NATIONAL RESEARCH UNIVERSITY ITMO

Faculty of Software Engineering and Computer Technology

**Artificial Intelligence Systems**

Laboratory Work № 1

Student

Phan Ngoc Tuan

Group № P3321

Supervisor: Yulia A.Koroleva

Contents

[**1.** **INTRODUCTION** 3](#_Toc209877925)

[**2.** **TASK STATEMENT** 3](#_Toc209877926)

[**3.** **THEORETICAL BACKGROUND** 3](#_Toc209877927)

[3.1 Linear Regression Algorithm 3](#_Toc209877928)

[3.2 Logistic Regression Algorithm 4](#_Toc209877929)

[**4.** **DATASET PREPROCESSING** 4](#_Toc209877930)

[**5.** **MODEL IMPLEMENTATION** 7](#_Toc209877931)

[**6.** **MODEL TRAINING** 12](#_Toc209877932)

[**7.** **MODEL EVALUATION** 14](#_Toc209877933)

[**8.** **CONCLUSION** 17](#_Toc209877934)

# **INTRODUCTION**

This laboratory work aims to provide a practical understanding of fundamental machine learning algorithms by implementing Linear Regression and Logistic Regression from scratch using an object-oriented programming (OOP) approach. The goal is to build, train, and evaluate these models on real-world datasets, analyze the learning process through visualization, and interpret the results

1. **TASK STATEMENT**

The following objectives were defined for this work:

1. Implement the Linear Regression and Logistic Regression algorithms from scratch using OOP principles in Python
2. Train and test the models on two datasets:
   * Linear Regression for predicting continuous values on the “California Housing” dataset
   * Logistic Regression for binary classification on the “Titanic” datasets
3. Visualize and analyze the learning process by plotting the loss reduction and metric improvement (e.g., F1-score, Accuracy, R²) over training epochs.

The implementation was allowed to use only the following libraries for auxiliary operations: NumPy, Pandas, Polars, PyTorch, Matplotlib, and Seaborn.

1. **THEORETICAL BACKGROUND**

## Linear Regression Algorithm

The goal of Linear Regression id to find a linear relationship between input features **X** and a continuous target variable **y**. The core steps of the gradient descent algorithm are as follows:

where are the weights, is the bias, is the learning rate, and is the number of samples.

## Logistic Regression Algorithm

Logistic Regression is used for binary classification. It applies a sigmoid function to the linear output to predict probabilities. The weight update rules are derived using gradient descent on the binary cross-entropy loss:

1. **DATASET PREPROCESSING**

In this section, the dataset is preprocessed to facilitate machine understanding of its contents. The main steps included:

* Checking dataset structure, missing values, duplicates.
* Visualizing distributions and correlation matrix.
* Dropping redundant columns.
* Normalizing numerical features to the range [-1, 1].

The preprocessing code is presented in Listing 1. Figures 1 and 2 illustrate the correlation matrices before and after processing.

import matplotlib.pyplot as plt  
import torch  
import pandas as pd  
import numpy as np

df\_house = pd.read\_csv('datasets/California\_Houses.csv')  
df\_house.head()

df\_house.describe()

df\_house.hist(bins=20, figsize=(20, 15))  
plt.show()

df\_titanic = pd.read\_csv('datasets/Titanic-Dataset.csv')  
df\_titanic.head()

df\_titanic.describe()

plt.style.use('default')  
  
  
def plot\_correlation\_matrix(table):  
 numeric\_cols = table.select\_dtypes(include=[np.number]).columns  
 corr\_matrix = table[numeric\_cols].corr().values  
  
 plt.figure(figsize=(14, 10))  
  
 im = plt.imshow(corr\_matrix, cmap='coolwarm', interpolation='nearest')  
 plt.colorbar(im, label='Correlation коэф.')  
  
 plt.xticks(range(len(numeric\_cols)), numeric\_cols, rotation=45, ha='right')  
 plt.yticks(range(len(numeric\_cols)), numeric\_cols)  
  
 for i in range(len(numeric\_cols)):  
 for j in range(len(numeric\_cols)):  
 plt.text(j, i, f'{corr\_matrix[i, j]:.2f}',  
 ha='center', va='center',  
 color='white' if abs(corr\_matrix[i, j]) > 0.5 else 'black')  
  
 plt.title('Correlation matrix')  
 plt.tight\_layout()  
 plt.show()

*# Check null*df\_house.isnull().any(axis=0)

df\_titanic.isnull().any(axis=0)

df\_titanic['Age'] = df\_titanic['Age'].fillna(df\_titanic['Age'].median())  
df\_titanic['Embarked'] = df\_titanic['Embarked'].ffill()  
  
selected\_cols\_titanic = df\_titanic.drop(columns=['Cabin', 'Name', 'Ticket'])

selected\_cols\_house = df\_house.drop\_duplicates()  
selected\_cols\_house.shape

plot\_correlation\_matrix(selected\_cols\_house)

selected\_cols\_house = selected\_cols\_house.drop(columns=['Tot\_Bedrooms', 'Population', 'Longitude', 'Households', 'Distance\_to\_LA', 'Distance\_to\_SanDiego', 'Distance\_to\_SanFrancisco', 'Distance\_to\_SanJose'])  
  
plot\_correlation\_matrix(selected\_cols\_house)

plot\_correlation\_matrix(selected\_cols\_titanic)

selected\_cols\_titanic = selected\_cols\_titanic.drop(  
 columns=['PassengerId', 'SibSp', 'Sex\_male', 'Embarked\_Q', 'Embarked\_S'])  
  
plot\_correlation\_matrix(selected\_cols\_titanic)

def normalize\_data(df):  
 normalized\_df = df.copy()  
  
 for col in normalized\_df.columns:  
 col\_min = df[col].min()  
 col\_max = df[col].max()  
  
 if col\_max == col\_min:  
 normalized\_df[col] = 0.0  
 else:  
 normalized\_df[col] = 2 \* (normalized\_df[col] - col\_min) / (col\_max - col\_min) - 1  
  
 return normalized\_df

Code listing 1 - preprocessing script

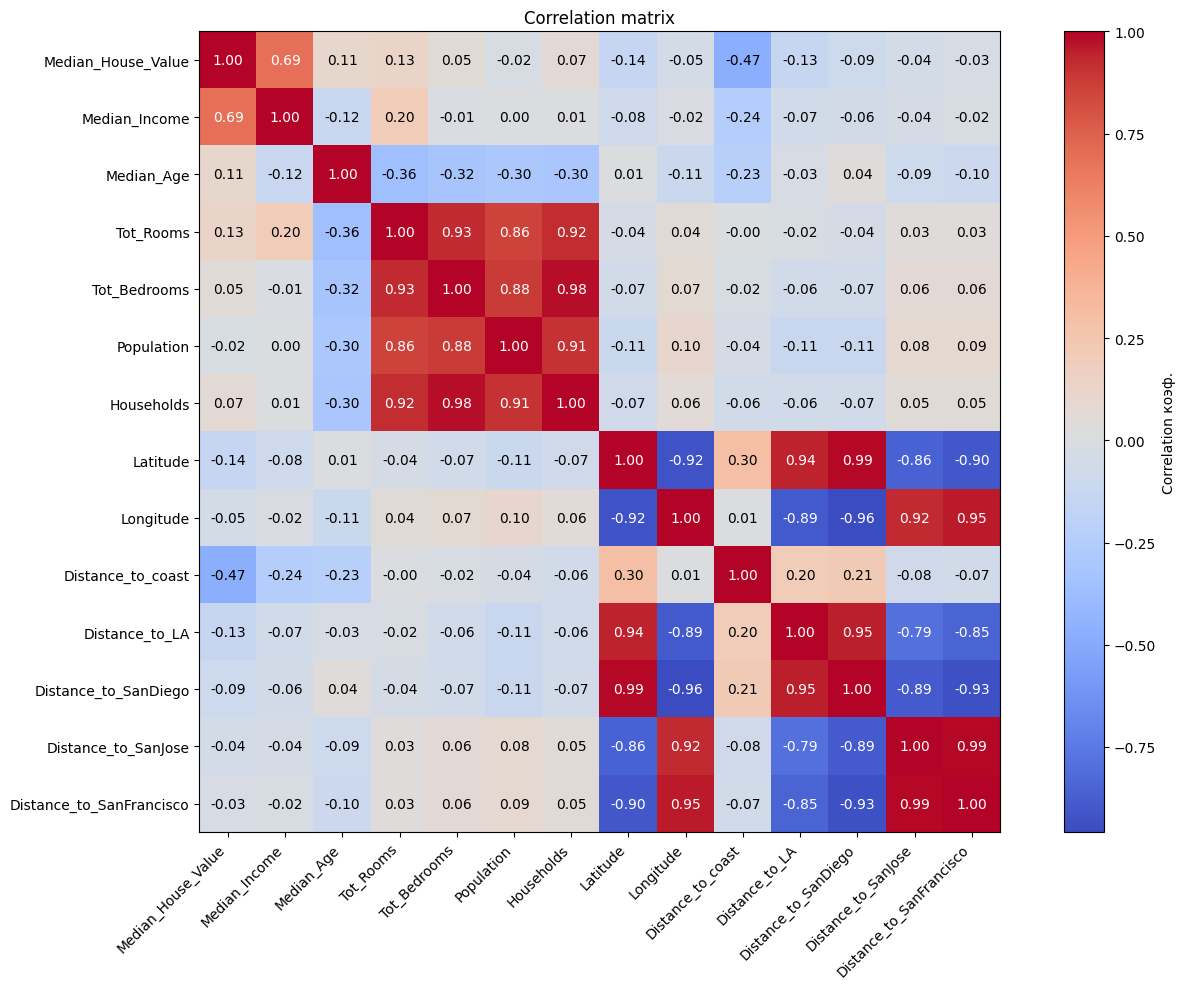


Fig. 1 – Correlation matrix before preprocessing

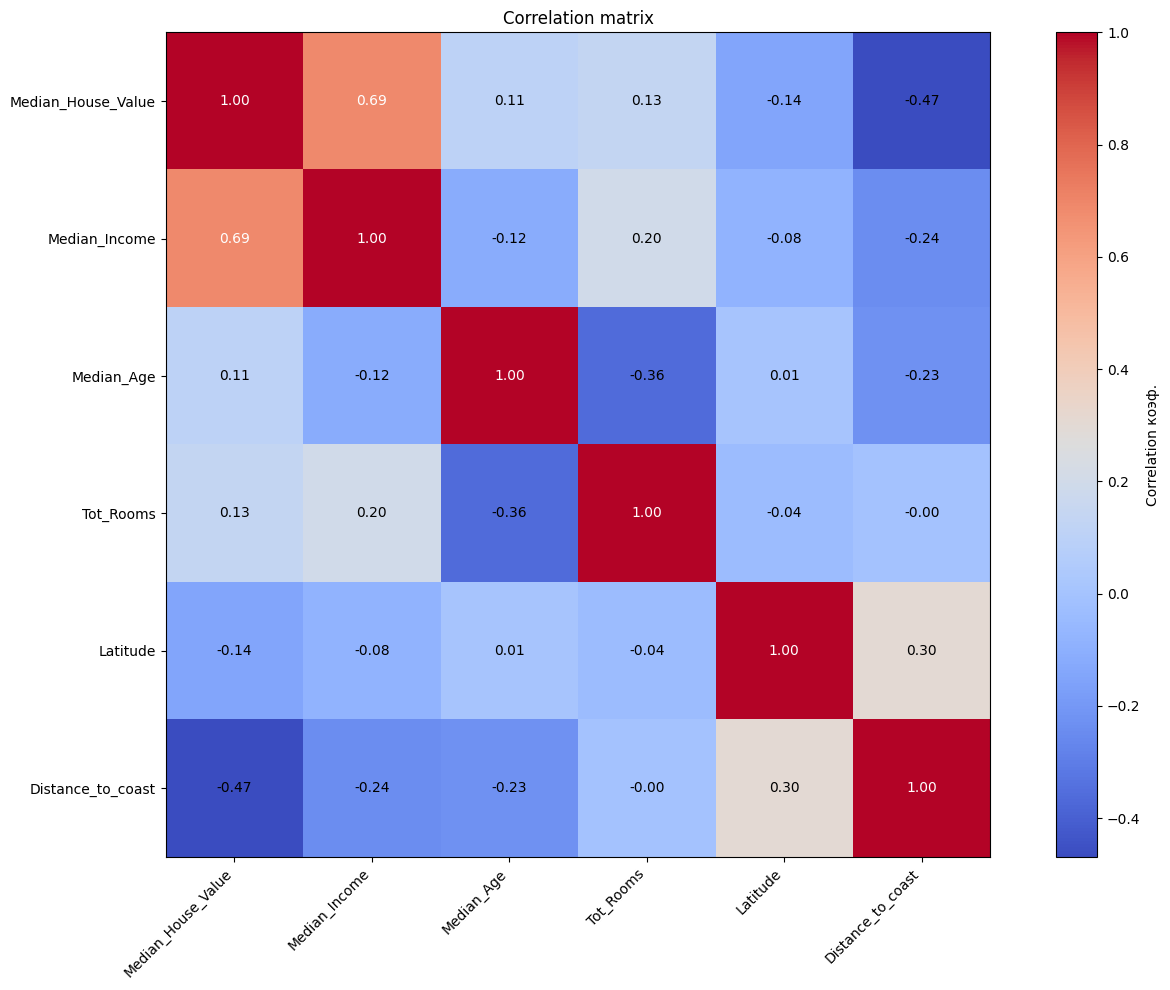
****

Fig. 2 – Correlation matrix after preprocessing

1. **MODEL IMPLEMENTATION**

Custom classes were developed for both regression models.

* Class Linear Regression – implements linear regression with metrics: MSE, RMSE, MAE, and R².
* Class Logistic Regression – implements logistic regression with metrics: Accuracy, Precision, Recall, F1-score, and binary cross-entropy loss.

Both models include methods for training (fit), prediction (predict), evaluation (evaluate), and visualization of training dynamics.

class LinearRegression:  
  
 def \_\_init\_\_(self, learning\_rate: float = 0.01, num\_iterations: int = 1000, regularization: str = None,  
 alpha: float = 0.01, l1\_ratio: float = 0.5):  
 self.learning\_rate = learning\_rate  
 self.num\_iterations = num\_iterations  
 self.train\_metrics = []  
 self.weights = None  
 self.bias = None  
 self.loss\_his = []  
 self.metric\_his = []  
 *#Lab 2 task 1* self.regularization = regularization  
 self.alpha = alpha  
 self.l1\_ratio = l1\_ratio  
  
 def calc\_metrics(self, y, y\_pred):  
 *# MSE* mse = torch.mean((y - y\_pred) \*\* 2)  
  
 *# RMSE* rmse = torch.sqrt(mse)  
  
 *# MAE* mae = torch.mean(torch.abs(y - y\_pred))  
  
 *# R²* ss\_total = torch.sum((y - torch.mean(y)) \*\* 2)  
 ss\_residual = torch.sum((y - y\_pred) \*\* 2)  
 r2 = 1 - (ss\_residual / ss\_total)  
  
 return {  
 'MSE': mse.item(),  
 'RMSE': rmse.item(),  
 'MAE': mae.item(),  
 'R2': r2.item()  
 }  
  
 def fit(self, X: torch.Tensor, y: torch.Tensor) -> list:  
 if not isinstance(X, torch.Tensor) or not isinstance(y, torch.Tensor):  
 raise ValueError("X and y must be torch. Tensor objects")  
  
 num\_samples, num\_features = X.shape  
 self.weights = torch.randn((num\_features, 1), dtype=torch.float32) \* 0.01  
 self.bias = torch.zeros(1, dtype=torch.float32)  
  
 for epoch in range(self.num\_iterations):  
 y\_pred = torch.matmul(X, self.weights) + self.bias  
  
 mse\_loss = torch.mean((y\_pred - y) \*\* 2)  
  
 *# Lab 2 task 1* reg\_loss = 0  
 if self.regularization == 'l2':  
 reg\_loss = self.alpha \* torch.sum(self.weights \*\* 2)  
 elif self.regularization == 'l1':  
 reg\_loss = self.alpha \* torch.sum(torch.abs(self.weights))  
 elif self.regularization == 'elasticnet':  
 reg\_loss = self.alpha \* (  
 self.l1\_ratio \* torch.sum(torch.abs(self.weights)) + (1 - self.l1\_ratio) \* torch.sum(  
 self.weights \*\* 2))  
  
 total\_loss = mse\_loss + reg\_loss  
  
 self.loss\_his.append(total\_loss.item())  
  
 grad\_weights = (2 / num\_samples) \* torch.matmul(X.T, (y\_pred - y))  
 grad\_bias = (2 / num\_samples) \* torch.sum(y\_pred - y)  
  
 *# Lab 2 task 1* if self.regularization == 'l2':  
 grad\_weights += 2 \* self.alpha \* self.weights  
 elif self.regularization == 'l1':  
 grad\_weights += self.alpha \* torch.sign(self.weights)  
 elif self.regularization == 'elasticnet':  
 grad\_weights += self.alpha \* (  
 self.l1\_ratio \* torch.sign(self.weights) + 2 \* (1 - self.l1\_ratio) \* self.weights)  
  
 self.weights -= self.learning\_rate \* grad\_weights  
 self.bias -= self.learning\_rate \* grad\_bias  
  
 train\_metrics = self.calc\_metrics(y, y\_pred)  
 self.metric\_his.append(train\_metrics)  
  
 if epoch % 100 == 0:  
 print(f"Epoch {epoch}: Loss={mse\_loss.item()}, R²={train\_metrics['R2']}")  
  
 def predict(self, X: torch.Tensor) -> torch.Tensor:  
 if self.weights is None or self.bias is None:  
 raise ValueError("Model must be trained before making predictions")  
 if not isinstance(X, torch.Tensor):  
 raise ValueError("Input X must be a torch.Tensor")  
  
 return torch.matmul(X, self.weights) + self.bias  
  
 def evaluate(self, X, y):  
 y\_pred = self.predict(X)  
 metrics = self.calc\_metrics(y, y\_pred)  
 return metrics, y\_pred

Code listing 2 - Class Linear Regression

class LogisticRegression:  
  
 def \_\_init\_\_(self, learning\_rate: float = 0.01, num\_iterations: int = 1000, regularization: str = None,  
 alpha: float = 0.01, l1\_ratio: float = 0.5):  
 if learning\_rate <= 0:  
 raise ValueError("Learning rate must be positive.")  
 if num\_iterations <= 0:  
 raise ValueError("Number of iterations must be positive.")  
  
 self.learning\_rate = learning\_rate  
 self.num\_iterations = num\_iterations  
 self.weights = None  
 self.bias = None  
 self.loss\_his = []  
 self.metric\_his = []  
 self.regularization = regularization  
 self.alpha = alpha  
 self.l1\_ratio = l1\_ratio  
  
 def sigmoid(self, z: torch.Tensor) -> torch.Tensor:  
 z = torch.clamp(z, -10, 10)  
 return 1 / (1 + torch.exp(-z))  
  
 def binary\_cross\_entropy(self, y\_true: torch.Tensor, y\_pred\_proba: torch.Tensor) -> torch.Tensor:  
 epsilon = 1e-8  
 y\_pred\_proba = torch.clamp(y\_pred\_proba, epsilon, 1 - epsilon)  
 return -torch.mean(y\_true \* torch.log(y\_pred\_proba) + (1 - y\_true) \* torch.log(1 - y\_pred\_proba))  
  
 def calc\_metrics(self, y: torch.Tensor, y\_pred\_proba: torch.Tensor) -> dict:  
  
 y\_pred\_class = (y\_pred\_proba > 0.5).float()  
  
 accuracy = torch.mean((y\_pred\_class == y).float())  
  
 true\_positives = torch.sum((y\_pred\_class == 1) & (y == 1))  
 predicted\_positives = torch.sum(y\_pred\_class == 1)  
 actual\_positives = torch.sum(y == 1)  
  
 precision = true\_positives / (predicted\_positives + 1e-8)  
 recall = true\_positives / (actual\_positives + 1e-8)  
  
 f1 = 2 \* (precision \* recall) / (precision + recall + 1e-8)  
  
 loss = self.binary\_cross\_entropy(y, y\_pred\_proba)  
  
 return {  
 'accuracy': accuracy.item(),  
 'precision': precision.item(),  
 'recall': recall.item(),  
 'f1': f1.item(),  
 'loss': loss.item()  
 }  
  
 def fit(self, X: torch.Tensor, y: torch.Tensor) -> None:  
 if not isinstance(X, torch.Tensor) or not isinstance(y, torch.Tensor):  
 raise ValueError("Inputs X and y must be torch.Tensor objects.")  
 if X.ndim != 2 or y.ndim != 2:  
 raise ValueError("X must be 2D and y must be 2D (num\_samples, 1).")  
 if X.shape[0] != y.shape[0]:  
 raise ValueError("Number of samples in X and y must match.")  
 if y.shape[1] != 1:  
 raise ValueError("y must have shape (num\_samples, 1).")  
  
 num\_samples, num\_features = X.shape  
  
 self.weights = torch.randn((num\_features, 1), dtype=torch.float32) \* 0.01  
 self.bias = torch.zeros(1, dtype=torch.float32)  
  
 for epoch in range(self.num\_iterations):  
 linear\_output = torch.matmul(X, self.weights) + self.bias  
 y\_pred\_proba = self.sigmoid(linear\_output)  
  
 loss = self.binary\_cross\_entropy(y, y\_pred\_proba)  
  
 *# Lab 2 task 1* reg\_loss = 0  
  
 if self.regularization == 'l2':  
 reg\_loss = self.alpha \* torch.sum(self.weights \*\* 2)  
 elif self.regularization == 'l1':  
 reg\_loss = self.alpha \* torch.sum(torch.abs(self.weights))  
 elif self.regularization == 'elasticnet':  
 reg\_loss = self.alpha \* (  
 self.l1\_ratio \* torch.sum(torch.abs(self.weights)) + (1 - self.l1\_ratio) \* torch.sum(  
 self.weights \*\* 2))  
  
 total\_loss = loss + reg\_loss  
 self.loss\_his.append(total\_loss.item())  
  
 error = y\_pred\_proba - y  
 dw = torch.matmul(X.T, error) / num\_samples  
 db = torch.sum(error) / num\_samples  
  
 if self.regularization == 'l2':  
 dw += 2 \* self.alpha \* self.weights  
 elif self.regularization == 'l1':  
 dw += self.alpha \* torch.sign(self.weights)  
 elif self.regularization == 'elasticnet':  
 dw += self.alpha \* (  
 self.l1\_ratio \* torch.sign(self.weights) + 2 \* (1 - self.l1\_ratio) \* self.weights  
 )  
  
 self.weights -= self.learning\_rate \* dw  
 self.bias -= self.learning\_rate \* db  
  
 if epoch % 10 == 0:  
 metrics = self.calc\_metrics(y, y\_pred\_proba)  
 self.metric\_his.append(metrics)  
  
 if epoch % 100 == 0:  
 print(f"Epoch {epoch}: Loss={loss.item()}, "  
 f"Accuracy={metrics['accuracy']}, F1={metrics['f1']}")  
  
 def predict\_proba(self, X: torch.Tensor) -> torch.Tensor:  
 if self.weights is None or self.bias is None:  
 raise ValueError("Model must be trained before prediction.")  
 if X.shape[1] != self.weights.shape[0]:  
 raise ValueError("Input feature dimension must match model weights.")  
  
 return self.sigmoid(torch.matmul(X, self.weights) + self.bias)  
  
 def predict(self, X: torch.Tensor, threshold: float = 0.5) -> torch.Tensor:  
 probabilities = self.predict\_proba(X)  
 return (probabilities > threshold).float()  
  
 def evaluate(self, X: torch.Tensor, y: torch.Tensor) -> dict:  
 if y.shape[1] != 1:  
 raise ValueError("y must have shape (num\_samples, 1).")  
 y\_pred\_proba = self.predict\_proba(X)  
 return self.calc\_metrics(y, y\_pred\_proba)  
  
 def plot\_f1\_accuracy(self):  
 f1\_scores = [metrics['f1'] for metrics in self.metric\_his]  
 accuracies = [metrics['accuracy'] for metrics in self.metric\_his]  
  
 epochs = [i \* 100 for i in range(len(self.metric\_his))]  
  
 plt.figure(figsize=(10, 6))  
  
 plt.plot(epochs, f1\_scores, 'blue', label='F1 Score')  
 plt.plot(epochs, accuracies, 'red', label='Accuracy')  
  
 plt.xlabel('Epochs')  
 plt.ylabel('Value')  
 plt.title('F1 and Accuracy')  
 plt.legend()  
 plt.grid(alpha=0.3)  
 plt.ylim(0, 1.05)  
 plt.show()

Code listing 3 - Class Logistic Regression

1. **MODEL TRAINING**

For the California housing dataset, the linear regression model was trained.

X\_house = pd.read\_csv('in\_house.csv').values  
y\_house = pd.read\_csv('out\_house.csv').values  
  
X\_house = torch.tensor(X\_house, dtype=torch.float32)  
y\_house = torch.tensor(y\_house, dtype=torch.float32)  
  
train\_size = int(0.8 \* len(X\_house))  
X\_h\_train, X\_h\_test = X\_house[:train\_size], X\_house[train\_size:]  
y\_h\_train, y\_h\_test = y\_house[:train\_size], y\_house[train\_size:]  
  
*# Lab 2 task 1*reg\_types = [None, 'l1', 'l2', 'elasticnet']  
labels = ['No Reg', 'L1', 'L2', 'ElasticNet']  
losses\_all = []  
  
for reg in reg\_types:  
 model = LinearRegression(learning\_rate=0.01, num\_iterations=1000, regularization=reg, alpha=0.05, l1\_ratio=0.5)  
 model.fit(X\_h\_train, y\_h\_train)  
 losses\_all.append(model.loss\_his)

Code listing 4 - Linear Regression Model Training

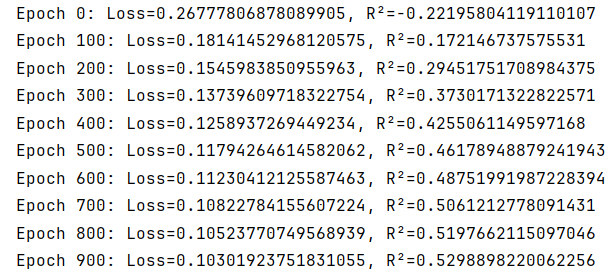


Fig. 3 – Loss and R² every 100 epochs.

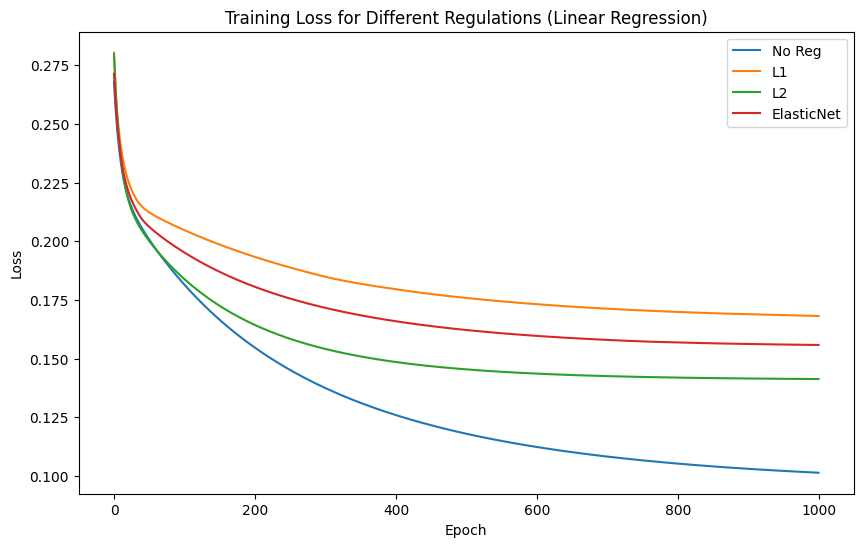


Fig. 4 – Error decrease during training (blue line).

For the Titanic dataset, the logistic regression model was trained.

X\_titanic = pd.read\_csv('in\_titanic.csv').values  
y\_titanic = pd.read\_csv('out\_titanic.csv').values  
  
X\_titanic = torch.tensor(X\_titanic, dtype=torch.float32)  
y\_titanic = torch.tensor(y\_titanic, dtype=torch.float32)  
  
train\_size = int(0.8 \* len(X\_titanic))  
X\_t\_train, X\_t\_test = X\_titanic[:train\_size], X\_titanic[train\_size:]  
y\_t\_train, y\_t\_test = y\_titanic[:train\_size], y\_titanic[train\_size:]  
  
reg\_types = [None, 'l1', 'l2', 'elasticnet']  
labels = ['No Reg', 'L1', 'L2', 'ElasticNet']  
losses\_all = []  
  
for reg in reg\_types:  
 model = LogisticRegression(learning\_rate=0.01, num\_iterations=1000,  
 regularization=reg, alpha=0.05, l1\_ratio=0.5)  
 model.fit(X\_t\_train, y\_t\_train)  
 losses\_all.append(model.loss\_his)

Code listing 5 - Logistic Regression Model Training

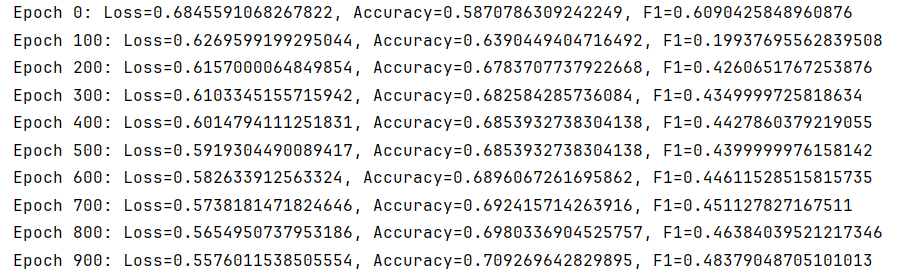


Fig. 5 – Loss, Accuracy and F1 every 100 epochs.

1. **MODEL EVALUATION**

Model evaluation results are presented in Listing 5.

Metrics on training and testing datasets:

* Mean Squared Error (MSE).
* Root Mean Squared Error (RMSE).
* Mean Absolute Error (MAE).
* R² score.

metrics\_train, y\_train\_pred = model.evaluate(X\_h\_train, y\_h\_train)  
  
metrics\_test, y\_test\_pred = model.evaluate(X\_h\_test, y\_h\_test)  
  
print("Training Metrics:")  
print(f"MSE: {metrics\_train['MSE']}")  
print(f"RMSE: {metrics\_train['RMSE']}")  
print(f"MAE: {metrics\_train['MAE']}")  
print(f"R²: {metrics\_train['R2']}")  
  
print("\nTest Metrics:")  
print(f"MSE: {metrics\_test['MSE']}")  
print(f"RMSE: {metrics\_test['RMSE']}")  
print(f"MAE: {metrics\_test['MAE']}")  
print(f"R²: {metrics\_test['R2']}")

Code listing 6 - Model evaluation

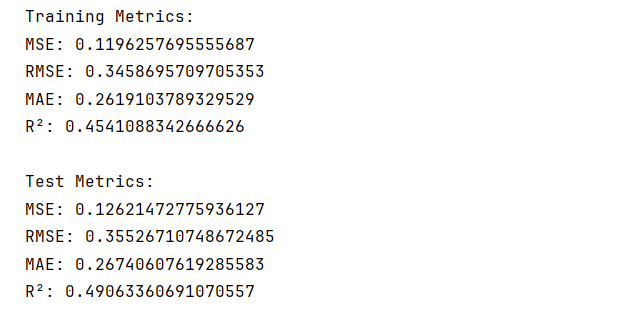


Fig. 6 – MSE, RMSE, MAE, R2

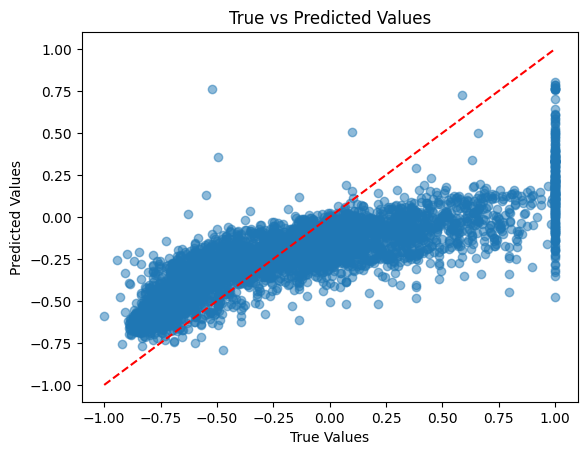


Fig. 7 – MSE, RMSE, MAE, R2

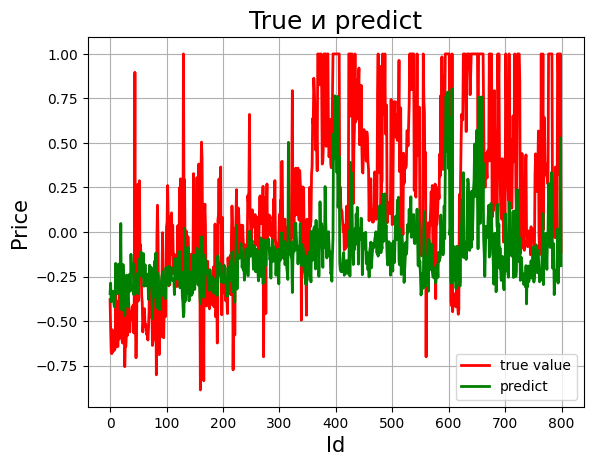


Fig. 8 – Real vs predicted comparison

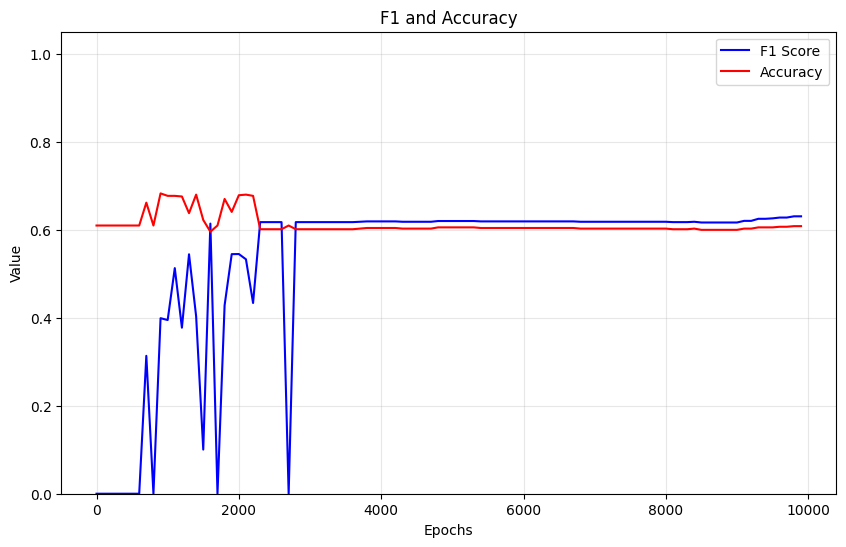
****

Fig. 9 – Logistic regression F1 and accuracy

1. **CONCLUSION**

This laboratory work allowed the student to study the stages of machine learning model training, understand the role of evaluation metrics, and practice preprocessing of datasets. The implementation of regression algorithms from scratch deepened the understanding of optimization methods and metric interpretation.