close the webdriver using `driver.quit()`.

You may need to adjust the locator strategy (e.g., class name, CSS selector, XPath) used in `WebDriverWait` according to the specific structure of the webpage you are scraping.

Exercise 4C - Implement HTML parsing and scraping with BeautifulSoup to extract information from a news article webpage.

Here's an example of how you can use BeautifulSoup to scrape information from a news article webpage:

```python
import requests
from bs4 import BeautifulSoup

# URL of the news article webpage
url = "https://www.example.com/news/article"

# Send a GET request to the webpage
response = requests.get(url)

# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Parse the HTML content of the webpage using BeautifulSoup
    soup = BeautifulSoup(response.content, "html.parser")

    # Extract the title of the article
    title = soup.find("h1").text.strip()
    print("Title:", title)

    # Extract the publication date of the article
    date = soup.find("time").text.strip()
    print("Publication Date:", date)

    # Extract the author(s) of the article
    author = soup.find("div", class_="author").text.strip()
    print("Author:", author)

    # Extract the main content of the article
    content = soup.find("div", class_="article-content").text.strip()
    print("Content:", content)

    # Extract other relevant information as needed

else:
    print("Failed to retrieve the webpage.")
```

In this example: - We send a GET request to the URL of the news article webpage using the `requests.get()` function. - If the request is successful (status code 200), we parse the HTML content of the webpage using BeautifulSoup. - We then use various methods provided by BeautifulSoup, such as `find()` or `find_all()`, to locate specific elements in the

HTML document, such as the title, publication date, author, and main content of the article. - We extract the text content of these elements using the `text` attribute and strip any leading or trailing whitespace using the `strip()` method. - Finally, we print or process the extracted information as needed.

Exercise 4D - Combine Scrapy, Selenium, and BeautifulSoup to scrape data from a website with both structured and dynamic content.

Combining Scrapy, Selenium, and BeautifulSoup can be quite powerful for scraping websites with both structured and dynamic content. Here's a general approach to achieve this:

1. **Use Scrapy for web crawling**: Scrapy is great for crawling websites and extracting structured data from multiple pages. You can define the structure of the website, including the URLs to visit and the data to extract from each page.
2. **Use Selenium for interacting with dynamic content**: Selenium can be used to interact with JavaScript-driven content, such as clicking buttons, filling out forms, or scrolling through pages. This is useful for scraping data from websites that heavily rely on JavaScript to render content dynamically.
3. **Use BeautifulSoup for parsing HTML content**: Once Selenium has loaded the dynamic content, you can pass the HTML source to BeautifulSoup for parsing. BeautifulSoup provides simple and intuitive methods for extracting data from HTML documents.

Here's a basic example of how you can combine these tools:

```
import scrapy
from scrapy.selector import Selector
from scrapy_selenium import SeleniumRequest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from bs4 import BeautifulSoup

class MySpider(scrapy.Spider):
    name = "my_spider"

    def start_requests(self):
        # Start with SeleniumRequest to load the initial page
        yield SeleniumRequest(url="https://example.com", callback=self
.parse)

    def parse(self, response):
        # Use Selenium to interact with dynamic content
        # For example, clicking a button to load more content
        driver = response.meta['driver']
        load_more_button = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.XPATH, "//button[@id=
'load-more-button']"))
        )
        load_more_button.click()

        # Wait for dynamic content to load
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.XPATH, "//div[@class=
'dynamic-content']"))
        )

        # Pass the HTML source to BeautifulSoup for parsing
        soup = BeautifulSoup(driver.page_source, "html.parser")

        # Extract structured data using Scrapy Selectors
        # For example, extract product titles
        titles = Selector(text=driver.page_source).xpath("//h2[@class=
'product-title']/text()").extract()

        # Extract dynamic content using BeautifulSoup
        # For example, extract text from a div with class 'dynamic-
content'
        dynamic_content = soup.find("div", class_="dynamic-content"
).get_text()

        # Process or yield the extracted data as needed
        yield {
            'titles': titles,
            'dynamic_content': dynamic_content
        }
```

In this example: - We start by using `SeleniumRequest` to load the initial page with Selenium. - Inside the `parse` method, we use Selenium to interact with dynamic content, such as clicking a button to load more content. - Once the dynamic content is loaded, we pass the HTML source to BeautifulSoup for parsing. - We use Scrapy Selectors to extract structured data (e.g., product titles) and BeautifulSoup to extract dynamic content (e.g., text from a div with a specific class). - Finally, we process or yield the extracted data as needed.

Module 5: Natual Language Processing

Upon completion of these exercises, students should be able to:

- Perform basic text processing tasks such as tokenization and sentiment analysis using TextBlob. *(Exercise 5A)*
- Explore NLTK's text corpora and implement advanced text analysis techniques such as sentiment analysis and text classification. *(Exercise 5B)*
- Fine-tune a pre-trained BERT model for sentiment analysis on a dataset of customer reviews. *(Exercise 5C)*
- Evaluate the performance of the fine-tuned BERT model and compare it with traditional NLP techniques. *(Exercise 5D)*

Exercise 5A - Perform basic text processing tasks such as tokenization and sentiment analysis using TextBlob.

TextBlob is a Python library that provides a simple API for common natural language processing (NLP) tasks, including tokenization and sentiment analysis. Here's how you can perform these tasks using TextBlob:

1. **Tokenization**: Tokenization is the process of breaking text into individual words or tokens. TextBlob provides a `words` property that can be used to tokenize text into words.
2. **Sentiment Analysis**: Sentiment analysis is the process of determining the sentiment or opinion expressed in a piece of text. TextBlob provides a `sentiment` property that returns a tuple with two values: polarity and subjectivity. Polarity measures the sentiment's positivity or negativity (-1 to +1), while subjectivity measures how subjective or opinionated the text is (0 to 1).

Here's a basic example demonstrating how to use TextBlob for tokenization and sentiment analysis:

```
from textblob import TextBlob

# Sample text for demonstration
text = "TextBlob is a simple and easy-to-use library for text processing
tasks. I love using it!"

# Tokenization
blob = TextBlob(text)
tokens = blob.words

print("Tokens:")
print(tokens)

# Sentiment Analysis
sentiment = blob.sentiment
polarity = sentiment.polarity
subjectivity = sentiment.subjectivity

print("\nSentiment Analysis:")
print("Polarity:", polarity)
print("Subjectivity:", subjectivity)

Output:
```

```
Tokens:
['TextBlob', 'is', 'a', 'simple', 'and', 'easy-to-use', 'library', 'for
', 'text', 'processing', 'tasks', 'I', 'love', 'using', 'it']

Sentiment Analysis:
Polarity: 0.46875
Subjectivity: 0.625
[source,python]
```

In this example: - We tokenize the sample text into individual words using the `words` property of TextBlob. - We perform sentiment analysis on the sample text using the `sentiment` property of TextBlob, which returns a tuple containing polarity and subjectivity scores. - The polarity score indicates a moderately positive sentiment, while the subjectivity score indicates that the text is somewhat subjective or opinionated.

Exercise 5B - Explore NLTK's text corpora and implement advanced text analysis techniques such as sentiment analysis and text classification.

NLTK (Natural Language Toolkit) is a powerful Python library for working with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.

Here's a basic example of how you can use NLTK for sentiment analysis and text classification:

1. **Sentiment Analysis**: NLTK provides a built-in sentiment analysis module that includes pre-trained models for analyzing sentiment in text. You can use these models to classify text as positive, negative, or neutral based on the sentiment expressed in the text.
2. **Text Classification**: NLTK allows you to train your own text classifiers using machine learning algorithms such as Naive Bayes, Decision Trees, or Maximum Entropy classifiers. You can use these classifiers to categorize text into predefined classes or categories.

Below is an example that demonstrates sentiment analysis and text classification using NLTK:

```python
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
from nltk.tokenize import word_tokenize
from nltk.corpus import movie_reviews
from nltk.classify import NaiveBayesClassifier
from nltk.classify.util import accuracy

# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('movie_reviews')

# Sentiment Analysis
def analyze_sentiment(text):
    sid = SentimentIntensityAnalyzer()
    sentiment_score = sid.polarity_scores(text)['compound']
    if sentiment_score > 0:
        return 'Positive'
    elif sentiment_score < 0:
        return 'Negative'
    else:
        return 'Neutral'

# Text Classification
# Prepare training data
documents = [(list(movie_reviews.words(fileid)), category)
             for category in movie_reviews.categories()
             for fileid in movie_reviews.fileids(category)]
# Shuffle the documents
import random
random.shuffle(documents)

# Define features extractor
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = list(all_words)[:2000]

def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# Train Naive Bayes classifier
featuresets = [(document_features(d), c) for (d,c) in documents]
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = NaiveBayesClassifier.train(train_set)

# Evaluate classifier accuracy
print("Classifier Accuracy:", accuracy(classifier, test_set))

# Test sentiment analysis
text = "I love this movie. It's fantastic!"
print("Sentiment Analysis Result:", analyze_sentiment(text))

# Test text classification
text = "This movie is a masterpiece."
features = document_features(word_tokenize(text))
print("Text Classification Result:", classifier.classify(features))
```

In this example: - We use NLTK's `SentimentIntensityAnalyzer` class for sentiment analysis, which returns a compound score indicating the sentiment polarity of the text. - We use NLTK's movie reviews corpus to train a Naive Bayes classifier for text classification. The classifier is trained to classify movie reviews as positive or negative. - We evaluate the accuracy of the classifier using a test set. - Finally, we test both sentiment analysis and text classification on sample texts.

Exercise 5C - Fine-tune a pre-trained BERT model for sentiment analysis on a dataset of customer reviews.

Fine-tuning a pre-trained BERT (Bidirectional Encoder Representations from Transformers) model for sentiment analysis on a dataset of customer reviews involves several steps. BERT is a state-of-the-art NLP model developed by Google that has achieved impressive results on various natural language understanding tasks.

Here's a general outline of the process:

1. **Prepare the Dataset**: Collect or create a dataset of customer reviews along with their corresponding sentiment labels (e.g., positive, negative, neutral).
2. **Preprocess the Data**: Tokenize the text data and convert it into a format suitable for input into the BERT model. This typically involves tokenizing the text into subwords, adding special tokens such as `[CLS]` and `[SEP]`, and padding/truncating sequences to a fixed length.
3. **Load the Pre-trained BERT Model**: Use a pre-trained BERT model (e.g., `bert-base-uncased`) from the Hugging Face `transformers` library.
4. **Fine-tune the Model**: Fine-tune the pre-trained BERT model on the customer review dataset. This involves training the model on the dataset using supervised learning with a suitable loss function (e.g., cross-entropy loss) and an optimizer (e.g., Adam optimizer). During fine-tuning, you may freeze certain layers of the BERT model and/or use differential learning rates.
5. **Evaluate the Model**: Evaluate the fine-tuned model on a separate validation set to assess its performance. You can use metrics such as accuracy, precision, recall, and F1-score to evaluate the model's performance.
6. **Fine-tune Hyperparameters**: Fine-tune hyperparameters such as learning rate, batch size, number of epochs, and model architecture to improve performance.
7. **Inference**: Once the model is trained and evaluated, you can use it to make predictions on new customer reviews to classify their sentiment.

Here's a high-level Python code example using the Hugging Face `transformers` library for fine-tuning BERT for sentiment analysis:

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification,
AdamW
from torch.utils.data import DataLoader, Dataset
from sklearn.model_selection import train_test_split

# Load and preprocess dataset
class CustomerReviewDataset(Dataset):
    def __init__(self, reviews, labels, tokenizer, max_length):
        self.reviews = reviews
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.reviews)

    def __getitem__(self, idx):
        review = str(self.reviews[idx])
        label = self.labels[idx]
        encoding = self.tokenizer(review, max_length=self.max_length,
truncation=True, padding='max_length', return_tensors='pt')
        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-
uncased', num_labels=3)

# Prepare dataset
reviews = [...]  # List of customer reviews
labels = [...]   # List of sentiment labels (0: negative, 1: neutral, 2:
positive)
train_reviews, val_reviews, train_labels, val_labels = train_test_split
(reviews, labels, test_size=0.2, random_state=42)
train_dataset = CustomerReviewDataset(train_reviews, train_labels,
tokenizer, max_length=128)
val_dataset = CustomerReviewDataset(val_reviews, val_labels, tokenizer,
max_length=128)

# Initialize DataLoader
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

# Fine-tune BERT model
optimizer = AdamW(model.parameters(), lr=2e-5)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
model.to(device)

for epoch in range(5):
    model.train()
    for batch in train_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids, attention_mask=attention_mask,
labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    model.eval()
    val_accuracy = 0
    for batch in val_loader:
        with torch.no_grad():
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)
            outputs = model(input_ids, attention_mask=attention_mask)
            predictions = torch.argmax(outputs.logits, dim=1)
            val_accuracy += (predictions == labels).float().mean
```

Exercise 5D - Evaluate the performance of the fine-tuned BERT model and compare it with traditional NLP techniques.

To evaluate the performance of the fine-tuned BERT model for sentiment analysis and compare it with traditional NLP techniques, we can follow these steps:

1. **Data Preparation**: Prepare a dataset of customer reviews with corresponding sentiment labels (e.g., positive, negative, neutral).
2. **Fine-tuning BERT**: Fine-tune a pre-trained BERT model using the prepared dataset for sentiment analysis. This involves training the BERT model on the customer reviews dataset and fine-tuning the model's parameters to improve its performance on sentiment analysis tasks.
3. **Evaluation Metrics**: Select appropriate evaluation metrics for sentiment analysis, such as accuracy, precision, recall, F1-score, and ROC-AUC score.
4. **Evaluation on Test Set**: Split the dataset into training and testing sets. Use the testing set to evaluate the performance of the fine-tuned BERT model and traditional NLP techniques.
5. **Performance Comparison**: Compare the performance of the fine-tuned BERT model with traditional NLP techniques (e.g., Naive Bayes, SVM, Logistic Regression) using the selected evaluation metrics.
6. **Statistical Analysis**: Conduct statistical tests (e.g., t-test) to determine if the performance difference between the fine-tuned BERT model and traditional NLP techniques is statistically significant.
7. **Visualization**: Visualize the performance comparison results using appropriate plots or charts to provide a clear understanding of the differences in performance between the models.

Here's a basic example of how you can perform these steps using Python libraries such as Hugging Face's Transformers for BERT and Scikit-learn for traditional NLP techniques:

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from transformers import BertTokenizer, BertForSequenceClassification,
Trainer, TrainingArguments

# Step 1: Data Preparation
# Load and preprocess the dataset of customer reviews with sentiment
labels

# Step 2: Fine-tuning BERT
# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-
uncased', num_labels=3)  # Assuming 3 classes: positive, negative,
neutral

# Fine-tuning parameters
training_args = TrainingArguments(
    per_device_train_batch_size=4,
    num_train_epochs=3,
    learning_rate=2e-5,
    logging_dir='./logs',
)

# Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)

# Fine-tune BERT model
trainer.train()

# Step 3: Evaluation Metrics
def evaluate_model(model, test_dataset):
    predictions = model.predict(test_dataset)
    y_true = test_dataset['labels']
    y_pred = predictions.predictions.argmax(-1)
    accuracy = accuracy_score(y_true, y_pred)
    report = classification_report(y_true, y_pred)
    return accuracy, report

# Step 4: Evaluation on Test Set
# Split dataset into training and testing sets
train_data, test_data = train_test_split(dataset, test_size=0.2,
random_state=42)

# Step 5: Performance Comparison
# Evaluate fine-tuned BERT model
accuracy_bert, report_bert = evaluate_model(model, test_data)

# Evaluate traditional NLP techniques (e.g., Naive Bayes, SVM, Logistic
Regression)
```

```
# Step 6: Statistical Analysis
# Perform statistical tests to compare the performance of BERT with
traditional NLP techniques

# Step 7: Visualization
# Visualize the performance comparison results
```

In this example: - We fine-tune the pre-trained BERT model on the dataset of customer reviews for sentiment analysis. - We evaluate the fine-tuned BERT model using accuracy and classification report metrics. - We compare the performance of the fine-tuned BERT model with traditional NLP techniques using the same evaluation metrics. - We perform statistical tests to determine if the performance difference between the models is statistically significant. - Finally, we visualize the performance comparison results to provide insights into the differences in performance.

Module 6: Time Series Analysis

Upon completion of these exercises, students should be able to:

- Explore and visualize the characteristics of time series data, including trend, seasonality, and noise. *(Exercise 6A)*
- Use Darts to build time series forecasting models and evaluate their performance on historical data. *(Exercise 6B)*
- Perform time series decomposition and anomaly detection using Kats on real-world datasets. *(Exercise 6C)*
- Extract relevant features from time series data using Tsfresh and apply them to machine learning tasks.*(Exercise 6D)*

Exercise 6A - Explore and visualize the characteristics of time series data, including trend, seasonality, and noise.

Exploring and visualizing time series data can provide valuable insights into its characteristics, including trend, seasonality, and noise. Here's how you can do it using Python with libraries such as Pandas, Matplotlib, and Statsmodels:

1. **Load the Time Series Data**: Load your time series data into a Pandas DataFrame.
2. **Visualize the Time Series**: Plot the time series data to get an initial understanding of its overall pattern.
3. **Decompose the Time Series**: Use decomposition techniques to separate the time series into its constituent components: trend, seasonality, and residual (noise).
4. **Visualize Trend and Seasonality**: Plot the trend and seasonality components of the time series separately.
5. **Visualize Residuals (Noise):** Plot the residual component of the time series to examine any remaining patterns or irregularities.

Here's an example implementation:

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose

# Step 1: Load the Time Series Data
# Assuming 'date' is the index and 'value' is the time series data
df = pd.read_csv('your_time_series_data.csv', parse_dates=['date'],
index_col='date')

# Step 2: Visualize the Time Series
plt.figure(figsize=(10, 6))
plt.plot(df.index, df['value'], label='Original Time Series')
plt.title('Time Series Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()

# Step 3: Decompose the Time Series
decomposition = seasonal_decompose(df['value'], model='additive')
```

```
# Step 4: Visualize Trend and Seasonality
plt.figure(figsize=(10, 6))
plt.plot(df.index, decomposition.trend, label='Trend')
plt.title('Trend Component of Time Series')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(df.index, decomposition.seasonal, label='Seasonality')
plt.title('Seasonality Component of Time Series')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()

# Step 5: Visualize Residuals (Noise)
plt.figure(figsize=(10, 6))
plt.plot(df.index, decomposition.resid, label='Residuals (Noise)')
plt.title('Residual Component of Time Series')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()
```

In this example: - We load the time series data into a Pandas DataFrame. - We visualize the original time series data. - We decompose the time series into its trend, seasonality, and residual components using the seasonal decomposition method from Statsmodels. - We visualize the trend, seasonality, and residual components separately to better understand the characteristics of the time series data.

Exercise 6B - Use Darts to build time series forecasting models and evaluate their performance on historical data.

Darts is a Python library designed for easy manipulation and forecasting of time series data. It provides various models and tools for time series forecasting. Here's a step-by-step guide on how to use Darts to build time series forecasting models and evaluate their performance on historical data:

1. **Install Darts**: If you haven't already installed Darts, you can do so using pip:

   ```
   pip install darts
   ```

2. **Load the Time Series Data**: Load your time series data into a Darts TimeSeries object.
3. **Split the Data**: Split the data into training and validation sets. Typically, you'll use a portion of the data for training and keep the rest for validation to evaluate the model's performance.
4. **Choose a Forecasting Model**: Choose a forecasting model from Darts, such as Exponential Smoothing, ARIMA, Prophet, or any other model that fits your data.
5. **Fit the Model**: Fit the chosen model to the training data.
6. **Make Forecasts**: Use the fitted model to make forecasts on the validation set.
7. **Evaluate Performance**: Evaluate the performance of the forecasts using appropriate metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), or Root Mean Squared Error (RMSE).
8. **Visualize Forecasts**: Visualize the forecasts along with the actual data to see how well the model captures the underlying patterns.

Here's a basic example demonstrating these steps:

```
from darts import TimeSeries
from darts.models import ExponentialSmoothing
from darts.metrics import mape
import matplotlib.pyplot as plt

# Step 2: Load the Time Series Data
```

```
# Assuming 'date' is the index and 'value' is the time series data
# Replace 'your_time_series_data.csv' with the path to your dataset
series = TimeSeries.from_csv('your_time_series_data.csv', parse_dates
=True, index_col='date')

# Step 3: Split the Data
train, val = series.split_before(pd.Timestamp('2022-01-01'))

# Step 4: Choose a Forecasting Model and Fit the Model
model = ExponentialSmoothing()
model.fit(train)

# Step 6: Make Forecasts
forecast = model.predict(len(val))

# Step 7: Evaluate Performance
mape_error = mape(val, forecast)
print(f"MAPE: {mape_error:.2f}%")

# Step 8: Visualize Forecasts
plt.figure(figsize=(10, 6))
series.plot(label='Actual')
forecast.plot(label='Forecast', lw=2)
plt.title('Forecasting with Exponential Smoothing')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()
```

In this example: - We load the time series data into a Darts TimeSeries object. - We split the data into a training set and a validation set. - We choose an Exponential Smoothing model and fit it to the training data. - We make forecasts on the validation set. - We evaluate the forecasts using Mean Absolute Percentage Error (MAPE). - We visualize the actual data and the forecasts to assess the model's performance.

Exercise 6C - Perform time series decomposition and anomaly detection using Kats on real-world datasets.

Kats is a powerful time series analysis toolkit developed by Facebook. It provides various functionalities for time series analysis, including time series decomposition and anomaly detection. Here's a step-by-step guide on how to perform time series decomposition and anomaly detection using Kats on real-world datasets:

1. **Install Kats**: If you haven't already installed Kats, you can do so using pip:

   ```
   pip install kats
   ```

2. **Load the Time Series Data**: Load your time series data into a pandas DataFrame or a Kats TimeSeriesData object.
3. **Perform Time Series Decomposition**: Use Kats to decompose the time series into its trend, seasonality, and residual components.
4. **Detect Anomalies**: Apply anomaly detection algorithms provided by Kats to detect anomalies in the time series.
5. **Visualize Results**: Visualize the original time series, its components after decomposition, and detected anomalies to gain insights into the data.

Here's a basic example demonstrating these steps:

```
from kats.consts import TimeSeriesData
from kats.detectors.trend_mk import MKDetector
from kats.detectors.outlier import OutlierDetector
from kats.detectors.seasonal_decomposition import
TimeSeriesDecompositionDetector
import matplotlib.pyplot as plt

# Step 2: Load the Time Series Data
# Assuming 'date' is the index and 'value' is the time series data
# Replace 'your_time_series_data.csv' with the path to your dataset
# Load data into a Kats TimeSeriesData object
ts_data = TimeSeriesData.load_csv('your_time_series_data.csv')

# Step 3: Perform Time Series Decomposition
# Use seasonal decomposition to decompose the time series into trend,
```

```
seasonality, and residual components
decomposition_detector = TimeSeriesDecompositionDetector(ts_data)
decomposition_result = decomposition_detector.run()

# Step 4: Detect Anomalies
# Use an anomaly detection algorithm, such as the Mann-Kendall trend
change detector
mk_detector = MKDetector(ts_data)
mk_outliers = mk_detector.detector()

# Alternatively, you can use other outlier detection algorithms provided
by Kats
# For example, you can use an outlier detector based on z-score
# outlier_detector = OutlierDetector(ts_data)
# zscore_outliers = outlier_detector.detector()

# Step 5: Visualize Results
# Plot the original time series, its decomposition components, and
detected anomalies
plt.figure(figsize=(12, 8))

# Plot original time series
plt.subplot(3, 1, 1)
plt.plot(ts_data.time, ts_data.value)
plt.title('Original Time Series')

# Plot decomposed components
plt.subplot(3, 1, 2)
plt.plot(ts_data.time, decomposition_result.trend)
plt.plot(ts_data.time, decomposition_result.seasonal)
plt.plot(ts_data.time, decomposition_result.residue)
plt.title('Decomposed Components')

# Plot detected anomalies
plt.subplot(3, 1, 3)
plt.plot(ts_data.time, ts_data.value)
plt.scatter(mk_outliers['time'], mk_outliers['value'], color='red',
label='Anomalies')
plt.title('Anomaly Detection')
plt.legend()

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

In this example: - We load the time series data into a Kats TimeSeriesData object. - We perform time series decomposition using Kats to decompose the time series into trend, seasonality, and residual components. - We detect anomalies using the Mann-Kendall trend change detector provided by Kats. - We visualize the original time series, its decomposition components, and detected anomalies to gain insights into the data.

Exercise 6D - Extract relevant features from time series data using Tsfresh and apply them to machine learning tasks.

Tsfresh is a Python library that provides functionalities for feature extraction from time series data. It automatically extracts a large number of features from time series data, which can then be used for various machine learning tasks such as classification, regression, and clustering. Here's how you can use Tsfresh to extract relevant features from time series data and apply them to machine learning tasks:

1. **Install Tsfresh**: If you haven't already installed Tsfresh, you can do so using pip: ` `pip install tsfresh` `
2. **Prepare Your Time Series Data**: Organize your time series data into a pandas DataFrame where each column represents a different time series, and each row represents a timestamp.
3. **Extract Features:** Use Tsfresh to extract features from your time series data. Tsfresh will automatically generate a large number of features based on the provided time series data.
4. **Preprocess Features**: Preprocess the extracted features as needed, such as handling missing values or scaling the features.
5. **Apply Machine Learning Models**: Use the extracted features as input to machine learning models for tasks such as classification, regression, or clustering.

Here's a basic example demonstrating these steps:

```python
from tsfresh import extract_features
from tsfresh.utilities.dataframe_functions import impute
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd

# Step 2: Prepare Your Time Series Data
# Create a pandas DataFrame with time series data
# Each column represents a different time series
# Each row represents a timestamp
# Replace 'your_time_series_data.csv' with the path to your dataset
```

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

# Step 3: Extract Features
# Extract features from the time series data using Tsfresh
# Set the column 'id' to uniquely identify each time series
extracted_features = extract_features(df, column_id='id')

# Step 4: Preprocess Features
# Handle missing values in the extracted features
impute(extracted_features)

# Step 5: Apply Machine Learning Models
# Split the data into training and testing sets
X = extracted_features  # Features
y = df['target']         # Target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train a Random Forest classifier
clf = RandomForestClassifier()
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

In this example: - We use Tsfresh to extract features from the time series data stored in a pandas DataFrame. - We preprocess the extracted features to handle any missing values. - We split the data into training and testing sets and train a Random Forest classifier using the extracted features. - We evaluate the performance of the trained classifier on the test set using accuracy as the metric.