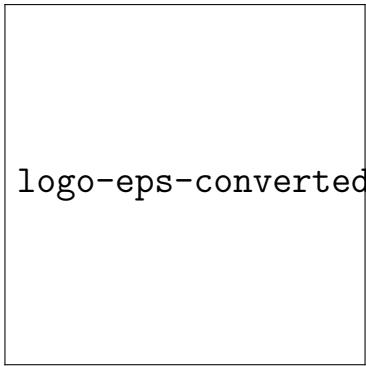


Statistics & Data Manipulation

Complete Guide with Python



logo-eps-converted-to.pdf

August 7, 2025

Contents

1	Introduction	3
2	Class codes	3
3	Part 1: Data Manipulation with Pandas	4
3.1	Loading and Inspecting Data	4
3.2	Handling Missing Data	5
3.3	Data Selection and Filtering	5
3.4	Grouping and Aggregation	6
4	Part 2: Descriptive Statistics	6
4.1	Measures of Central Tendency	6
4.2	Measures of Dispersion	6
5	Part 3: Data Visualization with Seaborn	7
6	Part 4: Inferential Statistics with SciPy	7
6.1	Core Concepts	7
6.2	T-test: Comparing Means of Two Groups	7
7	Part 5: Introduction to Machine Learning with Scikit-learn	8
7.1	The Scikit-learn API	8
7.2	Essential Preprocessing for ML	8
7.3	Example: Predicting Salary with Linear Regression	8
8	Conclusion	9

1 Introduction

Statistics is the science of collecting, analyzing, interpreting, presenting, and organizing data. In the modern era of data science and machine learning, a strong foundation in statistics is essential. Data manipulation, the process of cleaning and transforming raw data, is the prerequisite for any meaningful statistical analysis.

This handout provides a practical walkthrough of core statistical concepts and data manipulation techniques using Python, the de facto language for data science. We will leverage powerful libraries to make these tasks efficient and intuitive.

- **Pandas:** The primary library for data manipulation and analysis. It provides data structures like the DataFrame, which is perfect for handling tabular data.
- **NumPy:** The fundamental package for numerical computation in Python, forming the bedrock upon which Pandas is built.
- **Matplotlib & Seaborn:** Libraries for data visualization, crucial for exploring data and presenting results.
- **SciPy:** A library for scientific and technical computing, which includes a robust statistics module.
- **Scikit-learn (sklearn):** The premier library for machine learning in Python, offering tools for preprocessing, modeling, and evaluation.

2 Class codes

```

1 import numpy as np
2 import pandas as pd
3 from sklearn.impute import SimpleImputer
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.preprocessing import MinMaxScaler
6 from sklearn.decomposition import PCA
7
8 np.random.seed(42)
9 data = np.random.randn(500, 4) * [10, 5, 1, 0.5] + [50, 30, 10, 5]
10 df = pd.DataFrame(data,
11                     columns=['Feature1', 'Feature2', 'Feature3', 'Feature4'])
12 df.iloc[10:15, 2] = np.nan # Introduce missing values in 'Feature3'
13
14 # 2. Create an imputer instance (mean strategy)
15 imputer = SimpleImputer(strategy='mean')
16
17 # 3. Fit the imputer to Feature3 and transform
18 df[['Feature3']] = imputer.fit_transform(df[['Feature3']])
19
20 scaler = StandardScaler()
21 scaled_data = scaler.fit_transform(df)
22 df_scaled = pd.DataFrame(scaled_data, columns=df.columns)
23
24 # MinMax Norm
25 minmax_scaler = MinMaxScaler(feature_range=(0, 1))
26 scaled_data_mm = minmax_scaler.fit_transform(df)
27 df_minmax = pd.DataFrame(scaled_data_mm, columns=df.columns)
28
29 # Adding Outlier into the data
30 outliers = np.array([[200, 300, 20, 10],

```

```

31             [180, 280, 25, 7],
32             [160, -100, -5, 2]])
33 outliers_df = pd.DataFrame(outliers, columns=df.columns)
34 df = pd.concat([df, outliers_df], ignore_index=True)
35
36
37 # Sample data (replace with your data)
38 X = np.array([[1, 2], [2, 4], [3, 6], [4, 8]])
39
40 # 1. Standardize the data
41 scaler = StandardScaler()
42 X_scaled = scaler.fit_transform(X)
43
44 # 2. Create a PCA object and specify the number of components
45 pca = PCA(n_components=1) # Reduce to 1 dimension
46
47 # 3. Fit the PCA model and transform the data
48 X_reduced = pca.fit_transform(X_scaled)
49
50 # Outlier Detection
51 from sklearn.ensemble import IsolationForest
52
53 iso_forest = IsolationForest(
54     contamination=0.01, # Expect 1% anomalies
55     n_estimators=100, # Use 100 trees for better stability
56     max_samples=256, # Use 256 random samples per tree
57     max_features=2, # Consider only 2 features per split
58     random_state=42 # Ensure reproducibility
59 )
60
61 outlier_labels = iso_forest.fit_predict(df) # 1 for inlier, -1 for outlier
62
63 df['Outlier'] = outlier_labels
64 df['Outlier'].value_counts()
65
66 df_no_outliers = df[df['Outlier'] == 1].drop(columns='Outlier')
67 df_no_outliers.shape

```

3 Part 1: Data Manipulation with Pandas

Before we can analyze data, we must load, inspect, and clean it. Pandas makes this process straightforward.

3.1 Loading and Inspecting Data

The first step is always to load your data into a Pandas DataFrame. The most common format is CSV (Comma-Separated Values).

```

1 import pandas as pd
2 import numpy as np
3
4 # Create a sample DataFrame for demonstration
5 data = {
6     'age': [25, 32, 45, 21, np.nan, 32, 50],
7     'department': ['HR', 'Engineering', 'Marketing', 'Engineering', 'HR', 'HR',
8         'Marketing'],
8     'salary': [50000, 80000, 75000, 48000, 52000, 55000, 90000],
9     'years_experience': [2, 8, 15, 1, 3, 9, 20]

```

```

10 }
11 df = pd.DataFrame(data)
12
13 # In a real scenario, you would load from a file:
14 # df = pd.read_csv('your_data.csv')
15
16 # --- Inspecting the Data ---
17
18 # View the first 5 rows
19 print("First 5 rows (head):")
20 print(df.head())
21
22 # Get a concise summary of the DataFrame
23 print("\nDataFrame Info:")
24 df.info()
25
26 # Get summary statistics for numerical columns
27 print("\nDescriptive Statistics:")
28 print(df.describe())

```

3.2 Handling Missing Data

Real-world data is often messy and contains missing values (represented as ‘NaN’ in Pandas).

```

1 # Check for missing values in each column
2 print("Missing values per column:")
3 print(df.isnull().sum())
4
5 # Option 1: Drop rows with any missing values
6 df_dropped = df.dropna()
7 # print(df_dropped)
8
9 # Option 2: Fill missing values (Imputation)
10 # We can fill the missing age with the mean or median age
11 mean_age = df['age'].mean()
12 df_filled = df.fillna({'age': mean_age})
13 print("\nDataFrame after filling missing age with the mean:")
14 print(df_filled)

```

3.3 Data Selection and Filtering

We often need to select subsets of our data for analysis.

```

1 # Selecting a single column (returns a Series)
2 salaries = df_filled['salary']
3
4 # Selecting multiple columns (returns a DataFrame)
5 subset = df_filled[['age', 'salary']]
6
7 # Filtering rows based on a condition
8 high_earners = df_filled[df_filled['salary'] > 70000]
9 print("\nHigh earners (salary > 70000):")
10 print(high_earners)
11
12 # Filtering with multiple conditions
13 hr_high_earners = df_filled[(df_filled['salary'] > 50000) & (df_filled['department'] ==
   ↪ 'HR')]
14 print("\nHR employees with salary > 50000:")

```

```
15 print(hr_high_earners)
```

3.4 Grouping and Aggregation

The ‘groupby()’ operation is one of the most powerful features of Pandas. It allows you to split data into groups, apply a function to each group, and combine the results.

```
1 # Group by department and calculate the mean salary for each
2 avg_salary_by_dept = df_filled.groupby('department')['salary'].mean()
3 print("\nAverage salary by department:")
4 print(avg_salary_by_dept)
5
6 # You can apply multiple aggregation functions at once
7 dept_stats = df_filled.groupby('department').agg({
8     'salary': ['mean', 'min', 'max'],
9     'age': 'mean'
10 })
11 print("\nDetailed stats by department:")
12 print(dept_stats)
```

4 Part 2: Descriptive Statistics

Descriptive statistics summarize the central tendency, dispersion, and shape of a dataset’s distribution.

4.1 Measures of Central Tendency

- **Mean:** The average value. Sensitive to outliers. Formula: $\mu = \frac{1}{N} \sum_{i=1}^N x_i$
- **Median:** The middle value of a sorted dataset. Robust to outliers.
- **Mode:** The most frequently occurring value.

4.2 Measures of Dispersion

- **Variance:** The average of the squared differences from the Mean. Formula: $\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2$
- **Standard Deviation:** The square root of the variance. It measures the amount of variation or dispersion. Formula: $\sigma = \sqrt{\sigma^2}$
- **Range:** The difference between the maximum and minimum value.
- **Quartiles & IQR:** Values that divide the data into four equal parts. The Interquartile Range (IQR = Q3 - Q1) measures the spread of the middle 50% of the data.

```
1 # Using our filled DataFrame (df_filled)
2 print("Mean salary:", df_filled['salary'].mean())
3 print("Median salary:", df_filled['salary'].median())
4 print("Standard deviation of salary:", df_filled['salary'].std())
5 print("Full description (includes quartiles):")
6 print(df_filled['salary'].describe())
```

5 Part 3: Data Visualization with Seaborn

Visualizing data is key to understanding distributions, relationships, and outliers. Seaborn is a high-level library built on Matplotlib that is tightly integrated with Pandas.

```

1 import matplotlib.pyplot as plt
2 import seaborn as sns
3
4 # Set a nice style for the plots
5 sns.set_style("whitegrid")
6
7 # 1. Histogram: To see the distribution of a single numerical variable
8 plt.figure(figsize=(10, 6))
9 sns.histplot(df_filled['salary'], kde=True, bins=5)
10 plt.title('Distribution of Salaries')
11 plt.xlabel('Salary')
12 plt.ylabel('Frequency')
13 plt.show() # Use plt.show() to display the plot in a script
14
15 # 2. Box Plot: To visualize quartiles and detect outliers
16 plt.figure(figsize=(10, 6))
17 sns.boxplot(x='department', y='salary', data=df_filled)
18 plt.title('Salary Distribution by Department')
19 plt.show()
20
21 # 3. Scatter Plot: To see the relationship between two numerical variables
22 plt.figure(figsize=(10, 6))
23 sns.scatterplot(x='years_experience', y='salary', data=df_filled, hue='department',
24                  s=100)
25 plt.title('Salary vs. Years of Experience')
26 plt.show()
```

6 Part 4: Inferential Statistics with SciPy

Inferential statistics allows us to make inferences about a population based on a sample of data. This often involves hypothesis testing.

6.1 Core Concepts

- **Hypothesis Testing:** A method to test a claim or hypothesis about a population parameter.
- **Null Hypothesis (H_0):** The default assumption, often stating there is no effect or no difference (e.g., the mean salary of two departments is the same).
- **Alternative Hypothesis (H_1):** The claim we want to test (e.g., the mean salary of two departments is different).
- **p-value:** The probability of observing our data (or something more extreme) if the null hypothesis were true. A small p-value (typically < 0.05) provides evidence against the null hypothesis.

6.2 T-test: Comparing Means of Two Groups

The independent t-test checks if there is a significant difference between the means of two independent groups.

```

1 from scipy import stats
2
3 # Let's test if the mean salary of the Engineering department is different from HR
4 eng_salaries = df_filled[df_filled['department'] == 'Engineering']['salary']
5 hr_salaries = df_filled[df_filled['department'] == 'HR']['salary']
6
7 # Perform the independent t-test
8 t_statistic, p_value = stats.ttest_ind(eng_salaries, hr_salaries)
9
10 print(f"\nT-test results:")
11 print(f"T-statistic: {t_statistic:.2f}")
12 print(f"P-value: {p_value:.3f}")
13
14 # Interpret the result
15 alpha = 0.05
16 if p_value < alpha:
17     print("The difference in mean salaries is statistically significant (reject H0).")
18 else:
19     print("The difference in mean salaries is not statistically significant (fail to
→ reject H0).")

```

7 Part 5: Introduction to Machine Learning with Scikit-learn

Scikit-learn ('sklearn') is the cornerstone of machine learning in Python. It builds upon the data manipulation and statistical concepts we've discussed.

7.1 The Scikit-learn API

Scikit-learn has a consistent and simple API:

- **Estimator objects:** Every algorithm is its own object (e.g., 'LinearRegression', 'StandardScaler').
- **.fit(X, y):** The method used to train the model on data 'X' (features) and 'y' (target).
- **.predict(X):** After training, this method makes predictions on new data.
- **.transform(X):** For preprocessing objects, this method transforms the data.

7.2 Essential Preprocessing for ML

Machine learning models often require data to be in a specific format.

- **Feature Scaling:** Many models perform better when numerical features are on a similar scale. 'StandardScaler' standardizes features by removing the mean and scaling to unit variance.
- **Encoding Categorical Variables:** Models can't process text labels. We need to convert them to numbers. 'OneHotEncoder' converts a categorical variable into a new set of binary (0/1) columns.

7.3 Example: Predicting Salary with Linear Regression

Let's build a simple model to predict salary based on age, department, and years of experience.

```

1 from sklearn.model_selection import train_test_split
2 from sklearn.linear_model import LinearRegression
3 from sklearn.preprocessing import StandardScaler, OneHotEncoder
4 from sklearn.compose import ColumnTransformer
5 from sklearn.pipeline import Pipeline
6 from sklearn.metrics import mean_squared_error
7
8 # --- 1. Prepare Data ---
9 # Define features (X) and target (y)
10 X = df_filled[['age', 'department', 'years_experience']]
11 y = df_filled['salary']
12
13 # Identify categorical and numerical features
14 categorical_features = ['department']
15 numerical_features = ['age', 'years_experience']
16
17 # --- 2. Create a Preprocessing Pipeline ---
18 # This handles scaling and encoding in one step
19 preprocessor = ColumnTransformer(
20     transformers=[(
21         ('num', StandardScaler(), numerical_features),
22         ('cat', OneHotEncoder(), categorical_features))])
23
24 # --- 3. Create the Full Model Pipeline ---
25 # This chains the preprocessing and the linear regression model
26 model_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
27                               ('regressor', LinearRegression())])
28
29 # --- 4. Split Data into Training and Testing Sets ---
30 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
31                                                     random_state=42)
32
33 # --- 5. Train the Model ---
34 model_pipeline.fit(X_train, y_train)
35 print("\nModel training complete.")
36
37 y_pred = model_pipeline.predict(X_test)
38
39 mse = mean_squared_error(y_test, y_pred)
40 rmse = np.sqrt(mse)
41 r2 = model_pipeline.score(X_test, y_test) # R-squared
42
43 print(f"Root Mean Squared Error (RMSE) on test set: {rmse:.2f}")
44 print(f"R-squared on test set: {r2:.2f}")

```

8 Conclusion

This handout has guided you through the fundamental workflow of a data analysis project. You started with raw data and learned how to manipulate and clean it using **Pandas**. You then calculated **descriptive statistics** and created **visualizations** to understand the data's properties. Following that, you performed a **hypothesis test** to make statistical inferences. Finally, you integrated all these concepts to preprocess data and train a **machine learning model** with **Scikit-learn**.

Mastering these tools and concepts is the foundation for tackling more complex problems in data science, machine learning, and quantitative research. The key to proficiency is practice, so apply these techniques to your own datasets.