Module 1. Data analysis process

Welcome to course 3 of the Data Science certification. This course delves into the details of data analysis. You will learn about the data analysis process and tools, including Excel, Tableau, and Python.

This module examines data analysis and a good data analysis process. This will prepare us to explore applications, analysis, models, and visualization in the following modules.

Unit 1.1 Data analysis process

Data analysis is the systematic process of examining, cleaning, transforming, and modeling data to discover meaningful insights, draw conclusions, and support decision-making. At its core, data analysis is built on several theoretical foundations.

1. Statistical theory: the mathematical framework that enables us to understand patterns, relationships, and uncertainties in data.
2. Information theory: principles governing how information is measured, stored, and communicated
3. Computer science: algorithms and computational methods for processing and analyzing large datasets.
4. Domain knowledge: understanding of the specific field where the analysis is being applied.

The fundamental goal of data analysis is to transform raw data into actionable insights that can inform decision-making.

The data analysis process follows a structured approach with several key steps. Let's review them below.

1. Problem definition

- Clearly defining the research question or business problem.
- Identifying key objectives and desired outcomes.
- Determining what decisions will be made based on the analysis.
- Setting measurable success criteria.

2. Data collection

- Identifying relevant data sources.
- Gathering data through appropriate methods (surveys, databases, sensors, etc.).
- Documenting data collection procedures.
- Ensuring data quality at the source.

3. Data cleaning and preparation

- Handling missing values.
- Removing duplicates.
- Correcting inconsistencies.
- Standardizing formats.
- Dealing with outliers.
- Creating derived variables when necessary.

4. Data exploration

- Understanding data distribution and basic statistics.
- Identifying patterns and relationships.
- Detecting anomalies.
- Formulating initial hypotheses.

5. Data analysis

- Applying appropriate statistical methods.
- Testing hypotheses.
- Building and validating models.
- Performing sensitivity analysis.

In the previous certificate modules, we discussed data collection, cleaning and preparation in detail. In this module, we will focus on data exploration and analysis.

Unit 1.2 Data exploration

Once we have defined the problem, acquired the appropriate data, and finished with the pre-processing, cleaning, etc., the next important step is data exploration. It involves investigating and understanding your dataset's characteristics, patterns, and potential issues before conducting more sophisticated analyses. Here are a few essential techniques and best practices for exploring data, with practical examples.

1.2.1 Data assessment

The first step is to load your data and understand its basic structure.

- For Excel, you open the .csv or .xlsx in Excel and peruse all the columns and the data types in the data.
- For Tableau, like Excel, you connect the data source to a Tableau, click the Data tab, and see the data preview.
- It takes a bit more effort in Python, but this work won't go to waste.

Figure 1. Data assessment

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

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

# Display basic information about the dataset
print("Dataset Info:")
print(df.info())

# View the first few rows
print("\nFirst 5 rows:")
print(df.head())

# Get basic statistics
print("\nBasic statistics:")
print(df.describe())
```

Source: own elaboration.

1.2.2 Data quality

- Check data for missing values and duplicates.

Figure 2. Data quality

```
# Check for missing values
missing_values = df.isnull().sum()
print("Missing values per column:")
print(missing_values)

# Check for duplicates
duplicates = df.duplicated().sum()
print(f"\nNumber of duplicate rows: {duplicates}")
```

Source: own elaboration.

1.2.3 Distribution of the variables

Plot all the numerical variables, examine their distributions, and understand their characteristics well.

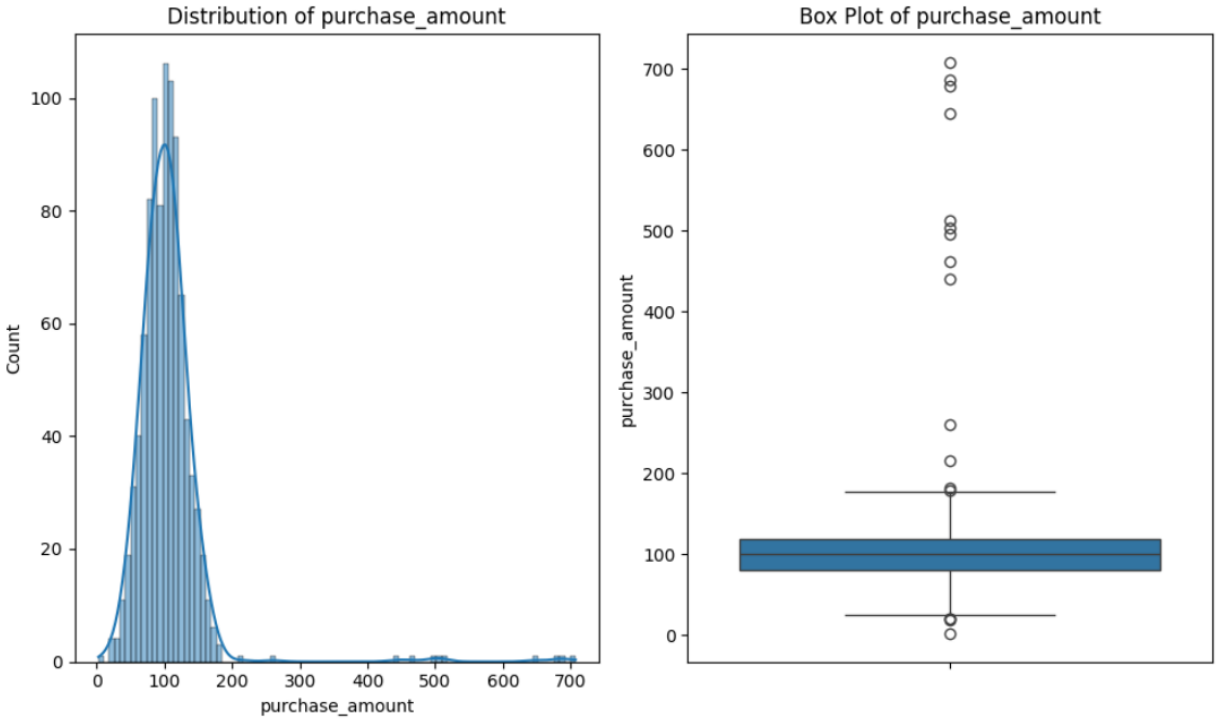
Figure 3. Distribution of the variables

```
def plot_numerical_distribution(df, column):
    plt.figure(figsize=(10, 6))

    # Histogram
    plt.subplot(1, 2, 1)
    sns.histplot(df[column], kde=True)
    plt.title(f'Distribution of {column}')

    # Box plot
    plt.subplot(1, 2, 2)
    sns.boxplot(y=df[column])
    plt.title(f'Box Plot of {column}')

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

Source: own elaboration.

Here is the plot of the distribution of the purchase amount.

Figure 4. Plot of the distribution of the purchase amount



Source: own elaboration.

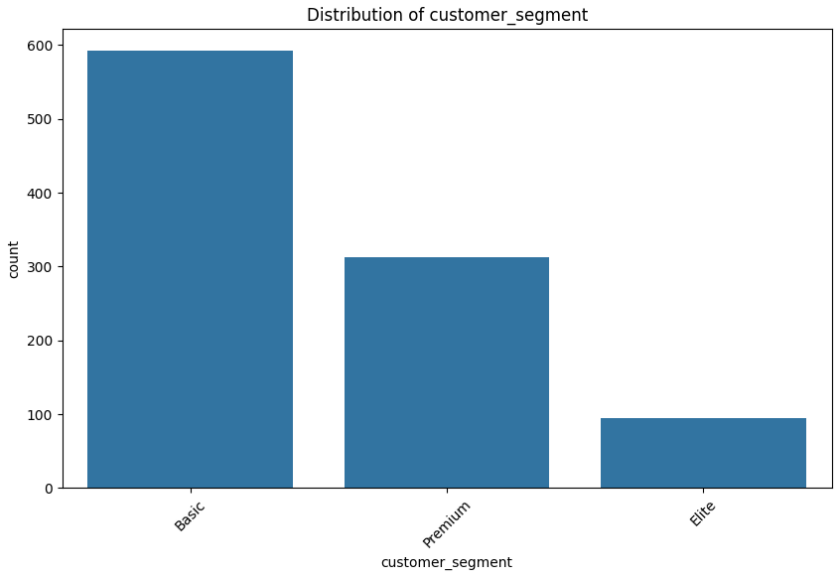
Figure 5. Plot and analyze the categorical variables

```
def analyze_categorical(df, column):
    # Value counts and percentage
    counts = df[column].value_counts()
    percentages = df[column].value_counts(normalize=True) * 100

    print(f"\nDistribution of {column}:")
    for value, count in counts.items():
        print(f"{value}: {count} ({percentages[value]:.2f}%)")

    # Bar plot
    plt.figure(figsize=(10, 6))
    sns.countplot(data=df, x=column)
    plt.title(f'Distribution of {column}')
    plt.xticks(rotation=45)
    plt.show()
```

Source: own elaboration.

Here is the plot of the distribution of the various categories of customer segment.

Figure 6. Distribution of customer segment



Source: own elaboration.

1.2.4 Relationship analysis

Create a correlation matrix and examine the relationships between all the different variables.

A correlation matrix helps you understand your dataset's relationships between numerical variables. It shows:

1. the strength of relationships between variables (from -1 to +1),
2. the direction of relationships (positive or negative),
3. which variables might be redundant or highly dependent on each other.

For example, in a housing dataset, a correlation matrix might reveal that square footage strongly correlates positively with price. In contrast, distance from the city center negatively correlates with price.

Cross tabulation helps you:

1. analyze relationships between categorical variables,
2. understand the distribution of data across different combinations of categories,
3. identify patterns or trends that might not be visible in single-variable analysis.

For instance, you might use cross-tabulation in a customer dataset to examine how purchase behavior varies across different age groups and gender categories, revealing that young females are your highest-spending demographic.

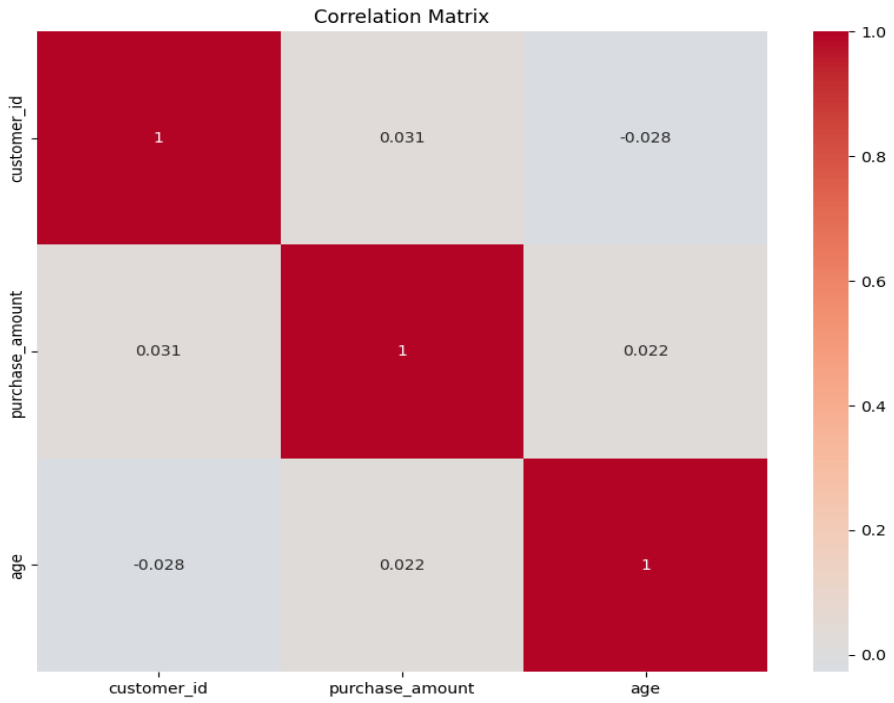
Figure 7. Correlation matrix

```
# Calculate correlation matrix
correlation_matrix = df.select_dtypes(include=['float64', 'int64']).corr()

# Create heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Matrix')
plt.show()
```

Source: own elaboration.

Here is the correlation matrix between all the numerical variables.

Figure 8. Correlation matrix between all the numerical variables



Source: own elaboration.

Figure 9. Analyze categorical relationship

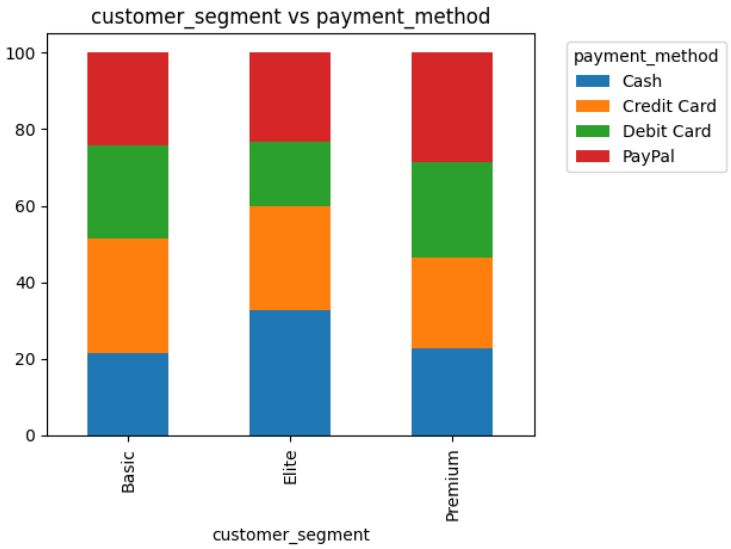
```python
def analyze_categorical_relationship(df, cat1, cat2):
    # Create cross-tabulation
    ctab = pd.crosstab(df[cat1], df[cat2], normalize='index') * 100

    print(f"\nRelationship between {cat1} and {cat2}:")
    print(ctab)

    # Stacked bar plot
    plt.figure(figsize=(10, 6))
    ctab.plot(kind='bar', stacked=True)
    plt.title(f'{cat1} vs {cat2}')
    plt.legend(title=cat2, bbox_to_anchor=(1.05, 1))
    plt.tight_layout()
    plt.show()

# Example usage
analyze_categorical_relationship(df, 'customer_segment', 'payment_method')
```

Source: own elaboration.

Here is the output of the categorical relationships between customer segments (basic, elite, premium) and payment methods used by them (cash, credit card, debit card, PayPal).

Figure 10. Output of the categorical relationships between customer segments and payment methods used by them

```
Relationship between customer_segment and payment_method:
payment_method         Cash  Credit Card  Debit Card     PayPal
customer_segment
Basic             21.621622    29.729730   24.324324  24.324324
Elite             32.631579    27.368421   16.842105  23.157895
Premium           22.683706    23.642173   25.239617  28.434505
```



Source: own elaboration.

1.2.5 Anomaly detection and outliers

The next step in the data exploration process is to look for anomalies and outliers.

Anomaly and outlier detection identifies data points, events, or observations that deviate significantly from the expected normal behavior or pattern. Looking for things that "don't quite fit" in your data.

Where is anomaly and outlier detection helpful?

- **Fraud detection in financial transactions.** Transactions of unusually high amounts or an abnormally high number of transactions in a short period that isn't normal.
- **Quality control in manufacturing.** Finding defective products by detecting measurements outside normal ranges.
- **Network intrusion detection.** Identifying suspicious login patterns or unauthorized login patterns.
- **Medical diagnosis anomalies**. Finding irregular heartbeat patterns in ECG data.

These are just a few examples, but there are many more applications.

Best practices:

1. Always investigate outliers before removing them, because outliers could be once-in-a-generation talents.

For example, Lionel Messi's shot-to-goal conversion ratio is an outlier of 2-3 standard deviations above an average elite football attacker.

2. Consider domain knowledge when setting thresholds.
3. Use multiple detection methods for validation.
4. Document your rationale for handling outliers.

This code below lets you detect outliers in a dataset. Anything below the lower bound or above the upper bound (as defined below) is categorized as an outlier.

Figure 11. Detect outliers in a dataset

```python
def detect_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = df[(df[column] < lower_bound) | (df[column] > upper_bound)][column]

    print(f"\nOutliers in {column}:")
    print(f"Number of outliers: {len(outliers)}")
    print(f"Percentage of outliers: {(len(outliers)/len(df))*100:.2f}%")
    print(f"Outlier values:\n{outliers}")

# Example usage
detect_outliers(df, 'purchase_amount')
```

Source: own elaboration.

Here is the output of the above code when run on sample data.

Figure 12. Output of the code when run on sample data

```
Outliers in purchase_amount:
Number of outliers: 17
Percentage of outliers: 1.70%
Outlier values:
74       21.407647
91      645.296749
```

Source: own elaboration.

There are 17 total outliers according to our definition of outliers in the code. Percentage of outliers is 17/1000 = 1.7 %.
(The output has been truncated to save space).

1.2.6 Time series analysis

Check to see if your data has a time series component. Then, look at the date columns in the data, format, analyze, and visualize them.

Figure 13. Time series analysis

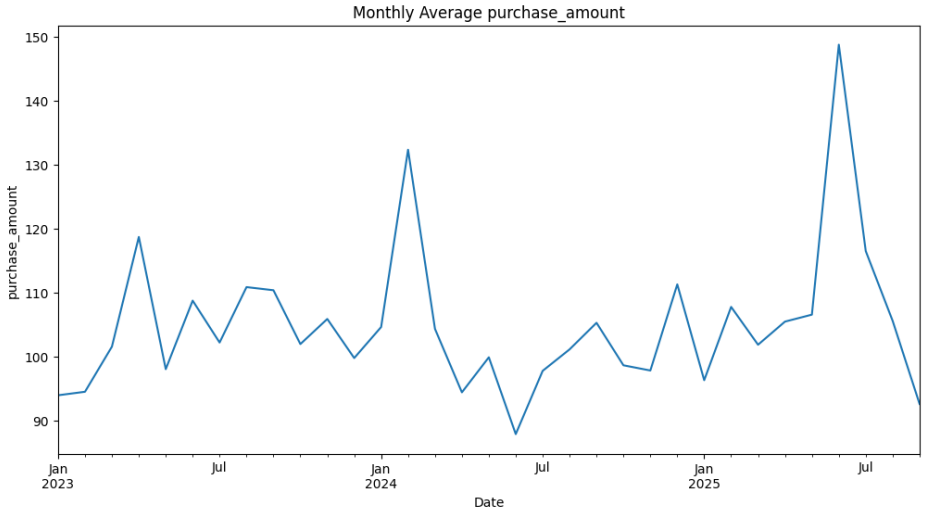
```
def analyze_time_series(df, date_column, value_column):
    # Convert date column to datetime
    df[date_column] = pd.to_datetime(df[date_column])

    # Set date as index
    ts_df = df.set_index(date_column)

    # Resample to monthly frequency and plot
    monthly = ts_df[value_column].resample('M').mean()

    plt.figure(figsize=(12, 6))
    monthly.plot()
    plt.title(f'Monthly Average {value_column}')
    plt.xlabel('Date')
    plt.ylabel(value_column)
    plt.show()

    # Calculate year-over-year growth
    yoy_growth = monthly.pct_change(12) * 100
    print(f"\nYear-over-year growth rates:")
    print(yoy_growth.tail())

# Example usage
analyze_time_series(df, 'purchase_date', 'purchase_amount')
```

Source: own elaboration.

The above example shows a graph of monthly average revenue over time. It also shows how much revenue increased or decreased compared to the same month in the previous year.

Here is the output visualization of the code above.

Figure 14. Output visualization



Source: own elaboration.

The .tail() output would show the most recent five months of year-over-year growth rates, which would be helpful in understanding recent performance trends.

Figure 15. The most recent five months of year-over-year growth rates

```
Year-over-year growth rates:
purchase_date
2025-05-31     6.668444
2025-06-30    69.199589
2025-07-31    19.137970
2025-08-31     4.395285
2025-09-30   -12.047298
```

Source: own elaboration.

1.2.7 Data exploration – best practices

1. Start with data quality checks.
2. Document the exploration process.
3. Create visualizations to tell a story.
4. Look for patterns and anomalies.
5. Business context.

1.2.8 Data exploration – common pitfalls

1. Skipping initial data quality checks.
2. Ignoring missing values.
3. Not handling outliers appropriately.
4. Making assumptions without validation.
5. Overlooking meaningful relationships.

Data exploration is an iterative process requiring technical skills and domain knowledge. By following these techniques and best practices, you can gain valuable insights into your data and prepare it for more advanced analyses.

Unit 1.3 Data analysis

Now that we have our clearly defined problem statement or question to answer and have gone through the steps of data exploration, we get to the most fun part: analyzing the data.

Data analysis involves applying various statistical and analytical methods to extract insights from the data and try to answer the question we want to answer.

It is common to spend the bulk of your time (70-80 %) getting to this point. The analysis doesn't take nearly as much time as cleaning, exploring, and fine-tuning the dataset. The secret to getting better at data analysis is to practice it repeatedly. The more datasets you analyze, the better and faster you get at it.

1.3.1 Descriptive statistics

Descriptive statistics summarize and help you understand the basic features of your dataset. Here is an example of an NBA basketball player statistics dataset. This code snippet summarizes them by calculating their mean, median, standard deviation, etc.

Figure 16. Descriptive statistics

```python
import pandas as pd
import numpy as np
from scipy import stats

# Sample basketball player statistics dataset
data = pd.DataFrame({
    'points': [25, 18, 30, 22, 28, 32, 15, 27, 24, 29],
    'rebounds': [10, 8, 12, 7, 9, 11, 6, 8, 10, 13],
    'assists': [5, 7, 4, 8, 6, 9, 5, 7, 6, 8],
    'minutes_played': [32, 28, 35, 30, 33, 36, 25, 31, 29, 34],
    'field_goal_pct': [0.48, 0.45, 0.52, 0.47, 0.50, 0.53, 0.44, 0.49, 0.51, 0.54]
})

# Basic statistical measures
def analyze_descriptive_stats(df, column):
    stats_dict = {
        'mean': df[column].mean(),
        'median': df[column].median(),
        'std': df[column].std(),
        'min': df[column].min(),
        'max': df[column].max(),
        'skewness': df[column].skew(),
        'kurtosis': df[column].kurtosis()
    }
    return pd.Series(stats_dict)

# Example usage
points_stats = analyze_descriptive_stats(data, 'points')
print("Player Scoring Statistics:\n", points_stats)
```

Figure 17. Code output

```
Player Scoring Statistics:
 mean        25.000000
median       26.000000
std           5.395471
min          15.000000
max          32.000000
skewness     -0.710944
kurtosis     -0.254187
dtype: float64
```

Source: own elaboration.

1.3.2 Hypothesis testing

Hypothesis testing is a statistical method used to make inferences about populations based on sample data. It starts with two hypotheses.

1. Null hypothesis (H_0): usually states there's no difference/relationship.

2. Alternative hypothesis (H_1): states there is a difference/relationship.

We use statistical tests to determine if we can reject the null hypothesis.

The p-value helps make this decision (commonly using α=0.05 threshold).

In the example below, we implement two standard hypothesis tests.

1. T-test (stats.ttest_ind):
    - Tests if two groups have different means.
    - Assumes data is usually distributed.
    - Good for comparing things like average scores or measurements.
2. Mann-Whitney U test (stats.mannwhitneyu):
    - Non-parametric alternative to t-test.
    - Doesn't assume normal distribution.
    - Tests if one group tends to have higher values than the other.

The function does the following.

1. Takes two groups of data and test type as input.
2. Performs either t-test or Mann-Whitney test.
3. Returns and prints:
    - test statistic,
    - P-value,
    - whether the result is significant ($p < 0.05$).

In the example usage:

- Compares points scored in the first half vs. the second half of games.
- If p-value < 0.05, suggests a significant difference between halves.
- If p-value → 0.05, suggests no significant difference.

For example, if players consistently score more in the second half, this test would help determine if that difference is statistically significant or just due to random chance.

Figure 18. Hypothesis testing

```python
def perform_hypothesis_test(group1, group2, test_type='t-test'):
    """
    Perform statistical hypothesis testing

    Parameters:
    group1, group2: Data samples to compare
    test_type: Type of test to perform ('t-test' or 'mann-whitney')
    """
    if test_type == 't-test':
        statistic, p_value = stats.ttest_ind(group1, group2)
        test_name = "Independent t-test"
    elif test_type == 'mann-whitney':
        statistic, p_value = stats.mannwhitneyu(group1, group2)
        test_name = "Mann-Whitney U test"

    print(f"\n{test_name} Results:")
    print(f"Statistic: {statistic:.4f}")
    print(f"P-value: {p_value:.4f}")
    print(f"Significant at α=0.05: {p_value < 0.05}")

# Example: Compare player performance in first half vs second half of games
first_half_points = data['points'][:5]
second_half_points = data['points'][5:]
perform_hypothesis_test(first_half_points, second_half_points)
```

Source: own elaboration.

Figure 19. Code output (independent t-test)

```
Independent t-test Results:
Statistic: -0.2217
P-value: 0.8301
Significant at α=0.05: False
```

Source: own elaboration.

1.3.3 Correlation analysis

Correlation analysis helps identify relationships between variables in the data. Correlation measures how two variables move about each other. The values range from -1 to +1:

- +1 means perfect positive correlation (as one increases, the other increases),
- -1 means perfect negative correlation (as one increases, the other decreases),
- 0 means no correlation.

We've examined this briefly in the data exploration section. Let's use a sports example to discuss it in more detail.

Figure 20. Correlation analysis

```
def analyze_correlations(df, method='pearson'):
    """
    Analyze correlations between variables

    Parameters:
    df: DataFrame containing variables
    method: Correlation method ('pearson', 'spearman', or 'kendall')
    """
    # Calculate correlation matrix
    corr_matrix = df.corr(method=method)

    # Calculate p-values for correlations
    def calculate_pvalue(x, y, method=method):
        if method == 'pearson':
            return stats.pearsonr(x, y)[1]
        elif method == 'spearman':
            return stats.spearmanr(x, y)[1]
        elif method == 'kendall':
            return stats.kendalltau(x, y)[1]

    p_values = pd.DataFrame(
        [[calculate_pvalue(df[c1], df[c2]) for c1 in df.columns] for c2 in df.columns],
        columns=df.columns,
        index=df.columns
    )

    return corr_matrix, p_values

# Example usage
corr_matrix, p_values = analyze_correlations(data)
```

Source: own elaboration.

Suppose we have a dataset with player statistics including:

- points per game (PPG),
- minutes played,
- field goal percentage (FG %),
- free throw attempts (FTA),
- assists,
- rebounds.

Figure 21. Dataset with player statistics

```
# Example basketball data use case
basketball_stats = pd.DataFrame({
    'points_per_game': [25.3, 20.1, 18.5, 22.4, 19.8],
    'minutes_played': [35.2, 31.5, 28.4, 33.7, 30.2],
    'field_goal_pct': [48.5, 45.2, 43.8, 47.1, 44.9],
    'free_throw_attempts': [8.5, 6.2, 5.8, 7.4, 6.8],
    'assists': [6.8, 4.5, 3.2, 5.7, 4.1],
    'rebounds': [5.2, 7.8, 6.4, 4.9, 8.1]
})

# Analyze correlations
corr_matrix, p_values = analyze_correlations(basketball_stats)
```

Source: own elaboration.

Figure 22. Code output (correlation matrix)

```
Correlation Matrix:
                points  rebounds   assists  minutes_played  field_goal_pct
points        1.000000  0.788092  0.234439        0.962767        0.913890
rebounds      0.788092  1.000000  0.063277        0.806053        0.897000
assists       0.234439  0.063277  1.000000        0.237619        0.263391
minutes_played  0.962767  0.806053  0.237619        1.000000        0.853249
field_goal_pct  0.913890  0.897000  0.263391        0.853249        1.000000

P-values:
                points  rebounds   assists  minutes_played  field_goal_pct
points        0.000000  0.006771  0.514433        0.000008        0.000217
rebounds      0.006771  0.000000  0.862134        0.004863        0.000434
assists       0.514433  0.862134  0.000000        0.508576        0.462167
minutes_played  0.000008  0.004863  0.508576        0.000000        0.001693
field_goal_pct  0.000217  0.000434  0.462167        0.001693        0.000000
```

Source: own elaboration.

This analysis would reveal relationships like:

1. strong positive correlation between minutes played and points scored (more playing time → more points),
2. moderate correlation between free throw attempts and points (players who attack the basket get more FTs and points),
3. possible correlation between assists and minutes (more playing time → more opportunities for assists).

The p-values would tell us which of these relationships are statistically significant. For example:

- p-value < 0.05: the correlation between minutes and points is significant.
- p-value → 0.05: the correlation between rebounds and field goal percentage might be due to chance.

This could help coaches and analysts:

- identify which factors most strongly influence scoring,
- understand player efficiency (points per minute),
- make decisions about playing time and strategy,
- find unexpected relationships in player performance.

1.3.4 Regression analysis

Regression analysis is a statistical method for modeling relationships between dependent and independent variables. This piece of code provides an example in basketball.

Regression:

- Tries to find the best mathematical relationship between variables.
- Can help predict one variable based on others.
- Shows how much each factor influences the outcome.

For example, in basketball, this could predict points scored based on the following:

- Minutes played.

- Field goal percentage.
- Previous performance.
- Other relevant stats.

Figure 23. Regression analysis

```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error
import matplotlib.pyplot as plt

def perform_regression_analysis(X, y, test_size=0.2):
    """
    Perform linear regression analysis

    Parameters:
    X: Independent variables
    y: Dependent variable
    test_size: Proportion of data to use for testing
    """
    # Split data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, random_state=42
    )

    # Create and train the model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate metrics
    r2 = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))

    # Print results
    print("\nRegression Results:")
    print(f"R-squared: {r2:.4f}")
    print(f"RMSE: {rmse:.4f}")
    print("\nCoefficients:")
    for feature, coef in zip(X.columns, model.coef_):
        print(f"{feature}: {coef:.4f}")
    print(f"Intercept: {model.intercept_:.4f}")

    return model

# Example usage: Predict points based on minutes played and field goal percentage
X = data[['minutes_played', 'field_goal_pct']]
y = data['points']
regression_model = perform_regression_analysis(X, y)
```

Source: own elaboration.

Figure 24. Code output (regression results)

```
Regression Results:
R-squared: 0.7144
RMSE: 1.6033

Coefficients:
minutes_played: 1.3017
field_goal_pct: 24.8773
Intercept: -27.9995
```

Source: own elaboration.

Let's break down what this code does.

1. **Data preparation:**

   X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

- Splits data into training (80 %) and testing (20 %) sets.
- X contains predictors (e.g., minutes, FG %).
- y contains target (e.g., points scored).

2. **Model training:**

   model = LinearRegression()

   model.fit(X_train, y_train)

- Creates a linear regression model.
- Finds the best-fit line through the data.
- Example: Points = (β_1 × Minutes) + (β_2 × FG %) + Intercept.

3. **Making predictions:**

   y_pred = model.predict(X_test)

- Uses model to predict points for test data.
- Can estimate how many points a player might score given their minutes and FG %.

4. **Evaluating performance:**

r2 = r2_score(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test,y_pred))

- R-squared: how well the model fits (0-1, higher is better).
- RMSE: average prediction error in the same units as points.

5. **Results analysis:**

for feature, coef in zip(x.columns, model.coef_):

print(f"{feature}: {coef:.4f}")

- Shows how each factor affects points.
- Example: "minutes_played: 0.5" means each extra minute = 0.5 more points.

Real basketball example

X = data[['minutes_played', 'field_goal_pct']]
y = data['points']
regression_model = perform_regression_analysis(X, y)

This could show:

- how minutes played affect scoring,
- impact of shooting percentage on points,
- predictive power for future performance,
- which factors matter most for scoring.

1.3.5 Cluster analysis

Cluster analysis is a method of grouping similar data points. Let's explain it using a basketball example.

Concept:

- Groups similar items (players in this case) into clusters.
- Players within a cluster are more similar to each other than to players in other clusters.
- Helps identify patterns and player types.

This code implements K-means clustering. It takes player statistics as input.

- Groups players into a specified number of clusters.
- Identifies different playing styles or player types.

Figure 25. Cluster analysis

```python
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

def perform_clustering(data, n_clusters=3):
    """
    Perform K-means clustering

    Parameters:
    data: DataFrame containing features for clustering
    n_clusters: Number of clusters to create
    """
    # Standardize the features
    scaler = StandardScaler()
    scaled_data = scaler.fit_transform(data)

    # Perform clustering
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    clusters = kmeans.fit_predict(scaled_data)

    # Add cluster assignments to original data
    result_df = data.copy()
    result_df['Cluster'] = clusters

    # Calculate cluster statistics
    cluster_stats = result_df.groupby('Cluster').agg(['mean', 'std'])

    print("\nCluster Statistics:")
    print(cluster_stats)

    return result_df, kmeans

# Example usage: Cluster players based on their playing style
clustering_features = data[['points', 'rebounds', 'assists']]
clustered_data, kmeans_model = perform_clustering(clustering_features)
```

Source: own elaboration.

Figure 26. Code output (cluster statistics)

```
Cluster Statistics:
            points                rebounds                assists
              mean       std        mean       std        mean       std
Cluster
0         18.333333  3.511885    7.000000  1.000000   6.666667  1.527525
1         29.333333  2.516611   10.666667  2.516611   8.000000  1.000000
2         26.750000  2.753785   10.250000  1.258306   5.250000  0.957427
```

Source: own elaboration.

Let us break down the code step by step.

1. **Data standardization:**

scaler = StandardScaler()

scaled_data = scaler.fit_transform(data)

- Standardizes stats to the same scale.
- Prevents one stat (like points) from dominating just because it has larger numbers.
- Example: converting 25 PPG and 5 APG to comparable scales.

2. **Clustering:**

   kmeans = KMeans(n_clusters=n_clusters, random_state=42)

   clusters = kmeans.fit_predict(scaled_data)

- Groups players into n_clusters (3 in this example).
- Each player is assigned to the nearest cluster center.

3. **Result analysis:**

   result_df['Cluster'] = clusters

   cluster_stats = result_df.groupby('Cluster').agg(['mean', 'std'])

- Shows average stats for each cluster.
- Helps identify cluster characteristics.

Basketball example:

clustering_features = data[['points', 'rebounds', 'assists']]

clustered_data, kmeans_model = perform_clustering(clustering_features)

This might identify player types like:

- Cluster 0: scoring guards (high points, low rebounds, moderate assists).
- Cluster 1: big men (moderate points, high rebounds, low assists).
- Cluster 2: all-around players (moderate in all categories).

Some applications of clustering analysis:

- Player comparison.
- Team composition analysis.
- Finding similar players.
- Strategic matchup planning.

1.3.6 Data analysis – best practices

When performing data analysis, it's essential to:

1. choose appropriate methods based on your data type and research questions,
2. validate assumptions required by each method,
3. interpret results in the context of your domain knowledge,
4. document all steps and decisions made during the analysis,
5. consider the limitations and potential biases in your analysis.

Document and make it reproducible

- Maintain detailed documentation of all steps and decisions.
- Use version control for code and datasets.
- Create reproducible analysis workflows.
- Document assumptions and limitations.

Ensure data quality

- Validate data accuracy and completeness.
- Implement quality control checks.
- Maintain data lineage.
- Regular data audits.

Maintain statistical rigor

- Use appropriate statistical methods.
- Account for uncertainties and biases.
- Validate assumptions.
- Conduct sensitivity analyses.
- Cross-validate results.

Respect data privacy and confidentiality

- Protect data privacy and confidentiality.
- Obtain necessary permissions and consents.
- Consider potential biases in data and analysis.
- Be transparent about limitations and uncertainties.

1.3.7 What is the easiest way to learn Python and test the code in this module?

This can be done in multiple ways. You can use any one of the free online resources. Start with the tutorials at https://docs.python.org/3/tutorial/.

Recommended environment to run and test Python code online

1. Google Colab – https://colab.google.com
    - Runs Python code in your browser
    - Includes code examples and Gemini helper AI bot
    - Works great for learning data science

If you prefer to run Python locally on your computer, we recommend:

1. Download python from https://www.python.org
2. Install an IDE like Visual Studio Code – it has excellent support for Python, including the beneficial auto "code complete" feature.

The other option is to use Jupyter Notebooks instead of Visual Studio Code. Jupyter Notebooks are like Google Colab notebooks, but all the code and Python are run locally.

After installing Python from python.org, install Jupyter using pip (the Python package manager). Then, open your terminal or command prompt to run.

```
pip install jupyter
```

After Jupyter is installed, open a Jupyter notebook.

```
jupyter notebook
```

Conclusion

In this module, we learned about the different steps of data analysis, focusing on data exploration and analysis. We looked at how to look at data to find the shape and structure of data. Look at the outliers, missing values, and anomalies. We also looked at code samples using different analysis techniques.

In the upcoming modules, we will focus on data visualization and picking the appropriate technique based on the data type, its size, and the type of problem we are solving.

References

Python. (n.d.). The Python Tutorial. https://docs.python.org/3/tutorial/