KUTAISI INTERNATIONAL UNIVERSITY

BACHELOR'S THESIS IN

Computer Science

Statistics with `Python`: From Theory to Practice

Aleksandre Gordeladze, Saba Beridze
Giorgi Machitidze

Kutaisi 2025

Statistics with `Python`: from theory to practice
Analysis of Financial and Educational Infrastructure in Georgian Schools

Bachelor Thesis

for Bachelor of Computer Science
"Bachelor of Science" (BSc.)

Kutaisi International University

Program of Computer of Science
of Kutaisi International University

submitted by

Saba Beridze, Aleksandre Gordeladze
and Giorgi Machitidze

Kutaisi 2025

Advisor:

Prof. Dr. George Nadareishvili
School of Mathematics and Computer Science, Kutaisi International University

Reviewer:

Prof. Dr. Giorgi Chelidze
School of Mathematics and Computer Science, Kutaisi International University

Members of the Examination Commission:

Prof. Dr. ——— ———-
School of Mathematics and Computer Science, Kutaisi International University
Prof. Dr. ——— ———-
School of Mathematics and Computer Science, Kutaisi International University
Prof. Dr. ——— ———-
School of Mathematics and Computer Science, Kutaisi International University

Day of defense: –.–.2025

Declaration

We declare that this thesis has been composed solely by ourselves and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where stated otherwise by reference or acknowledgment, the work presented is entirely our own.

Acknowledgment

We express our gratitude to Professor George Nadareishvili for his invaluable guidance and support throughout the research process. His encouragement and advice greatly helped us in completing this project.

Our special thanks go to the Educational and Scientific Infrastructure Development Agency (ESIDA)[7], particularly to Mrs. Medea Shamugia, who provided the infrastructure-related database. Her passion for our project and effort to support us were truly remarkable.

We also want to thank Mrs. Mariam Tabatadze from the Ministry of Education, Science, Culture, and Sport of Georgia[11]. Her assistance and collaboration were invaluable in providing the necessary resources.

Contents

Chapter 1

Introduction

Background and objective of research

Data analysis and statistical methods play a significant role in understanding, gaining insights, and solving complex problems across various fields. This thesis focuses on how statistical methods and functions are incorporated in programming languages, specifically in `Python,` with a concentration on practical applications. As a real-world example, we analyze data from the educational sector in Georgia. We apply various methods to school utility bills, teacher salaries, and other similar components. We present the theory for statistical methods and demonstrate how they can be implemented to uncover trends, optimize resource distribution, and propose improvements in real-world settings.

While the example dataset focuses on Georgian schools, the main objective is to delve into the implementation and effectiveness of statistical methods such as regression, distribution, deviation, and others. The primary aim of our research is to provide a framework for applying these methods through programming, making it relevant for students and professionals interested in the computational aspects of statistical analysis or any curious mind who is interested in using modern statistical methods for modeling and prediction from data.

Overview of The Thesis Structure

Chapter 2 begins with a theoretical background and objectives, followed by a comprehensive definition of the problem to be addressed. This includes an introduction to the statistical methodologies and programming languages used to carry out the analysis. Key statistical notions such as preliminary statistics, deviation, expected value, distribution, and regression are defined, characterized, and examined in relation to their computational implementation.

Afterward, Chapter 3, analysis of the Georgian school data is presented using `Python` programming language, showcasing how these methods and functions can be applied to real-life problems. The outcomes of the analysis are then presented.

The full code and project details can be accessed on GitHub Repository.

Group Contributions

- **Saba Beridze:**

  - Developed the intuition behind Linear Regression concepts.
  - Implemented Linear regression.
  - Conducted missing value analysis.
  - Reformatted categorical data.
  - Merged tables for infrastructure conditions and revenue.
  - Performed regional budget predictions.

- **Giorgi Machitidze:**
  - Covered preliminary statistical concepts.
  - Explained ridge regression.
  - Restructured the data tables.
  - Conducted budget prediction regression analysis.

- **Aleksandre Gordeladze:**
  - Detailed multivariate regression.
  - Merged tables for student numbers and building details.
  - Performed financial data restructuring.
  - Conducted outlier detection.
  - Implemented regression for null financial values.

Chapter 2

Elementary statistics and regression

2.1 Preliminaries

The present section recalls some fundamental probabilistic and statistical concepts. This section is useful not only for interpreting results but also for employing statistical methodologies, particularly in linear regression analysis, which will be discussed in subsequent sections. Our goal is to state several key definitions that will be used in the analyses presented throughout this paper. We will provide necessary definitions and key points, including those related to random variables, expectations, variances, covariance, distribution, and correlation. In this chapter, we will usually not provide complete proofs for theorems and propositions. Interested readers can look up definitions and detailed proofs, for example, in [4] or [21].

By Ω we denote the sample space of an experiment that is the set of all possible outcomes or results of that experiment.

In practice, we always encounter cases where the sample space is finite; thus, for our purpose, it suffices for definitions and theorems to be formulated for finite sample spaces.

One of the most important concepts in probability and statistical theory is that of a random variable. A random variable can be defined as follows:

Definition 2.1.1. A random variable X is a function $X\colon \Omega \to \mathbb{R}$ from a sample space Ω to the real numbers.

When an experiment is performed, we are often interested in some function of the outcome rather than the actual outcome itself.

Example 2.1.2. when tossing dice, we may be interested in the sum of the two dice rather than the individual values shown on each die. Specifically, we might want to know that the sum is 5 without taking into account whether the actual outcomes were $(1,4), (2,3), (3,2)$, or $(4,1)$. Similarly, when flipping a coin, we may focus on the total number of tails that occur rather than the specific head-tail sequence that results.

These quantities of interest, more formally referred to as real-valued functions, defined on the sample space are random variables. Since the value of a random variable is determined by the outcome of an experiment, we can assign probabilities to its possible values. As our sample space is finite, the random variable also takes on a finite number of real values on this sample space.

Definition 2.1.3. Let P be a probability measure on some sample space Ω, and let X be a real-valued random variable defined on Ω. The probability mass function $p_X : \mathbb{R} \to [0, 1]$ of X is defined by

$$p(a) \coloneqq P\{X = a\} \quad \text{for all } a \in \mathbb{R}.$$

Notation 2.1.4. We will simply write p instead of p_X to avoid clutter when the context makes it clear which random variable is being referenced.

Thus, $p(a)$ represents the probability that X takes on value a.

Lemma 2.1.5. *If a is not one of the possible values of the random variable X, then the probability mass function satisfies*

$$p(a) = 0.$$

Furthermore, if the sequence $a_1, a_2, \ldots, a_n$ includes all possible values of X, then it follows that

$$\sum_{i=1}^{n} p(a_i) = 1.$$

Definition 2.1.6. If X is a random variable with a probability mass function $p(x)$, then the expectation or expected value of X, denoted by $E[X]$, is defined by

$$E[X] = \sum_{i=1}^{n} x_i p(x_i).$$

In other words, the expected value of X is a weighted average of the possible values that X can assume, with each value being weighted by the probability that X assumes it.

Now we list some properties of the expectation of the random variable that will be used later. For any real-valued function g, the expected value can be expressed as

$$E[g(X)] = \sum_{i} g(x_i) p(x_i).$$

If X and Y are independent random variables, then the expectation satisfies the following property

$$E[XY] = E[.X]E[Y].$$

If a and b are constants, then the expectation of a linear transformation of X is given by

$$E[aX + b] = aE[X] + b.$$

Furthermore, if $E[X_i]$ is finite for all $i = 1, \ldots, n$, then the expectation of the sum of these random variables can be expressed as

$$E[X_1 + \cdots + X_n] = E[X_1] + \cdots + E[X_n].$$

We would like to motivate the next definition with a simple example. Consider three random variables X, Y, and Z with their respective probability mass functions. Let $X = 0$ with probability 1, Y take values as follows:

$$Y = \begin{cases} 1 & \text{with probability } 0.5, \\ -1 & \text{with probability } 0.5. \end{cases}$$

Let Z take values as follows:

$$Z = \begin{cases} 1000 & \text{with probability } 0.5, \\ -1000 & \text{with probability } 0.5. \end{cases}$$

In this case, the expected values of these random variables are all equal to 0; however, there is a significantly greater spread in the possible values of the random variables. Since we expect X to assume values around its mean $E[X]$, it seems reasonable to measure the possible variation of X by examining how far X is from its mean, on average. One way to measure this variation is to consider the quantity $E[|X - \mu|]$, where $\mu = E[X]$. However, it turns out to be mathematically inconvenient to work with this quantity. Therefore, a more tractable measure is usually considered; namely, the expectation of the square of the difference between X and its mean. This brings us to the following definition.

Definition 2.1.7. if X is a random variable with a mean μ. Then the variance $\mathrm{Var}(X)$ of X is defined as

$$\mathrm{Var}(X) = E[(X - \mu)^2].$$

It can be shown that the formula above is equivalent to

$$\mathrm{Var}(X) = E[X^2] - (E[X])^2. \tag{2.1.1}$$

For any constants a and b, the variance of a linear transformation of a random variable is given by

$$\mathrm{Var}(aX + b) = a^2 \,\mathrm{Var}(X). \tag{2.1.2}$$

The square root of the variance $\mathrm{Var}(X)$ is referred to as the standard deviation of X, denoted by $\mathrm{SD}(X)$. This is a measure of how spread out the values of a dataset or a random variable are from the mean. Thus,

$$\mathrm{SD}(X) := \sqrt{\mathrm{Var}(X)}.$$

Another useful concept in probability theory is the cumulative distribution function (CDF). We discuss this function and its properties.

Definition 2.1.8. For a random variable X, the function F defined by

$$F(x) = P\{X \leq x\}, \quad \text{for all } x \in \mathbb{R}$$

It is called the cumulative distribution function, or simply the distribution function of X.

In other words, the distribution function specifies the probability that the random variable is less than or equal to x for all real values of x.

The cumulative distribution function F can also be expressed in terms of the probability mass function $p(a)$

$$F(a) = \sum_{x \leq a} p(x).$$

Variance provides a measure of how a single random variable deviates from its mean. However, when dealing with two random variables, we require a concept that can describe how these variables vary together. This leads us to the concept of *covariance.*

Consider two random variables X and Y, and observe their behavior relative to their respective means $E[X]$ and $E[Y]$. A natural way to quantify their joint relationship is to measure how their deviations from their means are related. For each realization of X and Y, the deviations

$X - E[X]$ and $Y - E[Y]$ capture how far the values of X and Y are from their expected values. By multiplying these deviations $(X - E[X])$ and $(Y - E[Y])$, we can assess whether the variables tend to move in the same direction or in opposite directions. To summarize this behavior over all possible values of X and Y, we take the expected value of these products of deviations. The resulting concept of covariance would be of three types.

- **Positive Covariance**: when one variable increases, the other variable tends to increase as well, and vice versa.
- **Negative Covariance**: when one variable increases, the other variable tends to decrease.
- **Zero Covariance**: there is no linear relationship between the two variables.

According to the above, the following definition of covariance is reasonable.

Definition 2.1.9. The covariance $\mathrm{Cov}(X, Y)$ between two random variables X and Y is defined as

$$\mathrm{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])].$$

By expanding the right side of this definition, we can derive the following expression

$$\mathrm{Cov}(X, Y) = E[XY - E[X]Y - XE[Y] + E[Y]E[X]].$$

This simplifies to

$$\mathrm{Cov}(X, Y) = E[XY] - E[X]E[Y]. \tag{2.1.3}$$

It is important to note that if X and Y are independent, then $\mathrm{Cov}(X, Y) = 0$. However, the converse is not necessarily true. To illustrate this with a simple example, let X be a random variable taking values {1, 0, -1} and such that $P(1) = P(0) = P(-1) = \frac{1}{3}$. Let Y be another random variable defined as follows

$$Y = \begin{cases} 1 & \text{if } X = 0, \\ 0 & \text{if } X \neq 0. \end{cases}$$

It is clear that X and Y are not independent random variables. The expected value of X can be calculated as

$$E(X) = \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot (-1) = 0.$$

Since $XY = 0$ either $X = 0$ or $Y = 0$, so $E(XY) = 0$. Therefore, we compute

$$\mathrm{Cov}(X, Y) = E(XY) - E(X)E(Y) = 0 - E(Y) \cdot 0 = 0.$$

The following proposition lists some important properties of covariance.

Proposition 2.1.10. *Let X, Y, X_i and Y_j for $i = 1, \ldots, n$, $j = 1, \ldots, m$ be random variables. Then*

1. $\mathrm{Cov}(X, Y) = \mathrm{Cov}(Y, X)$;
2. $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$;
3. $\mathrm{Cov}(aX, Y) = a\ \mathrm{Cov}(X, Y)$;
4. $\mathrm{Cov}\left(\sum_{i=1}^{n} X_i, \sum_{j=1}^{m} Y_j\right) = \sum_{i=1}^{n} \sum_{j=1}^{m} \mathrm{Cov}(X_i, Y_j)$.

We will not discuss the details of these properties; the first two properties follow directly from the definition of covariance. From properties 2 and 4, we can derive the variance of the sum of random variables. If we set $Y_j = X_j$ in the property 4, we obtain

$$\begin{aligned}\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) &= \mathrm{Cov}\left(\sum_{i=1}^{n} X_i, \sum_{j=1}^{n} X_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{Cov}(X_i, X_j) \\ &= \sum_{i=1}^{n} \mathrm{Var}(X_i) + \sum\sum_{i\neq j} \mathrm{Cov}(X_i, X_j).\end{aligned}$$

Since each pair of indices i, j (where $i \neq j$) appears twice in the double summation, the preceding formula can be rewritten as

$$\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{Var}(X_i) + 2\sum_{i=1}^{n-1} \sum_{i<j<=n} \mathrm{Cov}(X_i, X_j). \tag{2.1.4}$$

Although, $\mathrm{Cov}(X, Y)$ gives a numerical measure of the degree to which X and Y vary together, the magnitude of $\mathrm{Cov}(X, Y)$ is also influenced by the overall magnitudes of X and Y. In order to obtain a measure of association between X and Y that is not driven by arbitrary changes in the scales of one or the other random variable, we define a slightly different quantity next.

Definition 2.1.11. The correlation $\rho(X, Y)$ between two random variables X and Y, is defined as

$$\rho(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sqrt{\mathrm{Var}(X)\ \mathrm{Var}(Y)}}.$$

Remark 2.1.12. Note that both $\mathrm{Var}(X) > 0$ and $\mathrm{Var}(Y) > 0$.

Correlation is a standardized measure of the strength and direction of the linear relationship between two variables. It is derived from covariance and ranges between -1 and 1. Unlike covariance, which only indicates the direction of the relationship, correlation provides a standardized measure

Theorem 2.1.13. *The correlation of two random variables is always within the interval* $[-1, 1]$.

Proof. Let X and Y be two random variables with respective variances σ_1^2 and σ_2^2. Define a new random variable Z such that

$$Z = \frac{X}{\sigma_1} + \frac{Y}{\sigma_2}.$$

By definition, the variance of Z is always non-negative, thus by equation (2.1.4) we have

$$0 \leq \mathrm{Var}(Z) = \mathrm{Var}\left(\frac{X}{\sigma_1} + \frac{Y}{\sigma_2}\right) = \mathrm{Var}\left(\frac{X}{\sigma_1}\right) + \mathrm{Var}\left(\frac{Y}{\sigma_2}\right) + 2 \cdot \mathrm{Cov}\left(\frac{X}{\sigma_1}, \frac{Y}{\sigma_2}\right).$$

From equation (2.1.2) and from proposition (2.1.10, part 4 this expands to

$$0 \leq \frac{\mathrm{Var}(X)}{\sigma_1^2} + \frac{\mathrm{Var}(Y)}{\sigma_2^2} + 2\rho(X, Y).$$

Given that $\sigma_1^2 = \mathrm{Var}(X)$ and $\sigma_2^2 = \mathrm{Var}(Y)$, we can simplify this to

$$0 \leq 2 + 2\rho(X, Y).$$

From this inequality, it follows that

$$\rho(X, Y) \geq -1.$$

To prove that $\rho(X, Y) \leq 1$, we can similarly use

$$0 \leq \text{Var}\left(\frac{X}{\sigma_1} - \frac{Y}{\sigma_2}\right). \qquad \square$$

Remark 2.1.14. Note that if $\rho(X, Y) = -1$, then $\text{Var}(Z) = 0$, which implies that Z is a constant random variable. Therefore, there exists constant c such that,

$$\frac{X}{\sigma_1} + \frac{Y}{\sigma_2} = c.$$

Consequently, for constants a and b, we can write

$$Y = a + bX,$$

where $b = -\sigma_2/\sigma_1 < 0$. If $\rho(X, Y) = 1$, we similarly can express Y as

$$Y = a + bX,$$

but in this case, $b = \sigma_2/\sigma_1 > 0$. In general, the correlation coefficient is a measure of the degree of linearity between X and Y. Specifically, it indicates how closely the values of X and Y follow a straight-line (linear) pattern. A value of $\rho(X, Y)$ near 1 or -1 indicates a high degree of linearity between X and Y, whereas a value near 0 suggests that such linearity is absent. A positive value of $\rho(X, Y)$ indicates that Y tends to increase when X does, while a negative value indicates that Y tends to decrease when X increases. If $\rho(X, Y) = 0$, which occurs when $\text{Cov}(X, Y) = 0$, then X and Y are said to be uncorrelated [21].

When dealing with multiple random variables, it is sometimes useful to use vector and matrix notations. This makes the formulas more compact and lets us use facts from linear algebra.

Definition 2.1.15. When we have n random variables $X_1, X_2, \ldots, X_n$, we can put them in a (column) vector

$$X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}.$$

We call X a random vector.

Definition 2.1.16. The expected value vector or the mean vector of the random vector X is defined as

$$E[X] = \begin{bmatrix} E[X_1] \\ E[X_2] \\ \vdots \\ E[X_n] \end{bmatrix}.$$

Definition 2.1.17. The variance of an n-dimensional random vector $X = (X_1, X_2, \ldots, X_n)^\top$ is defined as the $n \times n$ covariance matrix, where the entry in the i-th row and j-th column is given by the covariance between X_i and X_j. Formally

$$\text{Var}(X) = E\left[(X - E[X])(X - E[X])^\top\right].$$

where the diagonal entries represent the variances of the individual components of X, and the off-diagonal entries represent the covariances between different components.

$$\mathrm{Var}(X) = \begin{bmatrix} \mathrm{Var}(X_1) & \mathrm{Cov}(X_1, X_2) & \cdots & \mathrm{Cov}(X_1, X_n) \\ \mathrm{Cov}(X_2, X_1) & \mathrm{Var}(X_2) & \cdots & \mathrm{Cov}(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Cov}(X_n, X_1) & \mathrm{Cov}(X_n, X_2) & \cdots & \mathrm{Var}(X_n) \end{bmatrix}$$

Linearity of expectation also carries over to random vectors.

Lemma 2.1.18. *Let $a \in \mathbb{R}^m$, $G \in \mathbb{R}^{m \times n}$, and let z be a random n-vector. Then*

$$E[a + Gz] = a + GE[z]$$

and

$$\mathrm{Var}[a + Gz] = G\ \mathrm{Var}[z]\, G^T.$$

Now, we are ready to state one of the most fundamental results in the theory of linear regression (discussed extensively in 2.2), known as the *Gauss-Markov Theorem*.

Theorem 2.1.19 (Gauss-Markov)**.** *Suppose we have, in matrix notation, a linear relationship:*

$$y = X\theta + \epsilon, \quad (y, \epsilon \in \mathbb{R}^m,\ \theta \in \mathbb{R}^n,\ X \in \mathbb{R}^{m \times n}),$$

where:

- *y is the vector of target values,*
- *X is the matrix of explanatory variables (design matrix),*
- *θ is the vector of unknown parameters (coefficients),*
- *ϵ is the vector of error terms.*

Then, under the assumptions:

- *$E[\epsilon_i] = 0$ (errors have zero mean),*
- *$Var(\epsilon_i) = \sigma^2$ (errors have constant variance and are uncorrelated), $\quad i = 1, 2, \ldots, m,$*

the Ordinary Least Squares (OLS) estimator of θ, given by:

$$\hat{\theta} = (X^\top X)^{-1} X^\top y,$$

is the Best Linear Unbiased Estimator (BLUE), meaning it has the smallest variance among all linear and unbiased estimators.

Proof (see [23]). Let $\bar{\theta} = Cy$ be another linear estimator of θ, with

$$C = (X^\top X)^{-1} X^\top + D,$$

where $D \in \mathbb{R}^{m \times n}$ is a non-zero matrix. As we are restricting to unbiased estimators, the minimum mean squared error (MSE) implies minimum variance. The goal is therefore to show that such an estimator has a variance no smaller than that of $\hat{\theta}$, the OLS estimator.

At first, let us calculate the expected value of $\bar{\theta}$:

$$E[\bar{\theta}] = E[Cy] = E[((X^\top X)^{-1}X^\top + D)(X\theta + \epsilon)].$$

Expanding this expression by Lemma (2.1.18)

$$E[\bar{\theta}] = ((X^\top X)^{-1}X^\top + D)X\theta + ((X^\top X)^{-1}X^\top + D)E[\epsilon].$$

Since $E[\epsilon] = 0$, this simplifies to

$$E[\bar{\theta}] = ((X^\top X)^{-1}X^\top + D)X\theta$$

and

$$E[\bar{\theta}] = (X^\top X)^{-1}X^\top X\theta + DX\theta = (I + DX)\theta.$$

For the linear estimator $\bar{\theta}$ to be unbiased, we must have

$$E[\bar{\theta}] = \theta,$$

which holds if and only if $DX = 0$.

Now, consider the variance of $\bar{\theta}$

$$\mathrm{Var}(\bar{\theta}) = \mathrm{Var}(Cy) = C \cdot \mathrm{Var}(y) \cdot C^\top. \tag{2.1.18}$$

Since $\mathrm{Var}(y) = \sigma^2 I$, we have

$$\mathrm{Var}(\bar{\theta}) = \sigma^2 CC^\top$$

and

$$\mathrm{Var}(\bar{\theta}) = \sigma^2 \cdot \left((X^\top X)^{-1}X^\top + D\right)\left(X(X^\top X)^{-1} + D^\top\right).$$

Expanding this expression

$$\mathrm{Var}(\bar{\theta}) = \sigma^2 \cdot \left((X^\top X)^{-1}X^\top X(X^\top X)^{-1} + (X^\top X)^{-1}X^\top D^\top + DX(X^\top X)^{-1} + DD^\top\right).$$

Since $DX = 0$ we have,

$$\mathrm{Var}(\bar{\theta}) = \sigma^2 \cdot (X^\top X)^{-1} + \sigma^2 \cdot DD^\top$$

and

$$\mathrm{Var}(\bar{\theta}) = \mathrm{Var}(\hat{\theta}) + \sigma^2 \cdot DD^\top,$$

where $\mathrm{Var}(\hat{\theta})$ is the variance of the OLS estimator. To verify this, we compute $\mathrm{Var}(\hat{\theta})$

$$\mathrm{Var}(\hat{\theta}) = \mathrm{Var}\left((X^\top X)^{-1}X^\top y\right) = \mathrm{Var}\left((X^\top X)^{-1}X^\top (X\theta + \epsilon)\right).$$

Thus

$$\mathrm{Var}(\hat{\theta}) = (X^\top X)^{-1}X^\top \mathrm{Var}(\epsilon)X(X^\top X)^{-1} = \sigma^2(X^\top X)^{-1}.$$

Therefore, the variance of $\bar{\theta}$ is

$$\mathrm{Var}(\bar{\theta}) = \mathrm{Var}(\hat{\theta}) + \sigma^2 \cdot DD^\top,$$

where the second term $\sigma^2 \cdot DD^\top$ is always positive semi-definite matrix and this implies that $\bar{\theta}$ has a variance no smaller than $\hat{\theta}$.

We used the fact during calculating $\mathrm{Var}(\hat{\theta})$ and $\mathrm{Var}(y)$ that the ϵ_i's are uncorrelated and have a common variance, that is,

$$\mathrm{Var}[\epsilon] = \sigma^2 I.$$ □

Some first uses of `Python`. Now, for illustration of some definitions given, we use `Python` library `NumPy` in the code below to generate two random 150 element datasets (random variables) X_1 and X_2. Then, we can calculate the covariance between X_1 and X_2 directly using the formula 2.1.3.

```
import math

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn
# Generate 150 random values for X1, scaled between 0 and 4
X1 = 4 * np.random.rand(150, 1)
# Generate 150 random values for X2, scaled between 0 and 4
X2 = 4 * np.random.rand(150, 1)
# calculate covariance
Cov_X1X2 = np.mean(X1 * X2) - np.mean(X1) * np.mean(X2)
```

The variances of X_1 and X_2 can be calculated by 2.1.1 as

```
# Calculate the variance of X1:
Var_X1 = np.mean(X1**2) - np.mean(X1) ** 2
# Calculate the variance of X2:
Var_X2 = np.mean(X2**2) - np.mean(X2) ** 2
```

The correlation coefficient is then computed by 2.1.11 as

```
# Calculate the correlation between X1 and X2:
corr = Cov_X1X2 / (math.sqrt(Var_X1) * math.sqrt(Var_X2))
>>>correlation coefficient:
-0.06039366070854374
```

We observe that the linear relationship between X_1 and X_2 is absent, as indicated by the correlation coefficient near to 0.

Alternately, we can calculate the correlation using the `corrcoef` function. This returns the correlation matrix, which provides the correlation coefficients between the rows of the input matrix

```
# Generate a new random variable y as a linear combination of X1 and X2 with added
    noise
y = 4 * X1 + 2 * X2 - 6 + np.random.randn(150, 1)
# Combine X1, X2, and y into a single matrix X with columns representing the variables
X = np.c_[X1, X2, y]
corr_np = np.corrcoef(X.T)
```

Here, `X.T` is a 2-D array containing multiple variables and observations. Each row of `X.T` represents a variable, and each column represents a single observation for all those variables. The resulting correlation matrix is

```
>>>Correlation matrix:
[[ 1.         -0.06039366  0.84589617]
 [-0.06039366  1.          0.44813406]
 [ 0.84589617  0.44813406  1.        ]]
```

We can also use the `pandas` library for data manipulation and the `seaborn` library for better visualization. For more information, please refer to the respective documentations [5, 19] and [6].

```
# Create a pandas DataFrame from the matrix X
df = pd.DataFrame(X, columns=['X1', 'X2', 'y'])
# Visualize the correlation matrix using a heatmap from Seaborn
seaborn.heatmap(corr_pd, annot=True)
plt.show()
```

The resulting visualization of the correlation matrix is given in Figure 2.1.

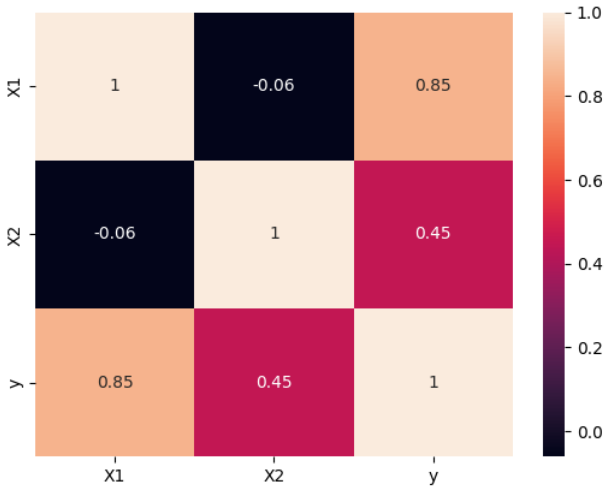


Figure 2.1: Color coded Correlation Matrix with a heat map from `seaborn.heatmap`.

2.2 Regression

2.2.1 Introduction to Regression

This section introduces the concepts and definitions of linear, multi-linear, and ridge regression. Regression is a statistical technique used to analyze the relationship between a dependent variable and one or more independent variables, with the goal of estimating this relationship. We will begin with the relatively simple case of linear regression.

Linear regression has been a cornerstone of statistical analysis for many years, and it remains a fundamental topic in numerous textbooks and research papers. While more advanced statistical techniques have emerged, linear and non-linear regression models are still widely used due to their efficiency in fitting data and their high level of interpretability. Despite the introduction of more complex models, the simplicity and transparency of regression analysis make it an indispensable tool in various fields.

This methodology is widely used in business, the social sciences, and many other fields. A few examples of applications are

1. Sales Forecasting: Predicts sales based on advertising spend, seasonality, or economic indicators.
2. Psychology: Used to explore relationships between psychological variables (like stress levels) and behaviors or outcomes (like academic performance).
3. Property Value Estimation: Used to estimate property prices based on factors like location, square footage, and amenities.

Having a solid understanding of linear regression is an excellent starting point for those interested in statistical analysis, as it provides the foundational knowledge required to explore more complex methods.

2.2.2 Intuition behind Regression

As an illustrative example, assume we have some limited data about mice weight and length and say we want to be able to predict the length of mice given a weight. Let x represent the weight and y denote the length of mice. The goal is to find the line that describes the relationship between x and y with minimal error, allowing us to make the stated prediction.

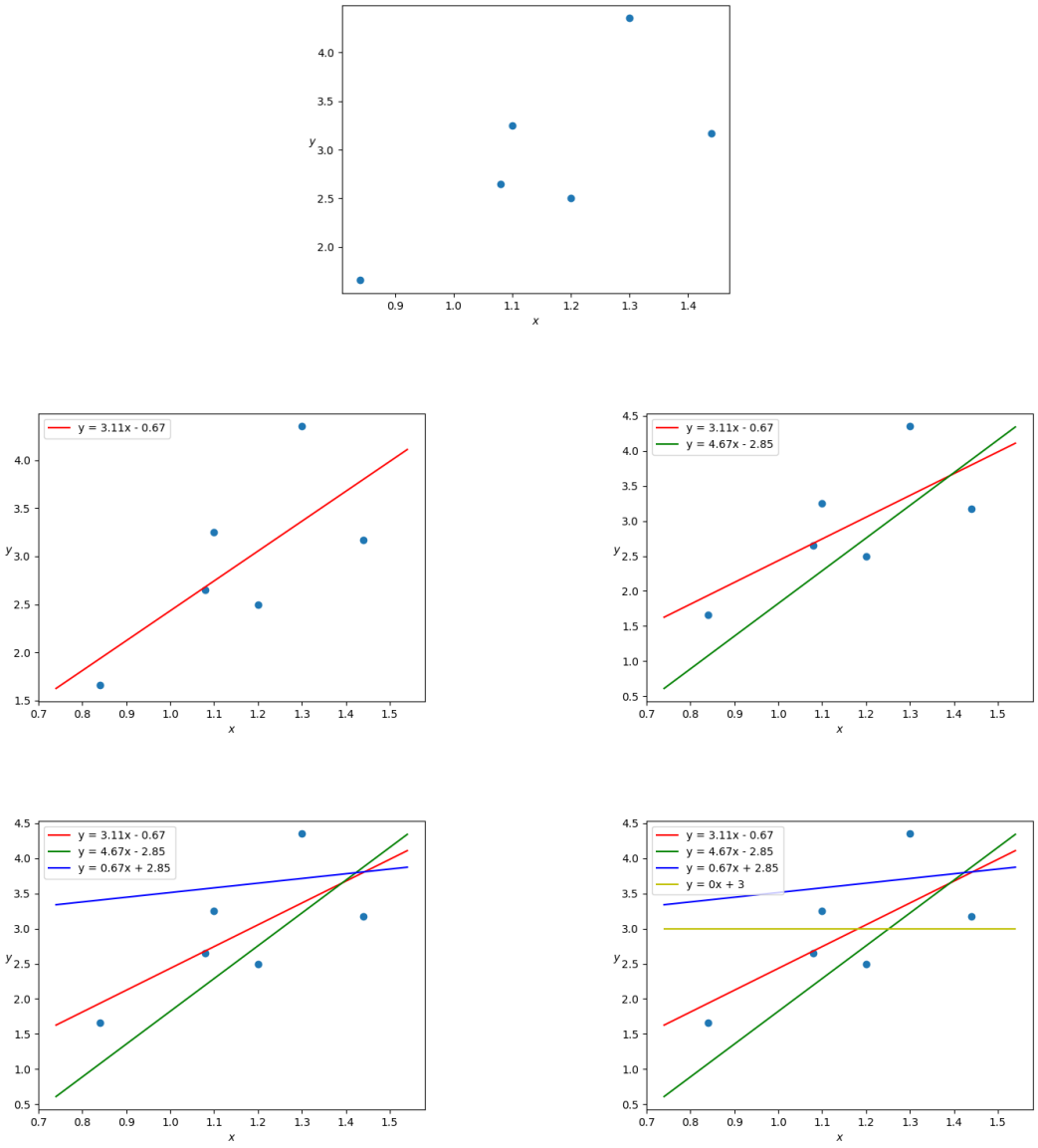


Figure 2.2: Regression lines for data $(x_0, y_0) = (0.84, 1.66)$, $(x_1, y_1) = (1.08, 2.65)$, $(x_2, y_2) = (1.1, 3.25)$, $(x_3, y_3) = (1.2, 2.5)$, $(x_4, y_4) = (1.3, 4.35)$ and $(x_5, y_5) = (1.44, 3.17)$.

While there are infinitely many lines, we want to choose the one that minimizes the distance between the observed data and predicted points on the line. Linear regression targets minimizing the total squared distance between each observed value and its corresponding predicted value on the line. In favorable cases, by Gauss-Markov Theorem 2.1.19, this results in a line that is the *best* linear representation of the trend in the data.

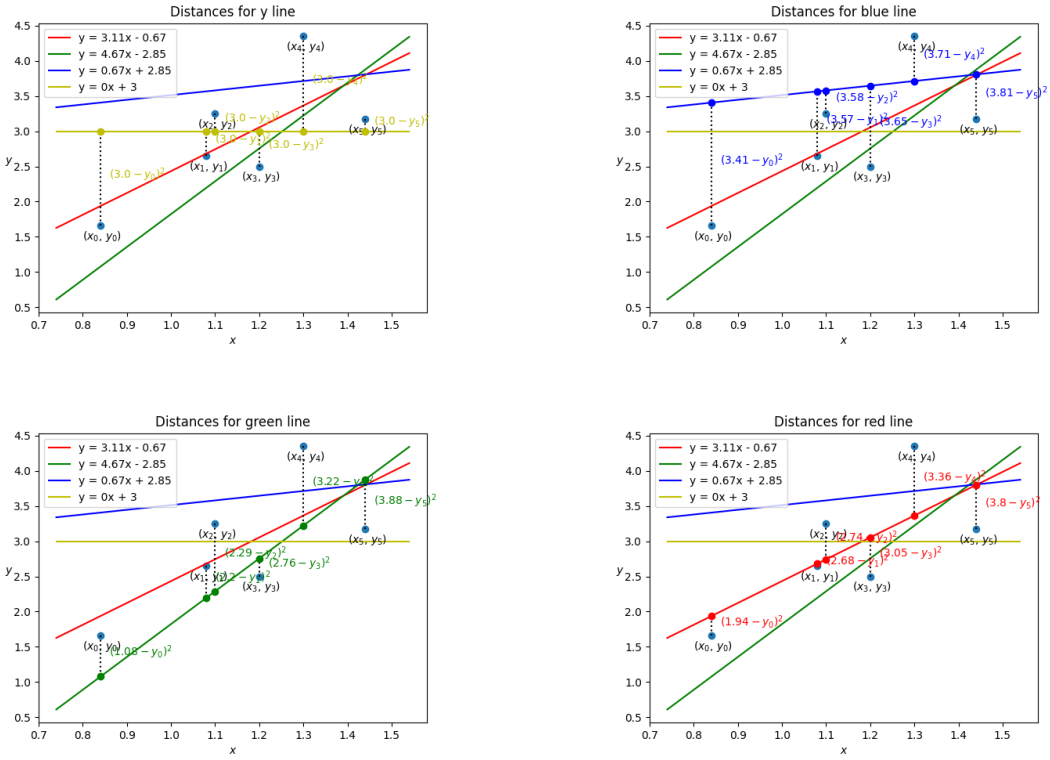


Figure 2.3: Distance Between observed data and predicted values

If we sum up squared distances we get different results for red, green, blue, and yellow lines, respectively.

$$\begin{aligned}(1.94-y_0)^2+(2.68-y_1)^2+(2.74-y_2)^2+(3.05-y_3)^2+\\ (3.36-y_4)^2+(3.8-y_5)^2=2.00783321;\\ (1.08-y_0)^2+(2.2-y_1)^2+(2.29-y_2)^2+(2.76-y_3)^2+\\ (3.22-y_4)^2+(3.88-y_5)^2=3.30819074;\\ (3.41-y_0)^2+(3.57-y_1)^2+(3.58-y_2)^2+(3.65-y_3)^2+\\ (3.71-y_4)^2+(3.81-y_5)^2=6.12847878;\\ (3.0-y_0)^2+(3.0-y_1)^2+(3.0-y_2)^2+(3.0-y_3)^2+\\ (3.0-y_4)^2+(3.0-y_5)^2=4.082.\end{aligned}$$

As we can see, the first, number corresponding to the red line is the smallest. In fact, if we do compute the linear regression on our data we find that the line corresponding to the distance is the best fitting line for our points, with the equation $y \approx 3.1 \cdot x - 0.67$. This regression line helps us to make predictions about mouse length. For example, if we observe a new data point for a mouse weighing 1.0 kg, we can estimate its length by substituting $x = 1$ into $y = 3.11 \cdot x - 0.67$ and get 2.44 cm.

2.2.3 Linear Regression

Simple linear regression is a somewhat straightforward approach for forecasting quantitative response y based on a single predictor variable x. This method assumes that there is a relatively linear relationship between y and x. Mathematically, we can write this linear relationship as follows

$$\hat{y}_i = \theta_0 + \theta_1 x_i + \epsilon_i, \tag{2.2.1}$$

where $\hat{y}_i$ is the value of the predicted variable in the ith trial; θ_0 and θ_1 are parameters; x_i is known constant, the value of predictor variable in the ith trial; ϵ_i is a random error term with mean $E[\epsilon_i] = 0$ and variance $\mathrm{Var}(\epsilon_i) = \sigma^2$; ϵ_i and ϵ_j are uncorrelated as that their covariance is zero

$$\mathrm{Cov}(\epsilon_i, \epsilon_j) = 0, \quad \forall i, j \text{ with } i \neq j, \quad i, j = 1, \ldots, n.$$

Regression model 2.2.1 is said to be simple or linear because there is only one predictor variable and no parameters appear as an exponent or are multiplied or divided by another parameter [10].

This can be written in a much more concise form using a vectorized form

$$\hat{y}_i = \theta \cdot x, \tag{2.2.2}$$

where θ is the model's parameter vector, containing the bias term θ_0 and the feature weight θ_1; x is the instance feature vector, containing x_0 and x_i where x_0 is always equal to 1; $h_\theta(x) = \hat{y}$ is the vector of predicted values containing $\hat{y}_1$ to $\hat{y}_n$; h_θ is the hypothesis function, using the model parameters θ.

As described in section 2.1 the most common performance metric to minimize is Root Mean Square Error (RMSE) is

$$\mathrm{RMSE}(X, h_\theta) = \frac{1}{n} \sum_{i=1}^{n} (\theta^T x^{(i)} - y^i)^2. \tag{2.2.3}$$

Mathematically this solves a problem

$$\min_\theta ||X\theta - y||_2^2,$$

where X is a matrix in $\mathbb{R}^{n \times 2}$

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}$$

To find the value of θ that minimizes RMSE (we sometimes refer to this as a *cost function*), we use the existence of a closed-form solution, so-called, *Normal Equation*

$$\hat{\theta} = (X^T X)^{-1} X^T y \tag{2.2.4}$$

where y is the vector of target values containing y^1 to y^n [8] and $\hat{\theta}$ denotes value of θ that minimizes the cost function.

We discuss how to practically find this minimizer in the next subsection.

Linear Regression in Python

We implement the Linear regression in Python. To make the logic behind this method clear, let us first start the implementation from scratch. This includes computing the normal equation and minimizing the cost function. As an example, we generate a random dataset as random variable X, and let $y = 4X - 6 + \epsilon$, where ϵ denotes an added error (often referred to as *a Gaussian blur*).

```
import numpy as np
import matplotlib.pyplot as plt
# Set the random seed for reproducibility
np.random.seed(0)
# Generate 150 random values for X, scaled between 0 and 4
X = 4 * np.random.rand(150, 1)
# Compute y as a linear combination of X with added noise
y = 4 * X - 6 + np.random.randn(150, 1)
```

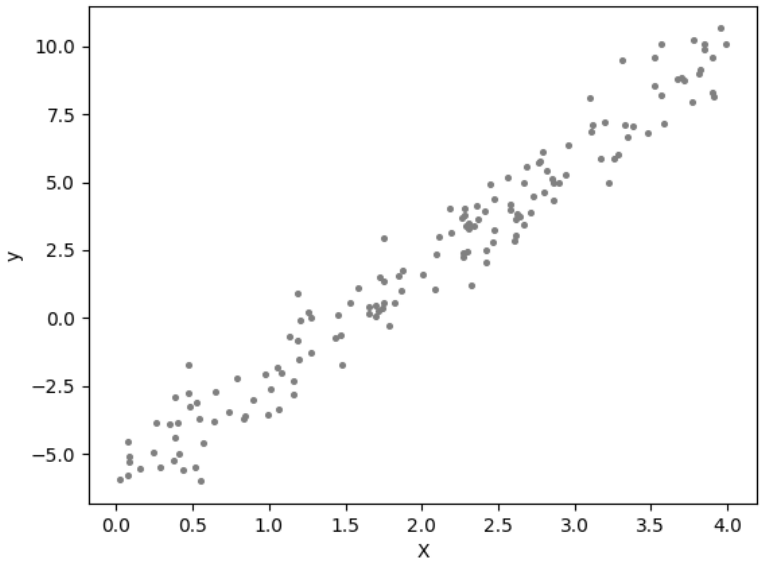


Figure 2.4: Random generated data

Now we compute $\hat{\theta}$ using the Normal Equation 2.2.4.

```
# Add a column of ones to X to account for the intercept term in linear regression
X_b = np.c_[np.ones((150, 1)), X]
# Compute the best-fitting parameters (theta) using the normal equation
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

The function that we used to generate data is $y = 4x_1 - 6$ plus the Gaussian Blur. We find

```
>>>theta_best
array([[-6.01006531]
     [ 3.96054788]])
```

In the best case scenario, we would have $\theta_0 = -6$ and $\theta_1 = 4$. However, taking error into consideration, we get $\theta_0 \approx -6.01$ and $\theta_1 \approx 3.96$. We predict new values

```
# Create a new input array X_new with values 0 and 4 for prediction
X_new = np.array([[0], [4]])
# Add a column of ones to X_new for the intercept term
X_new_b = np.c_[np.ones((2, 1)), X_new]
# Use the learned parameters (theta_best) to predict y values for X_new
y_predict = X_new_b.dot(theta_best)
>>>y_pred
array([[-6.01006531]
     [ 9.8321262 ]])
```

Using the equation of a line on two points we can now draw a regression line between our data points and get an estimation, which looks like Figure 2.5.

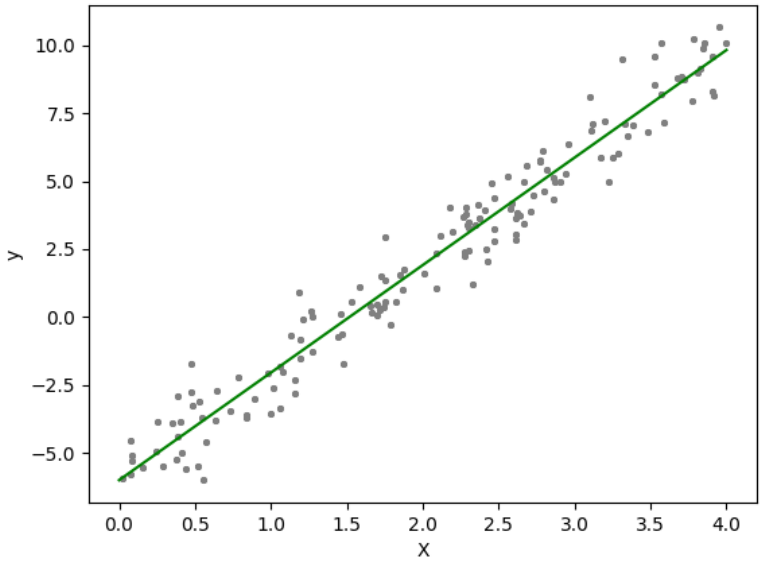


Figure 2.5: Linear Regression Model Predictions

We can also use `Python` package called `scikit-learn` to do linear regression. From an implementation standpoint, `LinearRegression` class is part of `scipy.linalg.lstsq()` which we can call directly using `numpy`.

```
theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>>theta_best_svd
array([[-6.01006531]
     [ 3.96054788]])
```

`np.linalg.lstsq()` computes $\hat{\theta} = X^{\dagger}y$, where $X^{\dagger}$ is the *pseudoinverse* of X, we could also use `np.linalg.pinv()` to compute in a more straightforward fashion.

```
>>>np.linalg.pinv(X_b).dot(y)
array([[-6.01006531],
       [ 3.96054788]])
```

Now we take a closer look at `scikit-learn`'s `LinearRegression` class. This has two main methods `fit(X,y)`, which fits a linear model to the given data, and `predict(X)`, which predicts using linear estimation. There is also the command `score(X, y, sample_weight=None)`,

that returns coefficient of determination R^2, defined as $(1 - \frac{u}{v})$ where u is the residual sum of squares `((y_true - y_pred) **2).sum()` and v is the total sum of squares `((y_true - y_true.mean())**2 ).sum()`. The best possible score is 1.0. This score can also be negative since the model can be arbitrarily bad. A constant model that always predicts the expected value of y, disregarding the input features, would get a score of 0.0. [16].

```
lin_reg = LinearRegression()
lin_reg.fit(X, y)
>>>lin_reg.intercept_, lin_reg.coef_
(array([-6.01006531]), array([[3.96054788]]))

 >>>lin_reg.predict(X_new)
array([[-6.01006531],
       [ 9.8321262 ]])
>>>lin_reg.score(X,y)
0.9587180103089048
```

After creating `LinearRegression` instance we can input our data into it. This returns the intercept and coefficient of the linear model. We can also check the performance metric of our model with `score()` method.

2.2.4 Multi-linear Regression

In the previous section, we explored simple linear regression, where the relationship between a response variable y and a single predictor variable x was modeled as a linear equation. However, many real-world problems require the consideration of multiple factors or predictors to accurately model the relationship with the response variable. Mathematically, we can express the linear relationship between y and $X_1, X_2, \ldots, X_n$ as follows.

$$\hat{y}_i = \theta_0 + \theta_1 x_i^1 + \theta_2 x_i^2 + \cdots + \theta_n x_i^n + \epsilon_i \tag{2.2.5}$$

where $\hat{y}_i$ is the value of the predicted variable in the ith trial; $\theta_0, \theta_1, \ldots, \theta_n$ are parameters; x_i^j is known constant, the value of predictor variable X_j in the ith trial; ϵ_i is a random error term with mean $E[\epsilon_i] = 0$ and variance $\mathrm{Var}(\epsilon_i) = \sigma^2$; ϵ_i and ϵ_k are uncorrelated as their covariance is zero. Here $i, k = 1, 2, 3, \ldots, m$ and $j = 1, 2, 3, \ldots, n$.

As described in section 2.1, the most common performance metric to minimize is Root Mean Square Error (RMSE), which for the linear regression case was given by equation (2.2.3). In multi-linear regression, we extend this concept to account for multiple predictor variables.

The problem is formulated as

$$\min_{\theta} \|X\theta - y\|_2^2$$

where X is a matrix in $\mathbb{R}^{m \times (n+1)}$

$$X = \begin{bmatrix} 1 & x_1^1 & x_1^2 & \ldots & x_1^n \\ 1 & x_2^1 & x_2^2 & \ldots & x_2^n \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ 1 & x_m^1 & x_m^2 & \ldots & x_m^n \end{bmatrix}$$

The next theorem helps us to find the minimizer of the given optimization problem.

Theorem 2.2.1. *Let $f : U \to \mathbb{R}$, where U is some open subset of $\mathbb{R}^n$, be a continuously differentiable strictly convex function. Let $\bar{x} \in U$ be a point such that*

$$\nabla f(\bar{x}) = 0$$

Then $\bar{x}$ is the unique global minimizer.

The reader can find the proof in [13]. The function given above is a strictly convex, continuously differentiable function, since it is a second-norm function. The gradient of the function is $2X^TX\theta - 2X^Ty$. The only thing we need to do to find the solution is solve the system of equations $2X^TX\theta - 2X^Ty = 0$ with respect to θ. Since X is a full-rank matrix, X^TX is invertible, which implies that the solution to the above system of linear equations is

$$\hat{\theta} = (X^TX)^{-1}X^Ty,$$

where y is the vector of target values containing y^1 to y^m.

Multi-linear Regression in Python

We will now extend our implementation to multi-linear regression, where the dependent variable y is influenced by multiple independent variables. As before, we introduce random vectors.

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
X1 = 4 * np.random.rand(150, 1)
X2 = 4 * np.random.rand(150, 1)
y = 4 * X1 + 2 * X2 - 6 + np.random.randn(150, 1)
```

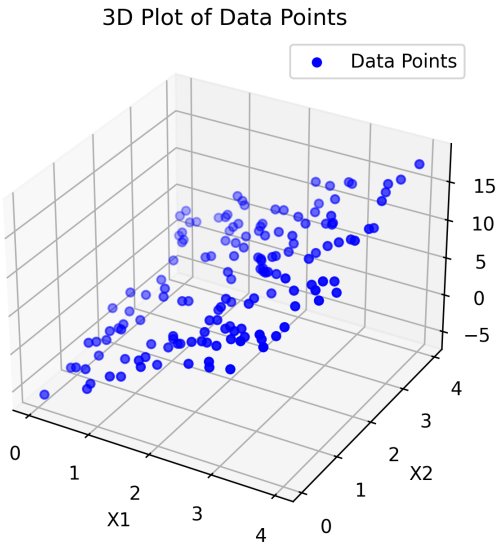


Figure 2.6: Random generated data

Similar to simple linear regression, we will compute the parameters $\hat{\theta}$ using the Normal Equation 2.2.4. To start, we modify the design matrix to include two independent variables ($X1$ and $X2$)

```
X_b = np.c_[np.ones((150, 1)), X1, X2]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Using the Normal Equation, we compute the parameters $\hat{\theta}$

```
>>>Theta best:
array([[-6.07869782]
     [ 3.89710346]
     [ 2.07909339]])
```

Here, the parameters indicate:

$$\theta_0 \approx -6.08 \quad \text{(The intercept)}.$$

$$\theta_1 \approx 3.90 \quad \text{(The coefficient for } X_1\text{)}.$$

$$\theta_2 \approx 2.08 \quad \text{(The coefficient for } X_2\text{)}.$$

These values approximate the relationship between y, X_1, and X_2, considering the random error added to the data. We can now predict new values for y given specific values of X_1 and X_2. Let us compute predictions for three points: $(X_1 = 1, X_2 = 0)$, $(X_1 = 0, X_2 = 0)$, and $(X_1 = 0, X_2 = 1)$.

```
X_new = np.array([[1, 0], [0, 0], [0, 1]])
X_new_b = np.c_[np.ones((X_new.shape[0], 1)), X_new]
y_predict = X_new_b.dot(theta_best)
>>>Predicted values:
array([[-2.18159436]
     [-6.07869782]
     [-3.99960443]])
```

Using the equation of a plane on three points, we can now draw a regression plane between our data points and get an estimation if Figure 2.7

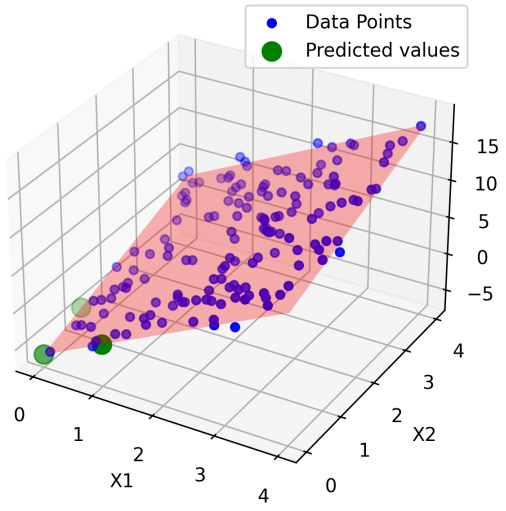


Figure 2.7: Multi-linear Regression Model Predictions

We can also use the `scikit-learn` library to compute the parameters of linear regression directly. We have already discussed the mathematical background of multi-linear regression, and using the library, we can calculate the parameters with the following code

```
reg = LinearRegression().fit(X, y)
intercept = reg.intercept_
coefficients = reg.coef_.flatten()
>>>Intercept:
array([-6.07869782])
>>>Coefficients:
array([3.89710346, 2.07909339])
```

2.2.5 Ridge Regression

Ridge regression, also known as Tikhonov regularization, is a technique used in linear regression to address the problem of multicollinearity among predictor variables. Multicollinearity occurs when independent variables in a regression model are highly correlated, which can lead to unreliable and unstable estimates of regression coefficients. Ridge regression shrinks the regression coefficients by imposing a penalty on their size. The ridge coefficients minimize a penalized residual sum of squares

$$\hat{\theta}_{\text{ridge}} = \arg\min_{\theta} \left[\sum_{i=1}^{m} \left(y_i - \theta_0 - \sum_{j=1}^{n} x_{ij}\theta_j \right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2 \right]. \tag{2.2.6}$$

Here $\lambda \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of λ, the greater the amount of shrinkage. The coefficients are shrunk toward zero (and each other).

An equivalent way to write the ridge problem is as follows, using a constraint on the coefficients

$$\hat{\theta}_{\text{ridge}} = \arg\min_{\theta} \left[\sum_{i=1}^{m} \left(y_i - \theta_0 - \sum_{j=1}^{n} x_{ij}\theta_j \right)^2 \right], \quad \text{subject to} \quad \sum_{j=1}^{n} \theta_j^2 \leq t, \tag{2.2.7}$$

where there is a one-to-one correspondence between the parameters λ in the first formulation and t in the second formulation.

When there are many correlated variables in a linear regression model, their coefficients can become poorly determined and exhibit high variance. A wildly large positive coefficient on one variable can be canceled by a similarly large negative coefficient on its correlated cousin. By imposing a size constraint on the coefficients, as in the second formulation, this problem is alleviated.

The ridge solutions are not equivariant under scaling of the inputs, and so one normally standardizes the inputs before solving the first formulation. This means transforming each column of the input matrix X to have zero mean and unit variance. In addition, notice that the intercept θ_0 has been left out of the penalty term. Penalization of the intercept would make the procedure depend on the origin chosen for y; that is, adding a constant c to each of the targets y_i would not simply result in a shift of the predictions by the same amount c. The solution to the first formulation 2.2.6 can be separated into two parts, after reparametrization using centered inputs, each x_{ij} gets replaced by $x_{ij} - \bar{X}_j$, where $\bar{X}_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$.

We estimate θ_0 by $\bar{y} = 1/m \sum_{i=1}^{m} y_i$. The remaining coefficients get estimated by a ridge regression without intercept, using the centered x_{ij}. Henceforth, we assume that this centering has been done, so that the input matrix X has n (rather than $n+1$) columns.

We write the criterion in equation 2.2.6 in matrix form as

$$\text{RSS}(\lambda) = \|y - X\theta\|_2^2 + \lambda\|\theta\|_2^2, \tag{2.2.8}$$

The ridge regression solutions are easily seen to be

$$\hat{\theta}_{\text{ridge}} = (X^T X + \lambda I)^{-1} X^T y, \tag{2.2.9}$$

where I is the $n \times n$ identity matrix. Notice that with the choice of quadratic penalty on θ, the ridge regression solution is again a linear function of y. The solution adds a positive constant λ to the diagonal of $X^T X$ before inversion. This makes the problem non-singular, even if $X^T X$ is not of full rank, and was the main motivation for ridge regression when it was first introduced in statistics [9].

Ridge regression in Python

We start by generating sample data with X_1 and X_2 highly correlated.

```
import numpy as np
from matplotlib import pyplot as plt

X1 = 4 * np.random.rand(150, 1)
X2 = X1 + 0.1 * np.random.rand(150, 1)
y = 4 * X1 + 4 * X2 - 6 + np.random.randn(150, 1)
```

We can also calculate correlation coefficient

```
correlation_coefficient = np.corrcoef(X1.flatten(), X2.flatten())[0, 1]
>>>Correlation coefficient:
0.9999964212341061
```

Linear regression is not appropriate in this case because the regression parameters are unstable. The instability arises as the matrix $X^T X$ is close to singular. Here are the parameters estimated using linear regression

```
>>>Theta best (Linear Regression):
array([[ -6.07869782]
     [-27.74025083]
     [35.63735429]])
```

Using ridge regression, we can achieve a more stable estimation. First, we combine X_1 and X_2 into a single matrix, standardize it (to have zero mean and unit variance), and compute the ridge regression parameters

```
X_combined = np.c_[X1, X2]
X_standardized = np.c_[X1 - np.mean(X1), X2 - np.mean(X2)]
alpha = 1  # Regularization parameter
I = np.eye(X_standardized.shape[1])  # Identity matrix for regularization

# Ridge regression closed-form solution
theta = np.linalg.inv(X_standardized.T.dot(X_standardized) + alpha * I).dot(
    X_standardized.T).dot(y)

# Compute predictions and intercept
y_pred = X_combined.dot(theta)
intercept = np.mean(y - y_pred)
```

The resulting parameters are:

```
>>>Intercept(Ridge regression):
array([-6.179273891863882])
>>>Coefficients(Ridge regression):
array([4.03482177 4.01439142])
```

The results can be visualized with an appropriate plot in Fig 2.8.

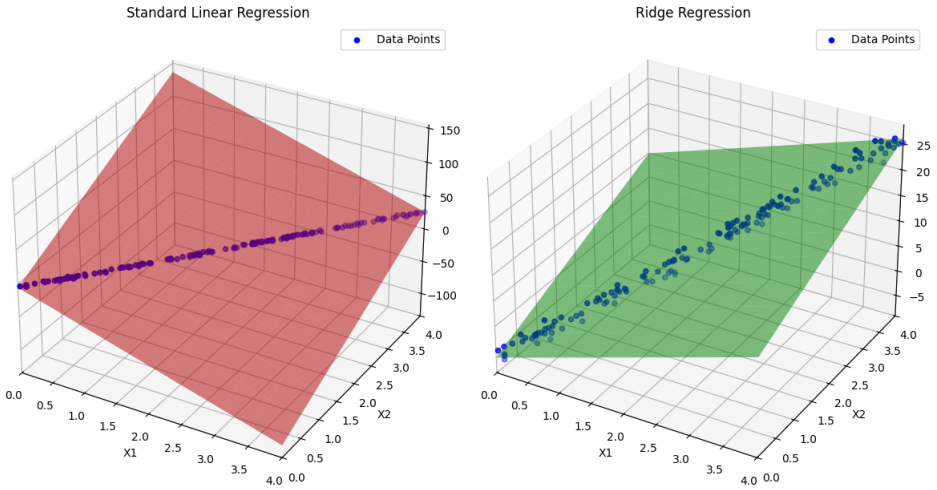


Figure 2.8: Ridge Regression vs linear regression

We can also use the `scikit-learn` library.

```
from sklearn.linear_model import Ridge

ridge_reg = Ridge(alpha=1, fit_intercept=True, solver="cholesky")
ridge_reg.fit(X_combined, y)
```

The parameter α controls the regularization strength. When $\alpha = 0$, the objective is equivalent to ordinary Least Squares Regression. However, using $\alpha = 0$ with the `Ridge` object is not recommended; we use `LinearRegression` instead. The parameter `fit_intercept` determines whether to fit an intercept. If set to `False`, the intercept is assumed to be 0, and the data should be centered beforehand. Here, the `solver` parameter specifies the computational routine. The `'cholesky'` solver uses a closed-form solution via Cholesky decomposition. We do not discuss these here and direct the reader towards the documentation [17]. The calculated parameters using the given code are as follows:

```
>>>Intercept(Ridge regression sklearn):
array([-6.179273891863816])
>>>Coefficients(Ridge regression sklearn):
array([4.03482177 4.01439142])
```

For different datasets, an appropriate value for α must be chosen. The choice of α depends on how strong the regularization needs to be in order to achieve a better approximation [17].

Chapter 3

From theory to practice: working with the real dataset

In Chapter 2, we explored the theoretical foundations and concepts of elementary statistics and linear regression models. In Chapter 3, we aim to provide a real-life implementation of these theories and connect the concepts to practical applications. This chapter focuses on preparing the data through pre-processing techniques and utilizing regression models to analyze the hidden patterns within it. By doing so, we will demonstrate how these ideas can be brought to life and applied to real-world data to extract meaningful insights.

3.1 Structure of Data

Data at our disposal relates to schools in Georgia. We have information about the number of pupils, infrastructure, budget, sports and activity areas, etc. The Educational and Scientific Infrastructure Development Agency together with the Ministry of Education, Science, Culture, and Sport of Georgia provided the data. Initially, there were five different Excel tables:

- `student number.xlsx`
- `school infrastructure.xlsx`
- `school infrastructure condition.xlsx`
- `school revenues.xlsx`
- `school buildings and areas.xlsx`

To grasp the size of the data used, let us analyze the structure and shape of these files. We can see the number of columns across different tables in Figure 3.1, as well as an outline of separate columns in Figure 3.2 and Figure 3.3

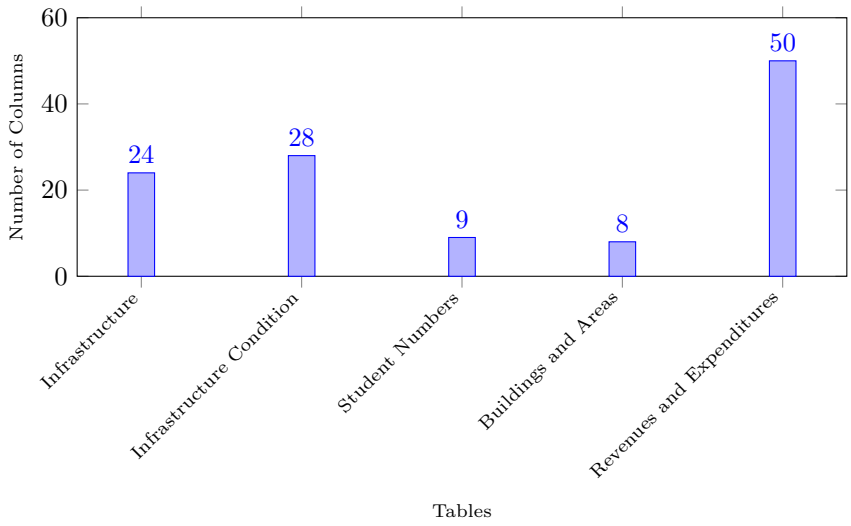


Figure 3.1: Number of Columns in Each Dataset

Table 1: School Infrastructure. This table consists of 2,557 entries with 24 columns, summarizing details about school infrastructure, including building and yard areas, the number of floors and classrooms, the type of heating, and the overall state of facilities like sports halls and cafeterias.

Table 2: School Infrastructure Condition. This table focuses on the physical condition of school buildings and facilities, such as the condition of roofs, facades, windows, heating systems, bathrooms, and various rooms like libraries and laboratories. It contains information about 2,557 schools with 28 columns.

Table 3: Student Numbers. This table includes information about schools, their region, district, name, and the total number of students enrolled in each school. It contains data from 2,078 schools with 9 columns.

Table 4: School Buildings and Areas. This table describes the types of school buildings, their IDs, additional information, and quantities. It provides an overview of the number and types of buildings associated with schools. Due to its inappropriate structure, this table has 13,375 rows and 8 columns.

Table 5: School Revenues and Expenditures. This table details the financial information of schools, including annual budgets, revenues (government funding, local funding, own income), and expenditures (salaries, office costs, utility expenses, and maintenance costs). It contains 2,084 rows and 50 columns.

Figure 3.2: Datasets columns

After cleaning, removing, and reconstructing, we end up with one consolidated table in two formats: one including different buildings across schools and one excluding them.

Table including different buildings. This table contains 1,835 rows and 83 columns, resulting in a total of 152,305 entries.

Table excluding different buildings This smaller table has 1,513 rows and 82 columns, bringing a total of 124,066 entries.

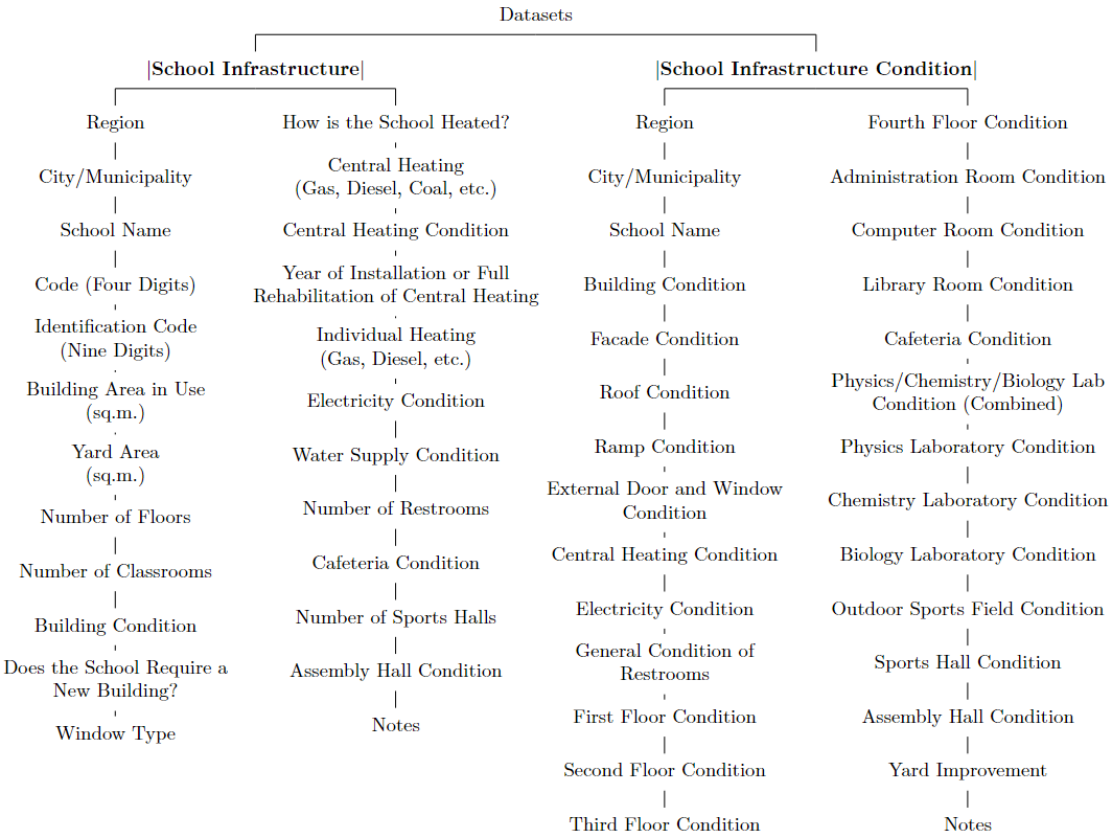


Figure 3.3: Dataset columns cont.

3.2 Preparing Data: Pre-Processing and Cleaning

In the first section, we will pre-process the given tables and clean the data. Pre-processing begins by restructuring each table, removing unnecessary columns, and renaming existing ones if needed. The next step is to join the tables to produce one large, concatenated table. To ensure consistency and usability, the following pre-processing steps were applied to the datasets.

3.2.1 Resturcturing the Tables

We will start by modifying two tables: `school infrastructure condition.xlsx` and `school infrastructure.xlsx`. We carry out two steps:

- Remove the prefixes 'ՆՆօՅ - ', 'ՆՆօՅ ', 'ՆՆօՅ-', 'ՆՆօՅ -', 'ՆՆօՅ', 'ՆՆօՅ- ' from the beginning of each school name.
- Add 'ՆՆօՅ - ' to ensure correct naming across all names and strip the names to remove any unnecessary spaces. This step will be applied to every table from now on as this fix makes sure prefixes are systematic across every dataset.

- Remove the `Notes` (’შენიშვნა’) column from the dataset.

```
# Define a list of prefixes to remove
prefixes = [’სსიპ - ’, ’სსიპ ’, ’სსიპ-’, ’სსიპ -’, ’სსიპ’, ’სსიპ- ’]

# Remove the prefixes from the beginning of each school name and strip
for prefix in prefixes:
    df[’სკოლის სახელწოდება’] = df[’სკოლის სახელწოდება’].str.
    replace(f’^{prefix}’, ’’, regex=True).str.strip()

# Add the correct prefix to every school name
df[’სკოლის სახელწოდება’] = ’სსიპ - ’ + df[’სკოლის სახელწოდება’]
# Drop column
df = df.drop(columns=’შენიშვნა’)
```

Next, we modify the `students number.xlsx` file. The following steps are applied:

- Remove the columns:
  - №, `Identification Code` (საიდენტიფიკაციო კოდი) and `School type` (სკოლის ტიპი).
- Rename columns:
  - `Name Of School` (სკოლის დასახელება) to `School Name` (სკოლის სახელწოდება);
  - `District` (რაიონი) to `City/Municipality` (ქალაქი/მუნიციპალიტეტი);
  - `Identification Code (1)` (საიდენტიფიკაციო კოდი.1) to `Code (Nine-Digit)` (კოდი (ცხრანიშნა));
  - `Code` to `Code (Four-Digit)` (კოდი (ოთხნიშნა));
  - `Total Number of Stud.` (სულ მოსწ. რაოდენობა) to `Number of Students` (მოსწავლეთა რაოდენობა).

  Doing this ensures consistent naming of columns across different tables, which simplifies the final step of merging them into a unified dataset.

```
# Drop unnecessary columns
df = df.drop(columns=[’№’, ’საიდენტიფიკაციო კოდი’, ’სკოლის ტიპი’])

# Rename columns to ensure consistency
df = df.rename(columns={
    ’სკოლის დასახელება’: ’სკოლის სახელწოდება’,
    ’რაიონი’: ’ქალაქი/მუნიციპალიტეტი’,
    ’საიდენტიფიკაციო კოდი.1’: ’კოდი (ცხრანიშნა)’,
    ’Code’: ’კოდი (ოთხნიშნა)’,
    ’სულ მოსწ. რაოდენობა ’: ’მოსწავლეთა რაოდენობა’
})
```

- Process the school names using the same method as described for `school infrastructure condition.xlsx` and `school infrastructure.xlsx`.

Next, we will process the final table: `school revenue.xlsx`:

- Remove the ’№’ column;
- Process the school names using the same method outlined above;

- Rename the columns for consistency.

Lastly, we have `school buildings and areas.xlsx`, which will not be taken into account as most of the data appears to be inaccurate, incomplete, or outdated. However, some columns from this dataset will still be utilized indirectly, as described later.

In the next step, we will concatenate the cleaned and restructured tables into a single dataset. This unified table will serve as the basis for further analysis, enabling us to apply statistical methods and models without discrepancies caused by inconsistent naming or formatting across the original datasets.

3.2.2 Preparing Unified Dataset

Joining `school infrastructure condition.xlsx` and `school infrastructure.xlsx`

As of now, we have four restructured tables that need to be joined together. In the beginning, we follow a systematic approach to merge `school infrastructure.xlsx` and `school infrastructure condition.xlsx`. Here are the key steps taken:

1. **Alliging indices**

   Both tables contain hierarchical information (region, municipality, and school name). Setting these as indices ensures alignment during the merge and eliminates inconsistencies in row ordering.

```
df1 = df1.set_index(['რეგიონი', 'ქალაქი/მუნიციპალიტეტი', 'სკოლის სახელწოდება'])
df2 = df2.set_index(['რეგიონი', 'ქალაქი/მუნიციპალიტეტი', 'სკოლის სახელწოდება'])
```

2. **Renaming Overlapping Columns**

   To preserve the original data from both tables, overlapping columns are suffixed with `.1`, preventing the loss of information during the merge. These columns will be reviewed, and only the usable values will be retained.

```
common_columns = df1.columns.intersection(df2.columns)
df2 = df2.rename(columns={col: f"{col}.1" for col in common_columns})
```

3. **Merging Datasets**

   Concatenation along columns combines all data into a single table, maintaining alignment by index. The index is reverted to `pandas`' default integer indexing

```
result = pd.concat([df1, df2], axis=1)
result.reset_index(inplace=True)
```

4. **Reordering Columns**

   After merging, the columns were reorganized to group related fields logically. This improves readability and usability for subsequent analysis.

```
new_order = [...]
result = result[new_order]
```

5. **Handling Redundant or Conflicting Data**

   Columns representing the same data were reconciled using `combine_first()`. After reuniting, redundant columns with `.1` suffixes were dropped.

```
for el in common_columns:
    result[el] = result[el].combine_first(result[f'{el}.1'])
result = result.drop(columns=[f'{col}.1' for col in common_columns])
```

With the help of the function below, we determine which columns or values had conflicting data and which ones should be retained. This ensured that only the necessary and accurate data was included in the final dataset.

```
def compare_columns(df, col1, col2):
    return np.where(
        (df[f'{col1}'] != df[f'{col2}']) &
        ~(pd.isna(df[f'{col1}']) & pd.isna(df[f'{col2}'])))
```

6. **Removing secondary buildings (optional)**

   This step is optional and performed as part of data preprocessing and experimentation. Some schools have multiple buildings, such as secondary or tertiary constructions, which we deemed unnecessary for the analysis. Our goal in this step was to keep only the main buildings.

   We observed that the names of most non-primary buildings ended with 'კორპუსი)', so we removed rows where the school name ended with ")". However, this approach did not fully address all cases, as there were anomalies and naming inconsistencies; for instance, some schools did not follow this pattern, and their names required manual review. To handle these anomalies, we examined schools with names that did not end with 'სკოლა' and handpicked specific rows to delete or rename to standardize the dataset.

   This process ensured that only the main buildings were included while unnecessary rows were removed, improving the quality and consistency of the data.

```
if withoutBuildings:
    maskDf1 = result[~result['სკოლის სახელწოდება'].str.endswith(')')]
    # used for manual observation
    # maskDf2 = maskDf1[~maskDf1['სკოლის სახელწოდება'].str.endswith('სკოლა')]
# List of rows to delete
rows_to_delete = [...]

# List of rows to rename
rows_to_rename = {...}
result = maskDf1[~maskDf1['სკოლის სახელწოდება'].isin(rows_to_delete)]
result.loc[:, 'სკოლის სახელწოდება'] =
    result['სკოლის სახელწოდება'].replace(rows_to_rename)
```

After this step, we have two versions of our dataset, one with different buildings and one without. We will continue operating on both of them.

As a result, we have joined `infrastructure.xlsx` and `school infrastructure condition.xlsx`. Now comes the most important part: joining the result above with `students number.xlsx`. Now, it happens that we are confident that the data in `students number.xlsx` is accurate for every entry. We will align every schools name and code based on this dataset and incorporate information about student numbers into our joined dataset.

Further merging with `students number.xlsx`

Tf-idf and cosine similarity. In order to properly merge these tables some extra techniques need to be discussed first. Term Frequency Inverse Document Frequency (TF-IDF) is a statistical measure used in text analysis to evaluate the importance of a term relative to a document and

the entire corpus. This is a common term weighting scheme in information retrieval, that has also found good use in document classification. The goal of using TF-IDF instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus [22].

Components of TF-IDF(see [20]):

Term Frequency (TF). Measures the frequency of a term t in document d

$$\mathrm{TF}(t, d) = \frac{\text{Frequency of } t \text{ in } d}{\text{Total terms in } d}.$$

Inverse Document Frequency (IDF). Measures the uniqueness of a term t across corpus D

$$\mathrm{IDF}(t, D) = \log\left(\frac{\text{Total number of documents in corpus}}{\text{Number of documents containing } t}\right).$$

TF-IDF Score. Product of TF and IDF, indicating the importance of a term t in the document d within a corpus D

$$\mathrm{TFIDF}(t, d, D) = \mathrm{TF}(t, d) \times \mathrm{IDF}(t, D).$$

In the context of our project, we use TF-IDF to convert Georgian school names into numerical representations. Each school name is treated as a "document" and each character n-gram (a sequence of n characters) is treated as a "term". This character-level analysis captures the similarities between school names that might have slight variations, such as typos or alternate spellings.

For example: 'სსიპ სკოლა 1' and 'სსიპ სკოლა №1' will have closely aligned TF-IDF vectors.

Cosine Similarity is the measure used to indicate how similar two non-zero vectors are in a multidimensional space. It is particularly useful in text analysis, as it quantifies the similarity between two documents irrespective of their magnitude. This property makes it ideal for comparing numerical representations like TF-IDF vectors, hence combining those two gives the best results.

Cosine similarity CS measures the cosine of the angle between two vectors in a vector space.

$$\mathrm{CS}(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

- A and B are the two vectors being compared.
- $A \cdot B$ is the dot product of the vectors.
- $\|A\|$ and $\|B\|$ are the magnitudes (or lengths) of A and B.

We use cosine similarity to compare the TF-IDF vectors of school names from two datasets (`infrastructure.xlsx` and `students number.xlsx`). This helps determine the most similar names between the two datasets and accurately merge them, even when there are minor variations in spelling or formatting.

For Example, high cosine similarity between 'სსიპ სკოლა 1' and 'სსიპ სკოლა №1' indicates that these names likely refer to the same school.

We use TF-IDF metric and cosine similarity concepts to join these two tables.

1. **Prepare school name preprocess function**

   This function will help to normalize school names across different tables, such as removing punctuations, whitespaces, English characters

```
def preprocess_name(name, remove_punctuation=True, remove_english=True,
    remove_whitespace=True,
            replace_n=True):
    if remove_punctuation:
        name = re.sub(r'[^\w\s]', '', name)  # Remove punctuation
    if remove_whitespace:
        name = re.sub(r'\s+', ' ', name).strip()  # Normalize whitespace
    if remove_english:
        name = re.sub(r'[a-zA-Z]', '', name)  # remove any english characters
    if replace_n:
        name = re.sub(r'\b[Nn](\d+)', r'№\1', name)
    return name
```

2. **Preprocess school names**

   Create a new column for cleaned names, so we do not modify real names too much

```
df1['სკოლის სახელწოდება_cleaned'] =
    df1['სკოლის სახელწოდება'].apply(lambda x: preprocess_name(x))
df2['სკოლის სახელწოდება_cleaned'] =
    df2['სკოლის სახელწოდება'].apply(lambda x: preprocess_name(x))
```

3. **Create TF-IDF matrix**

   In this step, each school name is treated as a document, and we compute the TF-IDF score for character-level n-grams (sequences of n characters) extracted from the names. This approach captures subtle variations and patterns in the names, enabling us to compare them effectively.

   The TF-IDF vectorizer is implemented using the `TfidfVectorizer` class from the `sklearn.feature_extraction.text` module. This class internally performs two intermediate steps: first, it converts the text collection into a `CountVectorizer` object, which generates a matrix of token counts. Then, it applies the `TfidfTransformer` to transform the count matrix into a normalized TF or TF-IDF representation [18].

```
vectorizer = TfidfVectorizer(analyzer='char', ngram_range=(1, 24))
all_names = pd.concat([df1['სკოლის სახელწოდება_cleaned'],
    df2['სკოლის სახელწოდება_cleaned']])
tfidf_matrix = vectorizer.fit_transform(all_names)
```

   `analyzer='char'`: Specifies that the TF-IDF computation should be based on character-level n-grams (sequences of characters), rather than word-level tokens. This is particularly useful for capturing small variations and patterns within text. `ngram_range=(1, 24)`: Defines the range of n-grams to consider. Here, n-grams of lengths between 1 and 24 characters are generated.

   For example:

   - A 1-gram for "school" would be ['s', 'c', 'h', 'o', 'o', 'l'].
   - A 3-gram would be ['sch', 'cho', 'hoo', 'ool'], and so on up to 24 characters.

   After creating the `TfidfVectorizer`, we merge the school names from both tables to create a unified corpus of all names. Next, we learn the vocabulary of n-grams from the input

data and compute the IDF scores for them. Finally, we apply the learned vocabulary to generate the TF-IDF matrix.

As a result, we obtain a sparse matrix where rows represent individual school names (documents) and columns correspond to the unique n-grams identified. The matrix contains the TF-IDF scores of each n-gram for each school name.

4. **Create similarity matrix from cosine similarities**

   We calculate the similarity between school names in two datasets (`df1` and `df2`) using their TF-IDF representations. The pairwise cosine similarity between the rows of the two matrices is computed. Cosine similarity measures the cosine of the angle between two vectors, providing a value between 0 and 1.

   We compare names in a unified dataset, where the first half of the TF-IDF matrix represents names from `df1` and the second half represents names from `df2`. In the end, this matrix contains pairwise cosine similarity scores between the school names from the two datasets. Rows correspond to school names in `df1`, and columns correspond to school names in `df2`. Each element `[i, j]` in the matrix represents the cosine similarity between the `i`th school name in `df1` and the `j`th school name in `df2`.

```
similarity_matrix = cosine_similarity(tfidf_matrix[:len(df1)], tfidf_matrix[len(
    df1):])
```

5. **Map Similar Names**

   After creating the similarity matrix, we can pair the most similar school names while disregarding typos and other minor errors. To achieve this, we set a threshold for the cosine similarity score. School names with a similarity score above the threshold are considered a match, and we pair them together. Additionally, we keep track of the similarity scores for these pairs, which will be used later in the analysis.

```
threshold = 0.1  # Higher threshold for stricter matching

columns_to_update = [...]

for df1_col, df2_col in columns_to_update:
    df1[df1_col] = [
        df2[df2_col].iloc[i] if sim.max() > threshold else None
        for sim, i in zip(similarity_matrix, similarity_matrix.argmax(axis=1))
    ]

df1['მსგავსების ქულა'] = [
    similarity_matrix[idx, similarity_matrix[idx].argmax()]
    for idx in range(len(similarity_matrix))
]
```

6. **Manually delete imprecise couples**

   After merging the names, we filter out pairs of school names with a similarity score of less than 0.98. As mentioned earlier, we have two versions of the dataset, one incorporating multiple buildings for the same school, and the other without them. When working with a table that includes different buildings, we must manually identify invalid pairs and remove them before proceeding with this step.

```
deleteSchools = [...]
renameSchools = {...}
df1 = df1[~df1['სკოლის სახელწოდება'].isin(deleteSchools)]
df1['სკოლის სახელწოდება'] = df1['სკოლის სახელწოდება'].replace(renameSchools)
```

For the table without multiple buildings, we remove every pair with a similarity score below 0.98 and different codes.

```
rows = df1[df1['მსგავსების ქულა'] < 0.98]
deleteCodes = rows[rows['კოდი (ცხრანიშნა)'] != rows['კოდი (ცხრანიშნა) ინფრასტრუქტურაში']]
```

7. **Clean school nameings**

   In both cases, we need to clean the school names by handling irregularities such as normalizing whitespaces and stripping unnecessary characters.

```
df1['სკოლის სახელწოდება'] = df1['სკოლის სახელწოდება'].apply(
    lambda x: preprocess_name(x, remove_punctuation=False, remove_english=False))
```

Next merge `school building and area.xlsx`

Now, we address the issue previously mentioned regarding `school building and area.xlsx`. Initially, this table contained information about:

- Yard area (sq. m.) (ეზოს ფართობი (კვ.მ.))
- Building area (sq. m.) (შენობის ფართობი (კვ.მ))
- Number of classrooms (საკალსო ოთახების რაოდენობა)
- Building capacity (students) (რამდენ მოსწავლეზეა გათვლილი შენობა)
- Number of shifts (საგაკვეთილო ცვლის რაოდენობა)
- Number of buildings (indirectly) (შენობების რაოდენობა)

The first three statistics were already available in the infrastructure-related tables, so we disregarded these columns. However, the remaining information about the number of shifts, the student capacity, and the number of buildings was inconsistent, outdated, and highly inaccurate. As such, we needed to find another, more reliable method to obtain this information.

solution was **web-scrapping**, particularly school's portal. This webpage provides limited but up-to-date information about schools, including the number of shifts and buildings.

To obtain these statistics we used three API endpoints from the portal

- API 1
- API 2
- API 3

These endpoints allowed us to fetch school-level data. Our method involved two steps: first, creating mappings from our database to the schools' portal, and second, retrieving useful information about a particular school from the portal.

1. **Retrieve the data from the API**

   Make an HTTP request to the Specified URL to get the schools' data

```
response = requests.get(urlAllSchools)
response.raise_for_status()
```

2. **Process the retrieved data**

   Extract and structure the relevant information, mapping school names with their corresponding IDs in the portal into a dictionary for easier access.

```
all_schools = response.json()['result']
school_info_map = {school['schoolName']: school['id'] for school in all_schools}
```

3. **Get ID of a school**

```
school_id = school_info_map.get(school_name_in_df1, None)
```

4. **Fetch information about a building number**

   As in Step 1, make an HTTP request to API 2 and API 3, but now save data about an individual school.

```
response_building = requests.get(f'{urlSingleSchool}{school_id}', timeout=10)
response_building.raise_for_status()
building_data = response_building.json()
building_count = building_data.get('buildingCount', None)

response_shift = requests.get(f'{urlSingleSchool}{school_id}/firstgradelimit',
    timeout=10)
response_shift.raise_for_status()
shift_data = response_shift.json()
shift = max(item['shift'] for item in shift_data)
```

   All of the above was done using concurrent operations, in order to, accelerate searching and fetching data. This approach speeds up the process by making multiple API requests in parallel.

   Our function missed some schools in the database because the naming was inconsistent, and the corresponding ID could not be found in the portal. Therefore, we manually searched for these schools and recorded the data.

5. **Create a dictionary {school name : (number of buildings, number of shifts)}**

```
school_list = school_dict =
    {
        'სსიპ - ბოლნისის მუნიციპალიტეტის დაბა კაზრეთის №2 საჯარო სკოლა': (1, 2),
                                ...
    }
```

6. **Update corresponding rows**

   Locate schools that need fixing and update values.

```
for school_name, (num_buildings, num_shifts) in school_dict.items():
    df1.loc[df1[school_column_name] ==
        school_name, ['შენობების რაოდენობა', 'ცვლების რაოდენობა']] = [
            num_buildings, num_shifts]
```

Last merge `school revenues.xlsx`

The last step involves merging the `school revenues.xlsx` file with the data processed so far. There are two scenarios depending on whether the data contains supplementary building information or not.

Scenario 1: Table Consisting Only of the Main Buildings.

1. **Set Multi-level Index**

   We set a multi-level index for both `df1` and `df2` using the columns `Region` ('რეგიონი'), `City/Disctirct` ('ქალაქი/მუნიციპალიტეტი'), and `Code (Nine-Digit)` ('კოდი (ცხრანიშნა)'). This ensures that both DataFrames are aligned based on these columns.

```
df1.set_index(['რეგიონი', 'ქალაქი/მუნიციპალიტეტი', 'კოდი (ცხრანიშნა)'], inplace=True)
df2.set_index(['რეგიონი', 'ქალაქი/მუნიციპალიტეტი', 'კოდი (ცხრანიშნა)'], inplace=True)
```

2. **Combine DataFrames Based on the Common Index**

   After setting the index, we use `pd.concat()` to combine `df1` and `df2` based on the common index. The `axis=1` parameter ensures the DataFrames are concatenated column-wise, and the `join='inner'` ensures that only rows with matching indices are kept.

```
df_combined = pd.concat([df1, df2], axis=1, join='inner').reset_index()
```

Scenario 2: Table with Complementary Buildings. If the table includes additional building data, the merging process is simpler.

1. **Merge DataFrames on the Common Column**

   We directly merge both tables on the common `Code (Nine-Digit)` ('კოდი (ცხრანიშნა)') column.

```
df_combined = pd.merge(df1, df2, on='კოდი (ცხრანიშნა)')
```

2. **Drop and Rename Columns**

   After merging, we drop columns that are redundant or unnecessary, such as duplicate information from `df2`. We then rename columns to standardize the naming across both DataFrames.

```
df_combined = df_combined.drop(columns=[...])
df_combined = df_combined.rename(columns={...})
```

Recap. In this section, we worked step by step to turn raw data into a clean and unified dataset ready for analysis. Starting with five separate Excel files, we carefully cleaned and organized the data by fixing names, removing unnecessary columns, and making everything consistent. To deal with differences in school names across the files, we used techniques like TF-IDF and cosine similarity, which helped us match names even if they had small errors. We also gathered extra information through web scraping and manual checks to fill in any missing details.

After completing the data cleaning and merging steps, the next tasks will involve label formatting and filling any remaining null values. These additional steps will ensure that the dataset is fully prepared for analysis, with standardized labels and all missing data properly addressed.

3.2.3 Missing Value Analysis and Reduction

Handling missing values is a crucial step in the pre-processing, as it guarantees the dataset's completeness and authenticity for further analysis. In this subsection, we analyze the extent of missing values in our dataset, categorize rows based on their missing values, and apply systematic reductions by setting thresholds for both columns and rows.

- **Categorizing Rows by Missing Values**

  To better understand the distribution of missing values in rows, we categorized them into bins of 5, assigning labels such as "0-5", "5-10", *etc.* This was implemented using the following:

```
def analysisNullValue(df, n):
    bins = np.arange(0, n + 1, 5)
    labels = [f"{i}-{i + 5}" for i in range(0, n - 4, 5)]
    missing_counts = df.isna().sum(axis=1)
    categories = pd.cut(missing_counts, bins=bins, labels=labels, right=True)
    result = {
        label: [(df.loc[idx]['სკოლის სახელწოდება'], missing_counts[idx])
                for idx in missing_counts.index[categories == label]]
        for label in labels
    }
    return result
```

  This categorization provided an organized view of the missing data, enabling decision-making on how to handle them.

- **Removing Columns with High Missing Values Percentages**

  Table 3.1 shows that there are columns with more than 60% missing values, making them redundant and easy to remove. We chose 40% as the threshold because such columns, despite having fewer null values, do not carry as much useful information for our project. Their relatively high level of incompleteness made them unsuitable for analysis.

Different Columns	**Null Percentage (%)**
Condition of IV Floor	89.44
Condition of III Floor	67.03
Condition of Biology Lab/Classroom	64.95
Condition of Chemistry Lab/Classroom	63.92
Condition of Physics Lab/Classroom	63.41
Condition of Combined Lab (Physics, Chemistry, Biology)	58.92
Individual Heating System (Gas, Diesel, Coal, Other)	51.00
Year of Central Heating Installation/Rehabilitation	46.75
Number of Sport Halls	31.78
Condition of Sport Halls	31.63
...	...

Table 3.1: Null Values Percentage by Columns (Biggest 10)

  The percentage of missing values for each column was calculated using.

```
missing_counts = df[cols].isna().sum()
missing_percent = missing_counts / len(df) * 100
cols_to_drop = cols[missing_percent > 40]
df.drop(columns=cols_to_drop, inplace=True)
```

  This step ensured that only columns with sufficient data were retained for further analysis.

- **Dropping Rows with Excessive Missing Data**

  There were also schools with little to no information, so we deleted such entries. Rows with more than 20 missing values were eliminated from the dataset. This threshold was chosen

as it represents more than half of the columns, hence rows that had drastic differences between the number of NA and non-NA values were deleted.

```
df.dropna(axis=0, thresh=n, inplace=True, subset=cols)
```

3.2.4 Reformatting columns

First, we analyze and reformat each column to improve the dataset's usability and reliability.

To achieve this, we categorized the columns into the following hypothetical types: **`indexical`**, **`conditional`**, **`numerical`**, and **`financial`**.

- **`Indexical` columns**

  These columns are used as an index for the final combination of the datasets. Such columns are consistent and do not contain null values (we will not be changing these columns.)

- **`Conditional` columns**

  These columns describe the condition of specific aspects of the school. Their values should be chosen from a predetermined set of options but often contain instances of human error.

  For conditional columns, we came up with a new unified set of values

  $$\{\text{does not have},\ \text{bad},\ \text{acceptable},\ \text{good}\}$$

  that fits every conditional column with three exceptions, each needing a separate set of unique values. Then we use `.unique()` method to find unique values and manually create a specialized key-value set to map them to our predetermined sets of values.

  An example of a specialized key-value pair:

```
"map_values": {
    'კარგი': ['სრულად რეაბილიტირებული'],
    'დამაკმაყოფილებელი': ['ძველი რეაბილიტირებული', 'ნაწილობრივ რეაბილიტირებული'],
    'ცუდი': ['არ არის რეაბილიტირებული', 'სარეაბილიტაციო', 'მთლიანად სარეაბილიტაციო'],
    'არ არსებობს': ['არა']
    }
```

  For one of the exceptional column `Central heating - (gas, diesel, coal, firewood, electricity, briquettes, solar system, other)` ('ცენტრალური გათბობა - (გაზი, დიზელი, ქვანახშირი, შეშა, ელექტროენერგია, ბრიკეტები, მზის სისტემა, სხვა)') we created the method:

```
def heating_system_type(value):
    value = str(value)
    if pd.isnull(value) or value ==
        'არ არსებობს' or value == 'სხვა':  # Handle NaN values
        return value
    # remove punctuation
    value = re.sub(r'[^\w\s]', ' ', value)
    result = ''
    if bool(re.search(r'გა.ი|გ.ზ|ცენტრ|ბუნებრივი აირი', value)):
        result = result + 'გაზი '
    if bool(re.search(r'[შს]ე[შს]', value)):
        result = result + 'შეშა '
    if bool(re.search(r'ბრ[იეუგ][კლი]', value)):
        result = result + 'ბრიკეტი '
    if bool(re.search(r'დიზ', value)):
        result = result + 'დიზელი '
```

```
    if bool(re.search(r'ენერ|დენ|ელექტრო|ელ გა', value)):
        result = result + 'ელექტროენერგია '
    if bool(re.search(r'ხშირ', value)):
        result = result + 'ქვანახშირი '
    if bool(re.search(r'ნაჭ', value)):
        result = result + 'თხილის ნაჭუჭი '
    if len(result) != 0:
        # remove the extra white space
        return result[:-1]
    return None
```

Where we used direct observation to create a set of patterns and `re.search()` method to reduce the number of unique values for the column from 146 to 26.

- **`Numerical` columns**

  These columns display a quantity of specific aspects of schools. However, they may occasionally include unnecessary information.

  For numeric columns, we created four special methods to address specific obstacles presented by such columns.

  - **yard_area method**

    This method extracts a list of numbers from each value and returns the first instance of a continuous string of digits, with one exception where the first two instances are concatenated.

```
def yard_area(value):
    numbers = re.findall(r'\d+', str(value))
    if pd.isnull(value):
        return value
    elif value == '13 360':
        return int(numbers[0] + numbers[1])
    elif numbers:
        return int(numbers[0])
```

    this method is used for `Yard Area (sq.m.)` 'ეზოს ფართობი (კვ.მ)' column.

  - **building_area method**

    This method differs from the `yard_area` method in exception handling. Suppose the length of the number list is greater than two. The method dynamically determines how many of the largest numbers to sum using `nlargest(np.sqrt(len(numbers)).astype(int))`, where the square root of the number of elements in the list is calculated. This special condition applies to entries that have multiple buildings, each represented by a digit. The exception disregards denominational numbers and sums only the areas of the buildings.

```
def building_area(value):
    numbers = re.findall(r'\d+', str(value))
    if pd.isnull(value):
        return value
    if len(numbers) > 2:
        return pd.Series(numbers).astype(np.int_).nlargest(np.sqrt(len(
    numbers)).astype(int)).sum()
    else:
        return int(numbers[0])
```

    this method was used for `Building area in use (sq.m)` ('შენობის ფართი, რომელიც გამოყენებაშია (კვ.მ)') column.

– **floor_number method**
This method determines the highest number of floors in any building that belongs to a given school. It includes a built-in key-value mapping set, selected through manual observation. The method removes everything except the digits and selects the highest digit appearing in the value. This approach is effective because no school has more than nine floors.

```
def floor_number(value):
    # Mapping of number words to digits
    word_to_digit = {
        'ერთ': '1',
        'ორ': '2',
        'სამ': '3',
        'III': '3',
        'IV': '4'
    }
    if pd.isnull(value):  # Handle NaN values
        return value

    # Convert the value to a string for processing
    value = str(value)

    # Replace number words with corresponding digits
    for word, digit in word_to_digit.items():
        value = re.sub(rf'\b{word}\b', digit, value)  # Replace whole words
    only

    # Extract all numeric characters
    digits = re.findall(r'\d', value)
    if digits:
        # Find the highest digit and return it as an integer
        return int(max(digits))
    return None  # If no digits are found, return None
```

this method was used for `Number of Floors` ('სართულების რაოდენობა') column.

– **heating_system_date method**
The column `Year of installation or complete rehabilitation of central heating` ('ცენტრალური გათბობის მოწყობის ან სრული რეაბილიტაციის წელი') represents the year of construction or rehabilitation of the heating system. Some entries contain multiple dates. This method removes all non-numeric characters and extracts the last four digits to retain the latest year.

```
def heating_system_date(value):
    if pd.isnull(value):  # Handle NaN values
        return value
    # Extract numbers from the string using regex
    numbers = re.findall(r'\d+', str(value))
    if numbers:
        # Combine all numbers found and get the last four digits
        last_four = ''.join(numbers)[-4:]
        return int(last_four)
    return None  # Return None if no numbers are found
```

- **Combined integration of `Conditional` and `Numerical` parts**

Now, we combine everything into a key-value set, `Filters`, where the key is the name of a column, and the value is a pair consisting of a specific function and mapping instructions tailored to that column. Each key-value pair is then passed to the `map_column_values` method, which reconstructs the column accordingly.

– **map_column_values method**

```
def map_column_values(df, column, mapping_dict, func, datatype):
    if mapping_dict is None:
        mapping_dict = {}
    reverse_mapping = {v:k for k, values in mapping_dict.items() for v in
    values}

    # Apply the mapping to the column
    df[column] = df[column].apply(lambda x: reverse_mapping.get(x, x))
    df[column] = df[column].apply(func)
    if datatype == "Int64":
        df[column] = pd.to_numeric(df[column]).astype(pd.Int64Dtype())
    else:
        df[column] = df[column].astype(datatype)
    return df
```

First, we separate and reverse the mapping. Each element in the value list becomes the key, and the original key, to which it was meant to be mapped, becomes the value. This transformation changes the mapping from a one-to-many structure to a one-to-one mapping.

```
reverse_mapping = {v: k for k, values in mapping_dict.items() for v in values
    }
```

Then, we apply the mapping to the column values if they match any key, followed by the specialized function.

```
df[column] = df[column].apply(lambda x: reverse_mapping.get(x, x))
df[column] = df[column].apply(func)
```

Finally, we convert the column type when necessary.

- **`Financial` columns**

  These columns contain budget-related information. They typically represent sums or differences of other financial columns.

  The reconstitution of financial columns is separate from other columns. First, we need to check if the dependencies are correct. The `check_outliers` method is used to manually check these dependencies.

```
def check_outliers(df, categories):
    result = []
    for category in categories:
        column_name = category['sum_column']
        sub_columns = category['sum_categories']

        # Calculate the sum of sub-columns for each row
        df['calculated_sum'] = df[sub_columns].sum(axis=1, skipna=True).round()

        # Compare the calculated sum with the original column values
        df['difference'] = (df[column_name] - df['calculated_sum']).round()

        # Identify rows where the difference is not within the allowed range (-2
    to 2)
        outliers = df.loc[~df['difference'].between(-2, 2), ['calculated_sum',
    column_name, 'difference']]

        if not outliers.empty:
            result.append(outliers)

```

```
        # Drop temporary columns
        df.drop(columns=['calculated_sum', 'difference'], inplace=True)

    return result
```

The variable `categories` is a directory of column names that highlights the dependencies of the said columns. The (-2, 2) range is allowed to account for the rounding error of the floating-point numbers. The `check_outliers` method can be fine-tuned to fit each specific dependency. After a thorough manual inspection, we found four schools to be removed where either budget did not match or contained null values in the accumulative columns.

There were also fifteen instances where the accumulative accounts were not null, but the individual accounts were. Fourteen instances were in the administrative office budget and one was in the special needs teachers' budget. We will be using regression to predict the distribution of the accumulative number over the individual accounts in Subsection 3.3.1.

Besides these working-case columns, the data is consistent; hence, we can overcome the problem of null values by writing zeros in their places.

```
working_columns = [
  'დირექცია, ადმინისტრაციულ-ტექნიკური-პერსონალის შრომის ანაზღაურება',
  'მ.შ. დირექცია, ადმინისტრაციულ-ტექნიკური-პერსონალის,
     ინკლიზიური განათლების მხარდამჭერი სპეციალისტები თანამდებობრივი სარგო (ხელფასი)',
  'მ.შ. დირექცია, ადმინისტრაციულ-ტექნიკური-პერსონალის პრემია',
  'მ.შ. სპეც მასწავლებლის შრომის ანზღაურება (საათობრივი დატვირთვა და სხვა
     დანამატები, სქემის ფარგლებში გათვალისწინებული დანამატების გარდა)',
  'მ.შ. სპეც მასწავლებლის სქემის ფარგლებში გათვალისწინებული დანამატების ოდენობა',
  'მ.შ. სპეც მასწავლებელთა პრემია'
]

for column in df.columns[df.columns.get_loc(
    ('2023 წლის ბიუჯეტი'):].difference(working_columns[1:]):
    df.loc[pd.isna(df[column]), column] = 0
```

Where `working_columns` are the columns that contain the special cases. To eliminate null values in `working_columns`, we check if the accumulative account is not null but the individual accounts are, and write zero if these conditions are not met.

```
# administration budget fixes
for i in working_columns[1:3]:
    df.loc[pd.isna(df[i]) & ~(~pd.isna(df[working_columns[0]]) & pd.isna(df[
    working_columns[1]]) & pd.isna(
        df[working_columns[2]])), i] = 0

# special-edd budget fixes
for i in working_columns[3:]:
    df.loc[pd.isna(df[i]) & (
            df['სკოლის სახელწოდება'] !=
            'სსიპ - ხულოს მუნიციპალიტეტის სოფელ დიდაჭარის საჯარო სკოლა'), i] = 0
```

3.2.5 Outlier Detection and Removal

Outliers are data points that deviate remarkably from the rest of the dataset. They are a result of measurement errors and data entry issues, as most of the data is handwritten by people. Regardless of their source, identifying them is an integral step. This ensures that analyses and machine learning models are not skewed or biased by extreme values.

We explore various statistical and machine learning-based methods for detecting outliers, such as Z-score, IQR method, Isolation Forest, DBSCAN, and Local Outlier Factor (LOF), as well as

ensemble detection. For dimension reduction, we use Principal Component Analysis (PCA). We will not discuss these concepts in detail and use them as needed for outlier detection. Readers can find further information about these in [1, 12, 2].

- **Select numerical columns**

  The following columns were selected for outlier analysis as they have numerical values and represent key metrics with unusual data points.

  - `Number of students` (მოსწავლეთა რაოდენობა)
  - `Building area in use (sq.m)` (შენობის ფართი, რომელიც გამოყენებაშია (კვ.მ))
  - `Plot area (sq.m)` (ეზოს ფართობი (კვ.მ))
  - `Number of floors` (სართულების რაოდენობა)
  - `Number of restroom facilities` (საპირფარეშო ოთახების რაოდენობა)
  - `Number of classrooms` (საკლასო ოთახების რაოდენობა)

- **Outlier detection techniques**

  - **Z-Score method**

    The Z-score measures how many standard deviations an observation is from the mean. Any observations with a Z-score above or below some threshold are typically considered outliers.

```python
def zscoreTest(df, column, normalize=False, threshold=3):
    zscoreDf = df.copy()
    if normalize:
        zscoreDf[column] = np.log1p(zscoreDf[column])
    zscores = zscore(zscoreDf[column], nan_policy='omit')
    return zscoreDf[np.abs(zscores) > threshold]
```

  - **Iterquartile Range (IQR) method**

    This method identifies outliers as values that are outside of the range $[Q1 - 1.5 \cdot IQR : Q3 + 1.5 \cdot IQR]$

    where Q1 and Q3 are the first and third quartiles, and $IQR = Q3 - Q1$

```python
def iqr_test(df, column):
    iqr_df = df.copy()
    Q1 = iqr_df[column].quantile(0.25)
    Q3 = iqr_df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = iqr_df[(iqr_df[column] < lower_bound) | (iqr_df[column] >
    upper_bound)]
    return outliers.dropna()
```

  - **DBSCAN method**

    DBSCAN identifies clusters as high-density regions separated by low-density areas, allowing arbitrary shapes. Core samples form clusters, while non-core samples connect to them. Parameters `min_samples` and `eps` define density. Outliers are points far from core samples. Visualization distinguishes clusters, core samples, non-core samples, and outliers [14].

```
def dbscan_test(df, column, eps=0.5, min_samples=5, dropna=True):
    dbscan_df = df[column].copy()
    dbscan_df = dbscan_df.dropna() if dropna else dbscan_df.fillna(0)
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    labels = dbscan.fit_predict(dbscan_df)
    outlier_indices = dbscan_df[labels == -1].index
    return df.loc[outlier_indices]
```

– **Local Outlier Factor (LOF) method**

The Local Outlier Factor (LOF) is an anomaly score that determines how much a sample deviates from its neighbors in terms of density. It is defined by using k-nearest neighbors to measure the local density of a sample. If a sample has a significantly lower density compared to its neighbors, it is considered an outlier based on its isolation in the local neighborhood [3]. LOF is used for novelty detection you *must* not use `predict`, `decision_function` and `score_samples` functions on the training set as this would lead to wrong results, hence for us only manual observation is a choice.

```
def lof_test(df, column, n_neighbors=20, contamination=0.1, normalize=False,
    dropna=True):
    lof_df = df[column].copy()
    lof_df = lof_df.dropna() if dropna else lof_df.fillna(0)
    if normalize:
        lof_df[column] = np.log1p(lof_df[column])
    lof = LocalOutlierFactor(n_neighbors=n_neighbors, contamination=
    contamination)
    labels = lof.fit_predict(lof_df)
    outlier_indices = lof_df[labels == -1].index
    return df.loc[outlier_indices]
```

– **Isolation Forest method**

Isolation Forest identifies anomalies by isolating data points through recursive random partitioning, represented as tree structures. Anomalies are isolated with shorter path lengths compared to normal points. The average path length across a forest of trees serves as a normality measure, with shorter lengths indicating potential anomalies [15].

```
def isolation_forest_test(df, column, n_estimators=100, contamination=0.01,
    random_state=42, normalize=False,
              dropna=True):
    iso_df = df[column].copy()
    iso_df = iso_df.dropna() if dropna else iso_df.fillna(0)
    if normalize:
        iso_df[column] = np.log1p(iso_df[column])
    iso_forest = IsolationForest(n_estimators=n_estimators, contamination=
    contamination, random_state=random_state)
    iso_df['Anomaly_Score'] = iso_forest.fit_predict(iso_df)
    outlier_indices = iso_df[iso_df['Anomaly_Score'] == -1].index
    return df.loc[outlier_indices]
```

– **Ensemble method**

This method aggregates results from multiple algorithms, combining their outlier scores for a decision.

```
def ensemble_outlier_detection(df, column, methods, dropna=True, threshold=
    None):
    if threshold is None:
        threshold = len(methods) - 2
    scores_df = df[column].copy()
```

```
    scores_df = scores_df.dropna() if dropna else scores_df.fillna(0)
    scores_df['Outlier_Score'] = 0  # Initialize combined score

    # Loop through each method
    for method_name, method_func in methods.items():
        # Get outlier indices from the method
        outlier_indices = method_func(df, column)

        # Mark outliers in the combined score
        scores_df.loc[outlier_indices.index, 'Outlier_Score'] += 1

    # Determine final outliers based on a score threshold
    final_outliers_idx = scores_df[scores_df['Outlier_Score'] > threshold].
    index
    final_outliers = df.loc[final_outliers_idx]

    return final_outliers
```

– **Iterative LOF method**

We also tried iterating LOF method several times. Each iteration identifies the most significant outliers, removes them, and repeats the process on the remaining dataset.

```
def iterative_lof_removal(data, n_neighbors=45, max_iterations=5,
    contamination='auto'):
    remaining_data = data.copy()
    all_outliers = []
    outlier_counts = {}

    for iteration in range(max_iterations):
        lof = LocalOutlierFactor(n_neighbors=n_neighbors, contamination=
    contamination)
        lof_labels = lof.fit_predict(remaining_data)
        outlier_indices = np.where(lof_labels == -1)[0]

        if len(outlier_indices) == 0:  # Stop if no outliers are found
            print(f"No more outliers detected at iteration {iteration + 1}.")
            break

        # Store outliers and their counts
        outliers = remaining_data[outlier_indices]
        all_outliers.append(outliers)
        outlier_counts[f"Iteration {iteration + 1}"] = len(outlier_indices)

        # Remove outliers
        remaining_data = np.delete(remaining_data, outlier_indices, axis=0)

    return remaining_data, all_outliers, outlier_counts
```

- **Choosing the method**

We conducted experiments by running these methods with different parameters, `n_neighbors`, `n_estimators`, `contamination` *etc.*

```
# 1. Z-score
zscores = zscoreTest(dfNumericalColumns, cols, normalize=True, threshold=3)

# 2. IQR Method
iqr_outliers = iqr_test(dfNumericalColumns, cols)
# 3. Isolation Forest
iso_forest_outliers = isolation_forest_test(dfNumericalColumns, cols,
    n_estimators=136,
```

```
                      contamination='auto', random_state=42, normalize=True,
                      dropna=True)
# 4. DBSCAN
dbscan_outliers = dbscan_test(dfNumericalColumns, cols, eps=500, min_samples=2,
    dropna=True)
# 5. LOF Parameters
lof_outliers = lof_test(dfNumericalColumns, cols, n_neighbors=45, contamination='
    auto', normalize=True, dropna=False)
# 6. Ensemble Outlier Detection
methods = {
  'IQR': lambda df, col: iqr_test(df, col),
  'Isolation Forest': lambda df, col: isolation_forest_test(df, col, n_estimators
    =136,
                                contamination=0.5, random_state=42,
                                normalize=True),
  'DBSCAN': lambda df, col: dbscan_test(df, col, eps=100, min_samples=2),
  'LOF': lambda df, col: lof_test(df, col, n_neighbors=45, contamination=0.5,
    normalize=True)
}

ensemble_outliers = ensemble_outlier_detection(dfNumericalColumns, cols, methods,
     dropna=False)
# 7. Iterative LOF
remaining_data, all_outliers, outlier_counts = iterative_lof_removal(
  df, n_neighbors=45, max_iterations=4, contamination=0.04
)
```

We have chosen union of Local Outlier Factor and Iterative LOF method as our preferred method because LOF adapts to local density, making it effective for our datasets. It works on non-parametric data, does not assume any specific data distribution, and extends easily to high-dimensional datasets. The iterative method ensures that some of the more hidden outliers are distinguished and observed.

Parameters were selected using GridSearch, which exhaustively evaluates combinations of hyperparameter values to identify the best-performing model. The optimal parameter set was determined through inspection and debugging.

```
def optimize_lof(df, column, normalize=False, dropna=True):
    n_neighbors_range = range(10, np.sqrt(len(df)).astype(int), 2)
    contamination_range = np.linspace(0.01, 0.2, 20)

    lof_df = df[column].copy()
    lof_df = lof_df.dropna() if dropna else lof_df.fillna(0)
    if normalize:
        lof_df[column] = np.log1p(lof_df[column])
    result = []
    for n_neighbors in n_neighbors_range:
        for contamination in contamination_range:
            lof = LocalOutlierFactor(n_neighbors=n_neighbors, contamination=
    contamination)
            labels = lof.fit_predict(lof_df)
            outlier_indices = lof_df[labels == -1].index

            result.append({
                'n_neighbors': n_neighbors,
                'contamination': contamination,
                'outliers': lof_df.loc[outlier_indices]
            })
    return result
```

- **Visualization of outliers**

For visualization, we need to convert our multidimensional space into some comprehensible dimension in order to observe outliers. Here, Principal Component Analysis (PCA) was used. PCA is a linear dimensionality reduction algorithm using Singular Value Decomposition of the data to project it to a lower dimensional space. It decomposes a multivariate dataset into a set of successive orthogonal components that explain a maximum amount of the variance.

```
pca_2d = PCA(n_components=2)
dfNumericalColumns_scaled = StandardScaler().fit_transform(
  dfNumericalColumns.fillna(dfNumericalColumns.mean().astype("Int64")))
df_pca_2d = pca_2d.fit_transform(dfNumericalColumns_scaled)

pca_2d_df = pd.DataFrame(
  data=df_pca_2d,
  columns=["Principal Component 1", "Principal Component 2"],
  index=dfNumericalColumns.index)
```

(PCA) was used to reduce the dataset's dimensionality for visualizing outliers. Two configurations were tested: 2 components and 3 components.

2 Components: the first two components explained 67% of the variance, enabling straightforward 2D visualization. This approach is simple and interpretable but may lose subtle patterns due to reduced variance retention.

3 Components: using three components increased the explained variance to approximately 80%, offering a more accurate representation of the dataset, hence our choice of component number was 3

```
def perform_pca(data, n_components):
    tmp = data.copy()
    scaled_data = StandardScaler().fit_transform(
        tmp.fillna(0)
    )

    # Apply PCA
    pca = PCA(n_components=n_components)
    transformed_data = pca.fit_transform(scaled_data)

    # Create a DataFrame for transformed data
    columns = [f"Principal Component {i + 1}" for i in range(n_components)]
    transformed_df = pd.DataFrame(data=transformed_data, columns=columns, index=
    data.index)

    # Calculate explained variance
    explained_variance_ratio = pca.explained_variance_ratio_.sum() * 100

    return transformed_df, explained_variance_ratio, pca, transformed_data

pca_2d_df, explained_variance_ratio_2d, pca_2d, transformed_data_2d = perform_pca
    (dfNumericalColumns, n_components=2)
pca_3d_df, explained_variance_ratio_3d, pca_3d, transformed_data_3d = perform_pca
    (dfNumericalColumns, n_components=3)
print(f'Explained variance ratio (2D): {explained_variance_ratio_2d:.2f}\%')
print(f'Explained variance ratio (3D): {explained_variance_ratio_3d:.2f}\%')

>>>Explained variance ratio (2D): 67.54\%
>>>Explained variance ratio (3D): 78.86\%
```

In Figure 3.4 and Figure 3.5, the distribution of data in reduced dimensions can be observed. The 2D plot provides a simple and intuitive visualization, but the 3D plot captures more

variance (78.86% compared to 67.54%), offering a more accurate representation of the dataset.

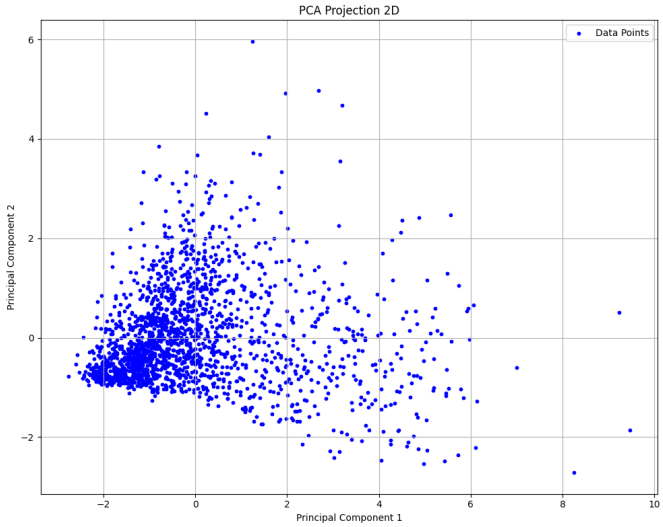


Figure 3.4: PCA points with 2 Component

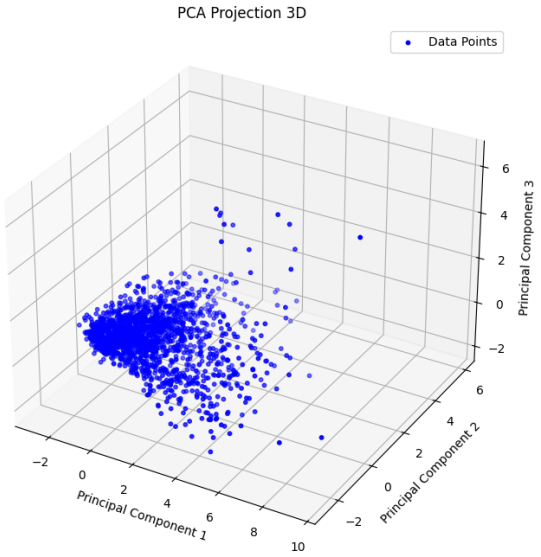


Figure 3.5: PCA points with 3 Component

Figure 3.6 and Figure 3.7 display the detected outliers using the Local Outlier Factor

(LOF) combined with an iterative removal method. Outliers are marked clearly, appearing in low-density regions of the PCA plot.

The LOF algorithm was first used with the original high-dimensional dataset to recognize initial outliers. Subsequently, the iterative LOF method was executed in the reduced-dimensional PCA space. This two-stage approach allowed for the identification of both obvious and more indistinct outliers that might have been overlooked using a single technique. By applying PCA, we projected the data into a comprehensible 2D and 3D space for visualization.

Curiously, some outliers pop up in high-density regions. This is a result of the combined approach where the pure LOF method exposed anomalies based on the original data density, while the iterative method polished these results in the reduced PCA space. Through manual inspection and parameter tuning, this strategy displayed optimal performance in identifying outliers in our dataset.

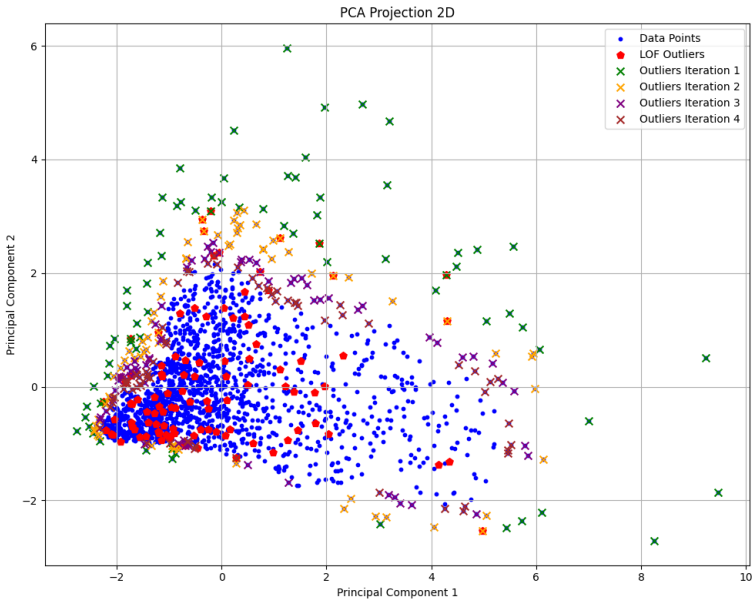


Figure 3.6: PCA points and outliers with 2 Component

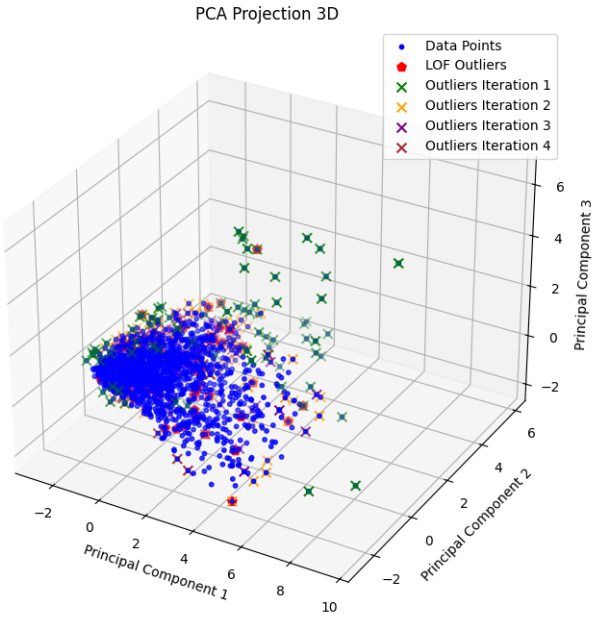


Figure 3.7: PCA points and outlier with 3 Component

The iterative removal process refines clusters by removing outliers in successive steps, as reflected in tighter clusters in the PCA-reduced space. While PCA offers valuable insights into the distribution of data, it is worth noting that it captures only linear relationships, and non-linear patterns might require additional methods for analysis.

- **Mapping to original data points**

  After capturing outliers in the PCA-transformed space, we need to map them back to the original high-dimensional dataset to further analyze their characteristics. This mapping ensures that the vision gained from the PCA space can be applied to real-world data.

  To achieve this, the indices of outliers in the transformed PCA space were matched with their corresponding indices in the original dataset.

```
def map_outliers_to_original_data(original_data, transformed_data, all_outliers):
    mapped_outliers = []

    for iteration, outliers in enumerate(all_outliers):
        # Find indices of outliers in the transformed data
        outlier_indices = [
            np.where((transformed_data == outlier).all(axis=1))[0][0]
            for outlier in outliers
        ]
        # Map back to the original data
        mapped_outliers.append(original_data.iloc[outlier_indices])

    # Combine all mapped outliers into a single DataFrame
```

```
    all_mapped_outliers = pd.concat(mapped_outliers)

    return all_mapped_outliers

all_mapped_outliers_3d = map_outliers_to_original_data(dfNumericalColumns,
    transformed_data_3d, all_outliers_3d)
```

- **Unify the methods and remove outliers**

  The final step involved merging the detected outliers from both approaches, followed by removing the identified data points from the dataset.

```
unionPCA_LOF = pd.concat([all_mapped_outliers_3d, lof_outliers])
unique_union_indices = unionPCA_LOF.index.unique()
df = df.drop(unique_union_indices)
```

3.3 Implementing regression models on real data

In this section, we will use the implementations of regression models discussed in Section 2.2 on the processed data to evaluate certain characteristics of the Georgian educational system. We aim to summarize the usefulness of the data and highlight the insight that can be derived from it.

3.3.1 Regression Predictions on null value financial data

We will predict and fill in the fifteen instances of financial data where the individual columns were null, but the cumulative column was not, as discussed in Subsection 3.2.4. The cumulative column contains the predictor variables, while the individual columns contain the predicted variables. We will train the model using the rows where the cumulative value is not zero and the individual columns are not null. After training, we will predict the null values in the individual columns based on the corresponding cumulative value.

- **perform_regression method**

  This method filters and defines the data, splits it into training and test datasets, fits the training data to the regression model, and predicts based on the test values. It then scores the predictions against the true test data to assess the quality of the regression.

  `feature_col` refers to the columns containing the predictor values (*i.e.*, the cumulative columns), while `target_cols` refers to the columns containing the predicted values (*i.e.*, the individual columns).

```
def perform_regression(df, feature_col, target_cols, model, test_size=0.2,
    random_state=42, print_prediction=False,
                       save_predictions=False, print_scores=False):
    # Filter valid rows (non-zero feature values and no missing targets)
    filtered_df = df.loc[
        df[feature_col].ne(0) & df[target_cols].notna().all(axis=1),
        [feature_col] + target_cols
    ]
```

  Split `filtered_df` into corresponding X and Y values.

```
    # Define features (X) and targets (Y)
    X = filtered_df[feature_col].to_numpy().reshape(-1, 1)
    Y = filtered_df[target_cols].to_numpy()
```

The `train_test_split` method performs seeded splits with the ratio determined by `test_size`, where the seed is set using `random_state`.

```
    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_size
    , random_state=random_state)
```

We then fit our training data into the model and calculate some accuracy metrics. The `mean_squared_error` method calculates the mean squared error between the predicted values and the true test values, while the `r2_score` computes the R^2 coefficient.

```
    # Fit the model
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Scoring
    scores = {
        "MSE": mean_squared_error(y_test, y_pred),
        "R2": r2_score(y_test, y_pred)
    }
    if print_scores:
        print("Scores:", scores)
```

To predict the null values, we simply extract the rows that contain the null value, extract the predictor values, predict the new values, and store them according to their index.

```
if print_prediction:
    nan_rows = df.loc[df[filtered_df.columns].isna().any(axis=1)]

if not nan_rows.empty:
    nan_X = nan_rows[feature_col].to_numpy().reshape(-1, 1)
    nan_predictions = model.predict(nan_X).round()

    if save_predictions:
        df.loc[nan_rows.index, target_cols] = nan_predictions
```

We will use the Ridge regression model described in Subsection 2.2.5

```
ridge_reg = Ridge(alpha=1, fit_intercept=True, solver="cholesky")
```

We will fit the `perform_regression` method to administrative budget columns, and special education teachers' budget columns individually.

```
# Perform regression for Admin columns
admin_results = perform_regression(
    df,
    feature_col=cols_Admin[0],
    target_cols=cols_Admin[1:],
    model=ridge_reg,
    print_prediction=True,
    save_predictions=True,
    print_scores=True
)
>>>mean_squared_error(y_test, y_pred)
12612644.378500288
>>>r2_score(y_test, y_pred)
0.591765430056786

# Perform regression for Special Education columns
spec_results = perform_regression(
```

```
    df,
    feature_col=cols_Spec_Ed[0],
    target_cols=cols_Spec_Ed[1:],
    model=ridge_reg,
    print_scores=True,
    save_predictions=True,
    print_prediction=True
)
>>>mean_squared_error(y_test, y_pred)
15687477.479263967
>>>r2_score(y_test, y_pred)
0.48186993748916707
```

where `cols_Admin` and `cols_Spec_Ed` contain the names of the columns respectfully.

After training the model, we applied it to predict the null values in the financial columns for both administrative and special education budgets, which were outlined in the Subsection 3.2.4.

For administrative budget columns, the predictor variable was the cumulative administrative budget, while the target variables included individual components of the administrative budget. You can see predictions in table 3.2

Feature Values	Predicted Targets Values
58,240	[55,942, 2,298]
88,520	[85,617, 2,903]
42,900	[40,909, 1,991]
46,690	[44,623, 2,067]
36,205	[34,347, 1,858]

Table 3.2: Some predictions for administrative budget columns

For *special education teacher budgets*, the cumulative budget served as the feature, while the target variables included individual budget components related to specific teacher payments and bonuses, see Table 3.3.

Feature Values	Predicted Targets Values
1,915	[689, 1,187, 39]

Table 3.3: Predictions for special education teacher budgets

In the process of looking for the best parameters and their scores, we used the `grid_search` method again to use the variety of parameters in `param_grid`.

```
# Finding best score/parameters
param_grid = {
    "alpha": [0.1, 1, 10, 100,1000,10000],  # Regularization strength
    "solver": ["auto", "svd", "cholesky", "lsqr", "saga"],  # Solvers to try
}

grid_search = GridSearchCV(
    estimator=ridge_reg,
    param_grid=param_grid,
    scoring="r2",  # Use negative MSE for regression
    cv=5,  # 5-fold cross-validation
    verbose=1,
    n_jobs=-1,  # Use all available processors
)

```

```
16
17 # Perform regression for Admin columns
18 admin_results = perform_regression(
19   df,
20   feature_col=cols_Admin[0],
21   target_cols=cols_Admin[1:],
22   model=grid_search,
23   print_prediction=True,
24   print_scores=True
25 )
26
27 >>>grid_search.best_params_
28 {'alpha': 1, 'solver': 'saga'}
29 >>>grid_search.best_score_
30 0.529909084601138
```

Methods `.best_params_` and `.best_score_` simply display the parameters from the `param_grid` that scored the highest, and the score that is based on `mean cross-validated`, which is an average score across multiple folds of the data. We used the `Ridge` solver since trying out the optimal parameters from `grid_search` yielded similar scores.

3.3.2 Predicting the 2023 budget using regression models

In our database, we have a variety of information related to schools, from the condition of bathrooms to revenue data. We have a column named `2023 Budget`, which we attempt to predict using linear and ridge regressions. We use the following procedure:

1. **Encode Categorical Columns**

   We created one unified categorical representation for these columns in Subsection 3.2.4. Now we need to encode them into labels, which can be easily done because we have already introduced order in our representations.

   $$[\text{does not exist}\,, \text{bad}\,, \text{satisfactory}\,, \text{good}] \to [0, 1, 2, 3]$$

   Hence we only need to invert this mapping.

   We select only categorical columns and then map them to their corresponding value, excluding negative numbers as they represent NaN values

   ```
   1 categorical_cols = df.select_dtypes(include=['category']).columns
   2 df[categorical_cols] = df[categorical_cols].apply(lambda x: x.cat.codes.where(x.
         cat.codes >= 0, np.nan))
   ```

   We take into account only half of our database without joining, as described in Subsection 3.2.2 because we aim to predict the budget based on the infrastructure of the school.

2. **Find Most Correlated Columns**

   We calculate the correlation between the `2023 Budget` and other columns using Formula 2.1.3, sort them in descending order, and select the top four columns. This number was determined by testing various column counts to optimize the R^2 score with respect to runtime. After, we drop any NaN values.

   ```
   1 cols = [...]
   2
   3 correleted_columns = df[cols].select_dtypes('number').corr()
   4     ['2023 წლის ბიუჯეტი'].sort_values(ascending=False).nlargest(5)
   5 working_df = df[correleted_columns.index].dropna().copy()
   ```

Decompose the given data frame into X and y, where X represents the feature columns and y represents the target column. Next, we split the data into training and testing sets, as described in subsection 3.3.1.

```
X = working_df.drop(columns=['2023 წლის ბიუჯეტი'])
Y = working_df['2023 წლის ბიუჯეტი']

X_train, X_test, y_train, y_test = train_test_split(X.values, Y.values,
    random_state=42, test_size=0.2)
```

Different Columns	Correlation
Number of Students	0.926098
Number of Classrooms	0.730389
Building Area (m^2)	0.637425
Number of Floors	0.533504

Table 3.4: Top 4 Correlation between variables and 2023 budget

3. **Performing Regression**

We try two regression models: `MultilinearRegression` 2.2.4 and `RidgeRegression` 2.2.5; then we compare results.

Linear Regression:

```
lr = LinearRegression()

lr.fit(X_train, y_train)
```

Ridge Regression:

```
ridge = Ridge(alpha=alpha)

ridge.fit(X_train, y_train)
```

The optimal value for `alpha` was derived using the following function. We took, 1000 evenly spaces values between $(10^{-4} : 10^{7})$ and ran regression over our data

```
def find_optimal_alpha(alphas, model, X_train, y_train, X_test, y_test):
    model_scores = []
    for alpha in alphas:
        model.set_params(alpha=alpha)
        model.fit(X_train, y_train)
        model_scores.append(model.score(X_test, y_test))
    # Output the optimal alpha and its corresponding score
    optimal_alpha = alphas[np.argmax(model_scores)]
    optimal_score = max(model_scores)
    return optimal_alpha, optimal_score

alpha = find_optimal_alpha(np.logspace(-4, 7, num=1000), Ridge(), X_train,
    y_train, X_test, y_test)
print(f"Optimal alpha: {alpha}")
>>>> Optimal alpha: 4684580.115873045
```

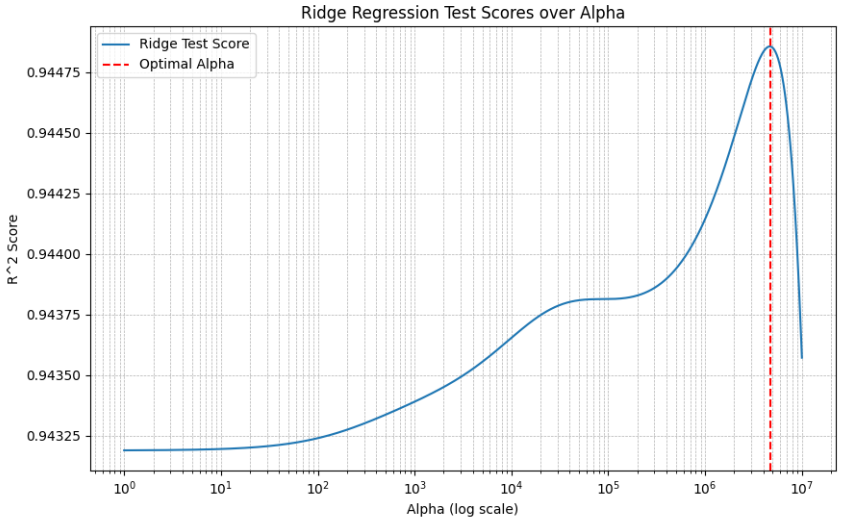


Figure 3.8: Optimal Alpha

```
# Linear Regression
print("Train score:", lr.score(X_train, y_train))
>>>> Train score: 0.8413126417293658

print("Test score:", lr.score(X_test, y_test))
>>>> Test score: 0.943189020874005

# Ridge Regression
print("Train score:", ridge.score(X_train, y_train))
>>>> Train score: 0.8397226702666185

print("Test score:", ridge.score(X_test, y_test))
>>>> Test score: 0.9448568274029476
```

4. **Predict some concrete budget**

   Given that we created a model, we can now predict some values. In what follows, we define a school with arbitrary values and predict the budget of a such school.

```
school = {
    'Number of students': 300,
    'Number of studying rooms': 15,
    'Building area in use (m²)': 2740,
    'Number of floors': 2,
}

print(ridge.predict([list(school.values())]))
>>>> [629164.75275435]

print(lr.predict([list(school.values())]))
>>>> [615784.14332664]
```

Regression with a Single Variable: Number of Students

In addition to using multiple features for regression, we also attempted running the regression using only one variable, `Number of Students`, which has the highest correlation with the `2023 Budget` (0.926098), as shown in Table 3.4.

The main difference lies in the selection of feature columns, as we are now using only the column with the strongest correlation:

```
tmp = df[cols].select_dtypes('number').corr()['2023 წლის ბიუჯეტი']
    .sort_values(ascending=False).nlargest(2)
```

With this setup, the regression models become simpler, employing only plain Linear Regression 2.2.3 and Ridge Regression 2.2.5 with a single variable. The procedure remains the same as described earlier, including splitting the data into training and testing sets, fitting the model, and evaluating its performance. The resulting scores are as follows:

```
# Linear Regression
>>>> Train score: 0.8277503287606119
>>>> Test score: 0.9493133590737497

# Ridge Regression
>>>> Train score: 0.8277503287606118
>>>> Test score: 0.9493133589933178
```

To test the model's predictions, we use the same hypothetical school setup, but this time with only `Number of Students` as the input feature:

```
school = {
    'Number of students': 300,
}

print(ridge.predict([list(school.values())]))
>>>> [617218.62034374]

print(lr.predict([list(school.values())]))
>>>> [619624.80165057]
```

As we can observe, the predictions are remarkably close to those obtained from the multi-feature model. This suggests that the `Number of Students` alone is a highly predictive feature for the `2023 Budget`, and in some cases, simpler models with fewer variables can yield comparable results. This highlights the importance of selecting features with strong correlations in regression analysis.

Regression model application on Region Values

Now we discuss one possible application of our model. We will be using a multiple-feature regression model developed in this chapter. We know that when testing, this model displayed a score of (0.9416350057351243), implying that our model's predictions are close to the true values of the data. In what follows, we divide the schools based on their region and try to predict their budget.

Our model was trained on a large portion of our data. If we try to predict only a small portion of data, there is a higher chance of error. However, if we selectively divide our data in some specific ways, this method might highlight error patterns. This hopefully will help us to gain more insight and maybe even identify some unseen variables that our model is not fit for.

First, we divide the dataset.

```
region_div = [(region_name, group) for region_name, group in df.groupby('    ')]
```

Now for each value in `region_div` we do identical decomposition into X and y and store them into a list, `X_div` and `Y_div` respectfully. Next, we score each element in `X_div` and `Y_div` with the ridge model that was defined earlier. We then print the results.

```
for i, j, k in zip(X_div, Y_div, unique):
>>>>print(f"Test score is: {ridge.score(i.values, j.values)} for the
    region {k}")
# Test score is: 0.5457454231439463 for the region [თბილისი]
# Test score is: 0.8399470512640222 for the region [სამეგრელო-ზემო სვანეთი]
# Test score is: 0.4945384527888499 for the region [რაჭა-ლეჩხუმი ქვემო სვანეთი]
# Test score is: 0.9097976715348092 for the region [იმერეთი]
# Test score is: 0.8588089535614656 for the region [კახეთი]
# Test score is: 0.5169280572048418 for the region [სამცხე-ჯავახეთი]
# Test score is: 0.9102024891529752 for the region [ქვემო ქართლი]
# Test score is: 0.9115612918865804 for the region [შიდა ქართლი]
# Test score is: 0.9470264688267866 for the region [აჭარა]
# Test score is: 0.5508406170255286 for the region [აფხაზეთი]
# Test score is: 0.8812616666031354 for the region [გურია]
# Test score is: 0.7039821866481288 for the region [მცხეთა-მთიანეთი]
```

We can see that four of the regions scored in the middle, with one of the notable outliers being `Tbilisi` (თბილისი) the capital of Georgia. There is no apparent underlying pattern between the outlier regions, other than the method by which they were divided. `Tbilisi` (თბილისი) region is located east and is one of highest in school count while `Abkhazeti` (აფხაზეთი) is located west and contains the least. `Samtskhe-Javakheti` (სამცხე-ჯავახეთი) is a southern region while `Racha-Lechkhumi and Lower Svaneti` (რაჭა-ლეჩხუმი ქვემო სვანეთი) northern region, and both containing an average number of schools. We conclude that in these four cases, our regression model explains only half of the budget. Further analysis is necessary to explain this phenomenon, which is outside of the scope of our project.

Conclusion

We have recalled some basic statistics theory and showed how to implement different statistical methods using various methods with `Python`. Along with extensive pre-processing, we applied methods recalled to the real world large data of Georgian Schools.

Future research could build on this work by incorporating longitudinal data to evaluate the long-term effects of policy changes and interventions. In addition, more advanced methodologies could be employed to refine the predictions and extend the scope of the analysis.

Bibliography

[1] C.C. Aggarwal. *Outlier Analysis*. Springer New York, 2013.

[2] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, 2016.

[3] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, May 2000.

[4] Morris H. DeGroot. *Probability and Statistics*. Pearson, 4th edition, 2012.

[5] NumPy Developers. numpy.corrcoef — numpy v2.2 manual. `https://numpy.org/doc/stable/reference/generated/numpy.corrcoef.html`. Accessed: 01.07.2025.

[6] Seaborn Developers. seaborn.heatmap — seaborn 0.12.2 documentation. `https://seaborn.pydata.org/generated/seaborn.heatmap.html`. Accessed: 01.07.2025.

[7] ESIDA. Esida. `http://esida.gov.ge/`. Accessed: 01.15.2025.

[8] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, New York, second edition, 2009. Many topics include neural networks, support vector machines, classification trees, boosting, random forests, ensemble methods, lasso, and more.

[10] M.H. Kutner. *Applied Linear Statistical Models*. McGrwa-Hill international edition. McGraw-Hill Irwin, 2005.

[11] mes. mes. `https://mes.gov.ge/`. Accessed: 01.15.2025.

[12] P. Newbold, W.L. Carlson, and B. Thorne. *Statistics for Business and Economics, Global Edition*. Pearson Education, 2019.

[13] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2006.

[14] Gael Varoquaux Olivier Grisel, Andreas Mueller. Dbscan - scikit-learn. `https://scikit-learn.org/1.5/modules/clustering.html#dbscan`. Accessed: 12.30.2024.

[15] Gael Varoquaux Olivier Grisel, Andreas Mueller. Isolationforest - scikit-learn. `https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.IsolationForest.html`. Accessed: 12.30.2024.

[16] Gael Varoquaux Olivier Grisel, Andreas Mueller. Linearregression scikit-learn. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression`. Accessed: 10.25.2024.

[17] Gael Varoquaux Olivier Grisel, Andreas Mueller. Ridge regression - scikit-learn. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html`. Accessed: 12.28.2024.

[18] Gael Varoquaux Olivier Grisel, Andreas Mueller. Tfidfvectorizer scikit-learn. `https://scikit-learn.org/1.5/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`. Accessed: 12.20.2024.

[19] pandas Developers. pandas.dataframe.corr — pandas 2.0.3 documentation. `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html`. Accessed: 01.07.2025.

[20] Juan Ramos. Using tf-idf to determine word relevance in document queries. Technical report, Department of Computer Science, Rutgers University, New Brunswick, NJ, USA, January 2003. `https://it.scribd.com/document/339224468/TF-IDF-to-determine-word-relevance-in-document-queries-pdf`.

[21] Sheldon Ross. *A First Course in Probability*. Pearson, 2019.

[22] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[23] Henri Theil. *Best Linear Unbiased Estimation and Prediction*. John Wiley & Sons, New York, 1971.

Verification of Independent Authorship

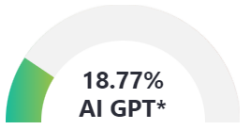
LLM models like Open AI Chat GPT and similar were used only for testing purposes, correcting notations and linguistical issues. The content itself was written without use of AI tools.

As requested, we will provide results of a detection tool:

- ZeroGPT - `https://www.zerogpt.com/`

ZeroGPT

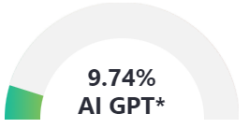
ZeroGPT was selected as our AI detection tool due to its accessibility, ease of use, and generous free token allowance. However, given the length of our thesis, which exceeded the 15,000-character limit per test, we divided the content into smaller batches to conduct a analysis. The results consistently indicated that our work was produced independently, with no significant AI involvement detected, results can be seen in Figure 3.9



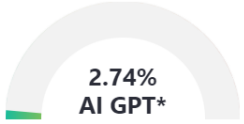
(a) ZeroGPT Batch 1

(b) ZeroGPT Batch 2

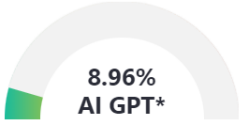
(c) ZeroGPT Batch 3



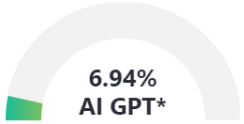
(d) ZeroGPT Batch 4



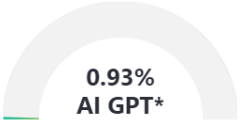
(e) ZeroGPT Batch 5



(f) ZeroGPT Batch 6



(g) ZeroGPT Batch 7



(h) ZeroGPT Batch 8

Figure 3.9: ZeroGPT results for all batches

The first two relatively high percentages are due to mathematical formulas in Chapter 2, which this detector was unable to classify accurately