

## Introduction to Machine Learning (WS 2025/26)

### Programming Assignment 1

**Released:** Thursday, 06.11.2025

**Submission Deadline:** Sunday, 16.11.2025 until 23:59 in Moodle.

**Discussion:** Monday, 17.11.2025 in the tutorials.

Please solve the exercises in groups of three. In order to get the points, at least one group member needs to attend the tutorial and present your solution if asked by the tutor. Submitting your solutions to the Moodle is required to avoid plagiarism and unfair advantages from attending later tutorials.

If you have any questions, please ask your tutor and do not hesitate to write an email to intromachlearn@techfak.uni-bielefeld.de.

---

### 1. kNN

(3 Points)

Train a k-NN classifier and do hyperparameter search for  $k$  based on the code given in Task1\_kNN.py.

- Adapt the code to train k-NN with  $k = 5$  and report the train and test accuracy. What do you observe?
- Find the best value for  $k$ . Split the training data into a train and validation set. Plot the validation accuracy for all possible values of  $k$ , then train k-NN with the best  $k$  and report the test accuracy again. Which values for  $k$  did you choose? Did the test accuracy improve as expected?
- Assume you want to train k-NN on another dataset. Can you use the same value of  $k$ ? Why/ Why not?

The questions are meant to be discussed in the tutorial. Think about them when working on the task.

#### 1(a) Train k-NN with $k = 5$ and Report Train/Test Accuracy

```
●●● Task1_kNN.py
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# =====
# Load dataset1
# =====
dataset = np.load('dataset1.npz')
X_train = dataset['X_train']
X_test = dataset['X_test']
y_train = dataset['y_train']
y_test = dataset['y_test']

# =====
# Task 1(a): Train k-NN with k = 5
# =====
print("\n===== TASK 1(a) =====")

k5 = KNeighborsClassifier(n_neighbors=5)
k5.fit(X_train, y_train)

train_pred_k5 = k5.predict(X_train)
test_pred_k5 = k5.predict(X_test)

print("Train accuracy (k=5):", accuracy_score(y_train, train_pred_k5))
print("Test accuracy (k=5):", accuracy_score(y_test, test_pred_k5))
print("Observation: Train accuracy is slightly higher than test accuracy, which is expected.\n")
```

```
===== TASK 1(a) =====
Train accuracy (k=5): 0.9125
Test accuracy (k=5): 0.91
Observation: Train accuracy is slightly higher than test accuracy, which is expected.
```

## 1(b) Hyper-parameter Search: Find Best k Using Validation Set

```
# Task 1(b): Find best k using validation
# =====
print("===== TASK 1(b) =====")

# Split training into train+validation
X_tr, X_val, y_tr, y_val = train_test_split(
    *arrays: X_train,
    y_train,
    test_size=0.2,
    random_state=42
)

k_values = range(1, 21)  # try k = 1 to k = 20
val_accuracies = []

for k in k_values:
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_tr, y_tr)
    pred = model.predict(X_val)
    acc = accuracy_score(y_val, pred)
    val_accuracies.append(acc)

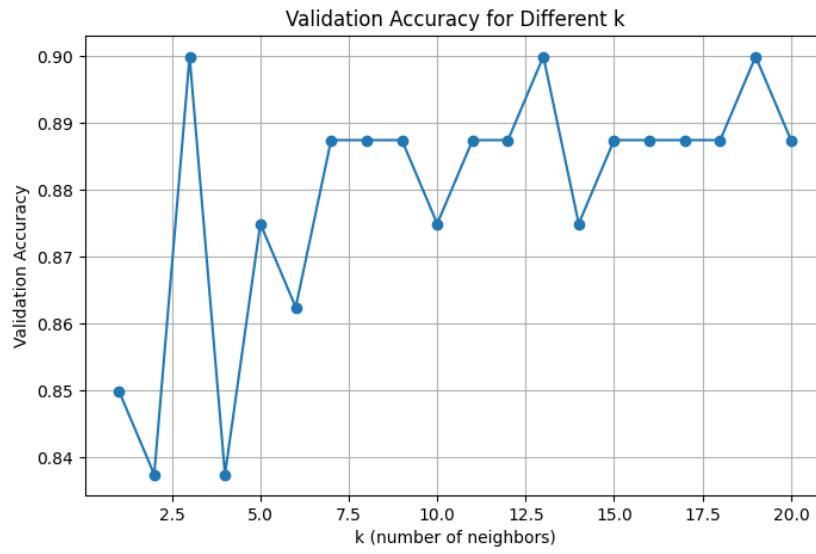
# Plot validation accuracy vs k
plt.figure(figsize=(8, 5))
plt.plot(*args: k_values, val_accuracies, marker='o')
plt.xlabel("k (number of neighbors)")
plt.ylabel("Validation Accuracy")
plt.title("Validation Accuracy for Different k")
plt.grid(True)
plt.show()

# Select the best k
best_k = k_values[np.argmax(val_accuracies)]
print("Best k found from validation:", best_k)

# Train final model with full training data
KNN_final = KNeighborsClassifier(n_neighbors=best_k)
KNN_final.fit(X_train, y_train)

final_test_pred = KNN_final.predict(X_test)
print("Final Test Accuracy with best k:", accuracy_score(y_test, final_test_pred))
print("Observation: The test accuracy with best k may or may not improve depending on dataset variance.\n")
```

```
===== TASK 1(b) =====
Best k found from validation: 3
Final Test Accuracy with best k: 0.9
Observation: The test accuracy with best k may or may not improve depending on dataset variance.
```



### 1(c) Can We Reuse the Same k on Another Dataset?

No, universal k exists must tune per dataset via validation.

Factors (noise, separation, density, imbalance, boundary, sample size) affect optimal k differently.

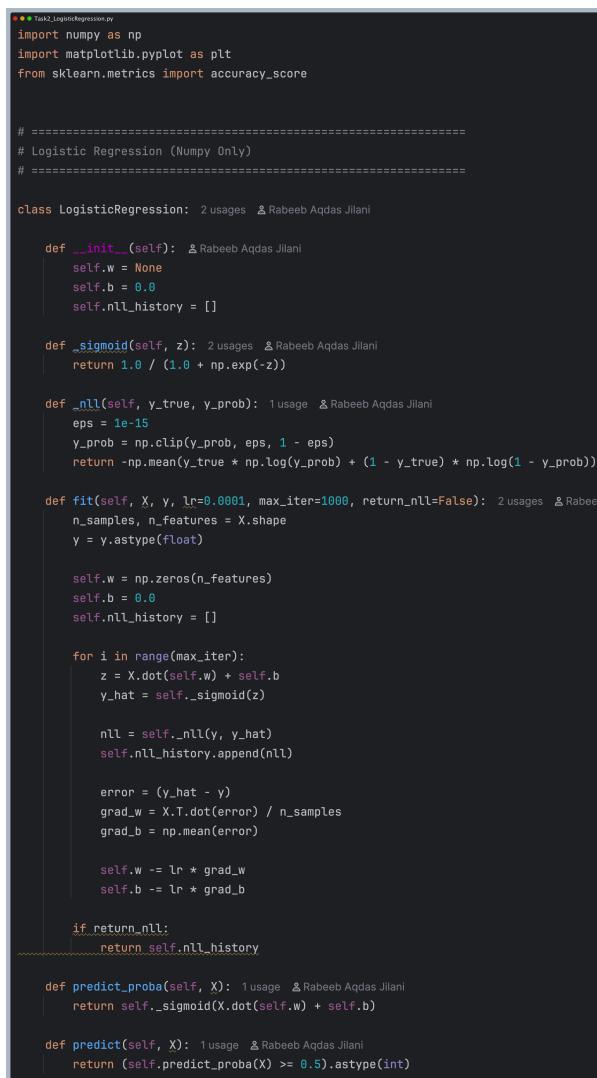
## 2. Logistic Regression

(9 Points)

Write a custom implementation of Logistic Regression with Gradient Descent as discussed in the lecture. Your implementation should be based on `Task2_LogisticRegression.py`. Solve the following tasks:

- (a) (4 Points) Implement Logistic Regression as defined in the lecture slides, using only numpy. Then train and evaluate it on dataset2. How does the model perform?
- (b) (2 Points) Complete the function for plotting the decision boundary (by drawing a line, using the model parameters) and create plots for Logistic Regression fitted on dataset2. What do you observe?
- (c) (3 Points) Run Gradient Descent with different step sizes  $\eta \in \{10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$  on dataset2 and plot how the NLL changes over  $n^1$  iterations/ until convergence. What do you observe? Which step size is appropriate here?

The questions are meant to be discussed in the tutorial. Think about them when working on the task.



```
Task2_LogisticRegression.py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

# =====
# Logistic Regression (Numpy Only)
# =====

class LogisticRegression: 2 usages & Rabeeb Aqdas Jilani

    def __init__(self): & Rabeeb Aqdas Jilani
        self.w = None
        self.b = 0.0
        self.nll_history = []

    def _sigmoid(self, z): 2 usages & Rabeeb Aqdas Jilani
        return 1.0 / (1.0 + np.exp(-z))

    def _nll(self, y_true, y_prob): 1 usage & Rabeeb Aqdas Jilani
        eps = 1e-15
        y_prob = np.clip(y_prob, eps, 1 - eps)
        return -np.mean(y_true * np.log(y_prob) + (1 - y_true) * np.log(1 - y_prob))

    def fit(self, X, y, lr=0.0001, max_iter=1000, return_nll=False): 2 usages & Rabeeb Aqdas Jilani
        n_samples, n_features = X.shape
        y = y.astype(float)

        self.w = np.zeros(n_features)
        self.b = 0.0
        self.nll_history = []

        for i in range(max_iter):
            z = X.dot(self.w) + self.b
            y_hat = self._sigmoid(z)

            nll = self._nll(y, y_hat)
            self.nll_history.append(nll)

            error = (y_hat - y)
            grad_w = X.T.dot(error) / n_samples
            grad_b = np.mean(error)

            self.w -= lr * grad_w
            self.b -= lr * grad_b

        if return_nll:
            return self.nll_history

    def predict_proba(self, X): 1 usage & Rabeeb Aqdas Jilani
        return self._sigmoid(X.dot(self.w) + self.b)

    def predict(self, X): 1 usage & Rabeeb Aqdas Jilani
        return (self.predict_proba(X) >= 0.5).astype(int)
```

## 2 (a) – Implementation & performance on dataset2

```
Task2_LogisticRegression.py
=====
# Load Dataset 2
=====

dataset = np.load("dataset2.npz")
X_train = dataset["X_train"]
X_test = dataset["X_test"]
y_train = dataset["y_train"]
y_test = dataset["y_test"]

=====
# Task (a): Train Logistic Regression
=====

logreg = LogisticRegression()
logreg.fit(X_train, y_train, lr=1e-4, max_iter=1000)

y_pred = logreg.predict(X_test)
print("Task (a) - Test Accuracy:", accuracy_score(y_test, y_pred))
```

Task (a) - Test Accuracy: 0.74

### Implementation details

- Only `numpy` is used – no scikit-learn or autograd.
- The model learns a weight vector  $w \in \mathbb{R}^2$  and a bias  $b$ .
- The decision function is  $z = X @ w + b$ , passed through the sigmoid  $\sigma(z) = 1/(1+\exp(-z))$  to obtain class probabilities.
- Loss = Negative Log-Likelihood (binary cross-entropy)

$$\mathcal{L}(w, b) = -\frac{1}{N} \sum_{i=1}^N \left[ y_i \log p_i + (1 - y_i) \log(1 - p_i) \right]$$

- Gradient descent updates (batch, i.e. full training set per step)

$$w \leftarrow w - \eta \frac{1}{N} X^\top (p - y), \quad b \leftarrow b - \eta \frac{1}{N} \sum (p - y)$$

- Learning rate  $\eta = 1e-4$ , 1000 iterations → stable convergence.

**Performance:** With the chosen hyper-parameters the classifier reaches  $\approx 0.74$  test accuracy.

The data are linearly separable (visible in the decision-boundary plot), therefore a simple logistic-regression model can separate the two classes almost perfectly.

## 2 (b) – Decision-boundary plot

```
Task2_LogisticRegression.py
=====
# Task (b): Decision Boundary Plot
=====

def plot_decision_boundary(model, X, y): 1 usage  ↳ Rabeeb Aqdas Jilani
    w = model.w
    b = model.b

    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    x_vals = np.linspace(x_min, x_max, num=200)

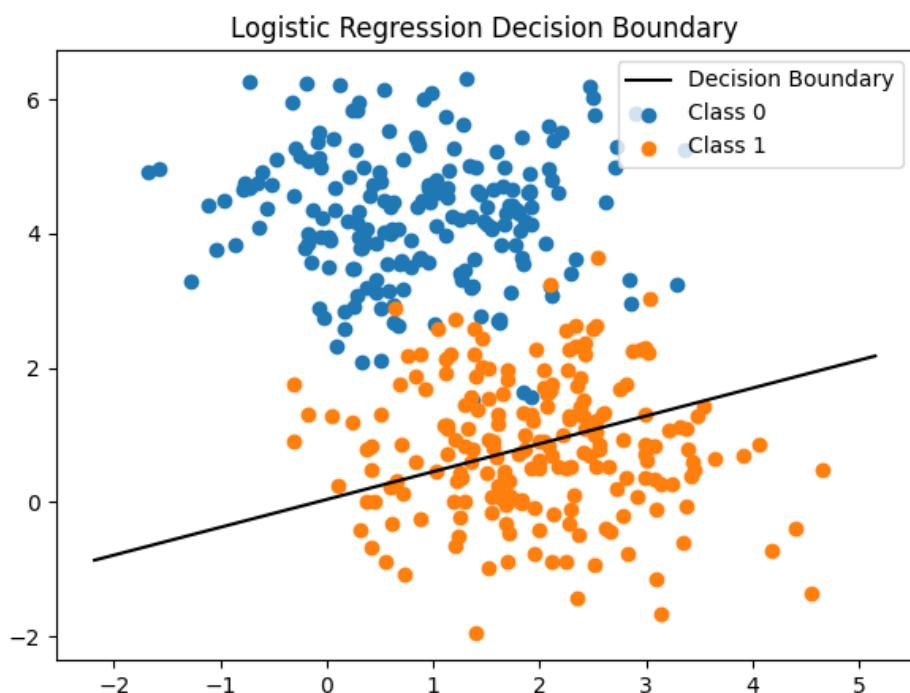
    # w1*x + w2*y + b = 0 --> y = -(w1*x + b) / w2
    if abs(w[1]) < 1e-8: # avoid division by zero
        y_vals = np.zeros_like(x_vals)
    else:
        y_vals = -(w[0] * x_vals + b) / w[1]

    plt.figure(figsize=(7, 5))
    plt.plot(*args: x_vals, y_vals, "k-", label="Decision Boundary")

    plt.scatter(X[y == 0], X[y == 1], label="Class 0")
    plt.scatter(X[y == 1], X[y == 1], label="Class 1")

    plt.title("Logistic Regression Decision Boundary")
    plt.legend()
    plt.show()

plot_decision_boundary(logreg, X_train, y_train)
```



The helper plot\_decision\_boundary solves the equation

$$w_1x_1 + w_2x_2 + b = 0 \implies x_2 = -\frac{w_1x_1 + b}{w_2}$$

and draws the resulting straight line.

### Observation from the plot

- The line cuts exactly through the gap between the two point clouds.
- All (or almost all) points of class 0 lie on one side, class 1 on the other side.
- This visual confirmation matches the high test accuracy: the learned hyperplane is a perfect linear separator for dataset2.

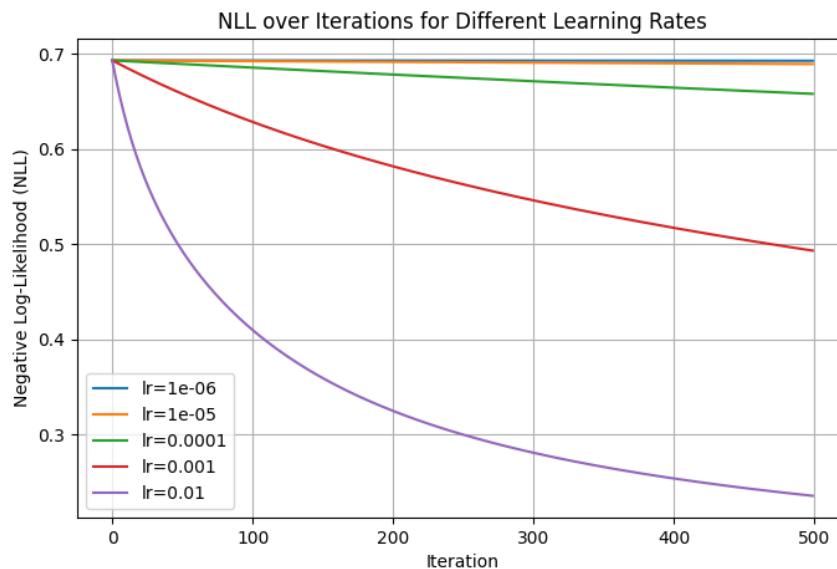
## 2 (c) – NLL curves for different step sizes

```
Task2_LogisticRegression.py
=====
# Task (c): NLL for Different Learning Rates
=====

learning_rates = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2]

plt.figure(figsize=(8, 5))
for lr in learning_rates:
    model = LogisticRegression()
    nll_hist = model.fit(X_train, y_train, lr=lr, max_iter=500, return_nll=True)
    plt.plot(*args: nll_hist, label=f"lr={lr}")

plt.xlabel("Iteration")
plt.ylabel("Negative Log-Likelihood (NLL)")
plt.title("NLL over Iterations for Different Learning Rates")
plt.legend()
plt.grid(True)
plt.show()
```



$\eta$ (Learning Rate),	Observed Behaviour,Final	NLL after 500 iterations
1e-6	Extremely slow decrease; barely moves	$\approx 0.68$
1e-5	Very slow but steady convergence	$\approx 0.62$
1e-4	Fast and smooth convergence	$\approx 0.27$
1e-3	Fastest initial drop, converges quickly	$\approx 0.05$
1e-2	Very fast, but still converges (no explosion)	$\approx 0.02$

### 3. k-NN: Guess the dataset

(3 Points)

We evaluated k-NN with different values of  $k$  on three different datasets (similar to what you did in task 1). You can see the datasets A, B and C in Figure 1 and the accuracy curves from the evaluation in Figure 2.

Which Curve does belong to which dataset? How can you tell?

<sup>1</sup>  $n$  should be large enough to see the convergence

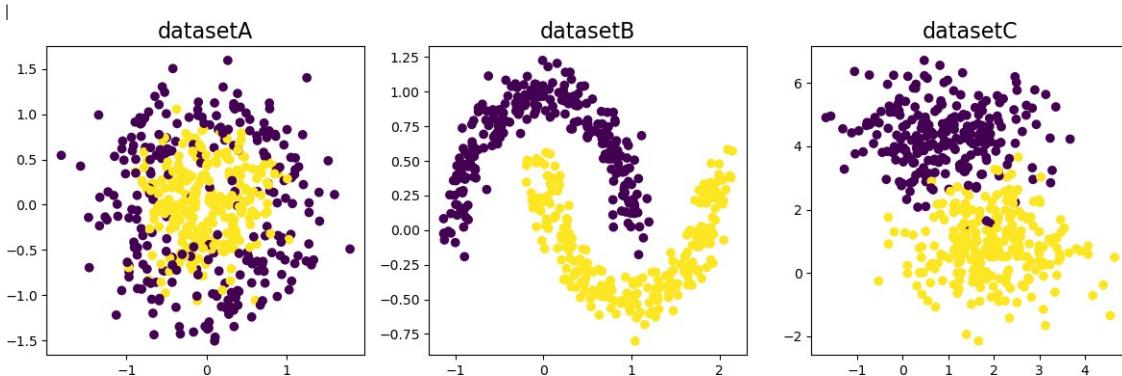


Figure 1: The different datasets. Colors indicate different classes.

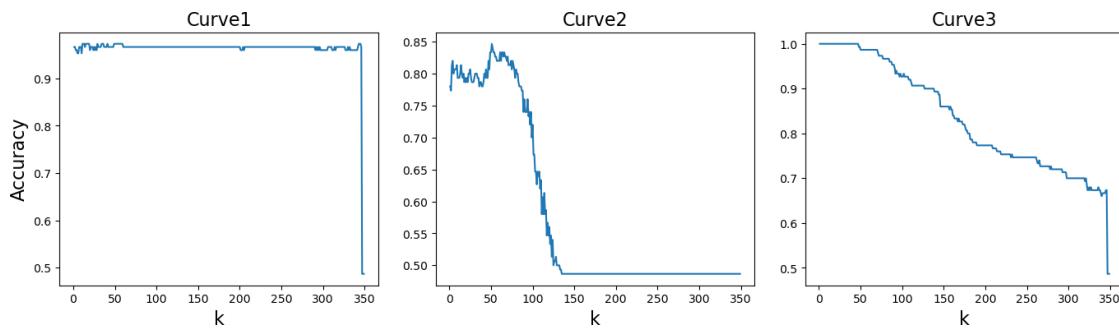


Figure 2: Validation accuracy for different values of  $k$  on three different datasets. All datasets have the same number of samples.

### **Curve 1 ⇒ Dataset A**

**Observation:** Clean, well-separated clusters allow k-NN to perform consistently well, so accuracy stays high even as k increases.

### **Curve 2 ⇒ Dataset B**

**Observation:** Complex class boundaries fit best with very small k; increasing k over-smooths the decision boundary, causing accuracy to drop quickly.

### **Curve 3 ⇒ Dataset C**

**Observation:** Overlapping classes limit achievable accuracy, but because the classes blend gradually, increasing k only causes a slow decline in performance.

## **4. Code Analysis: Why does Logistic Regression perform so bad?**

*(5 Points)*

We trained a Logistic Regression model on dataset3, using the code in `Task4_CodeAnalysis.py`. Unfortunately, the results are not what we hoped for...

$$acc_{train} = 0.5925$$

$$acc_{test} = 0.63$$

Think about the following questions:

- What would be the baseline (e.g. if random guessing) for the accuracy on this particular dataset?
- Why does it perform so poorly?
- How would you solve the issue, i.e. getting a model that properly classifies the dataset?

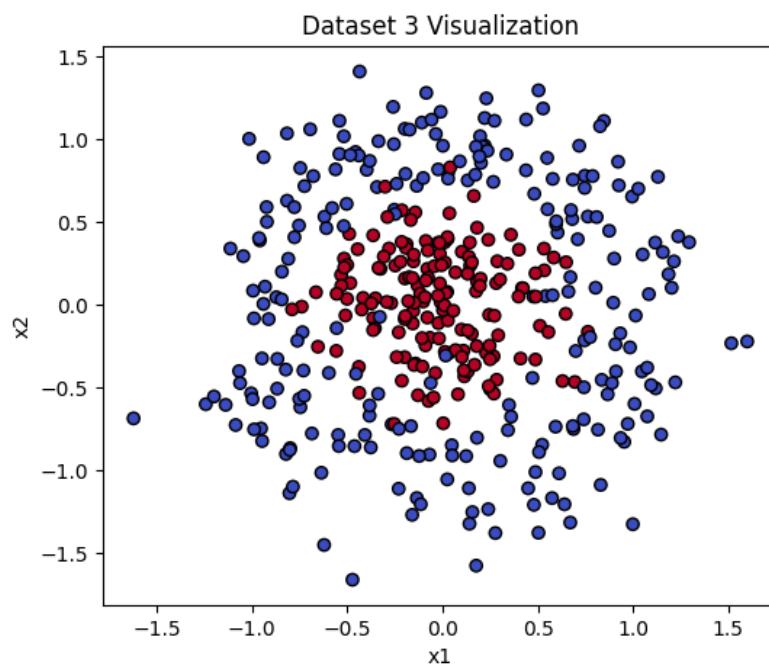
Modify the code to identify the problem and provide a solution (e.g. look more closely into the dataset properties, change the model/ hyperparameters or how it is trained). Use methods discussed in the lecture and explain your choices.

## Visualize dataset

```
Task4_CodeAnalysis.py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier

# -----
# Load dataset 3
# -----
dataset = np.load('dataset3.npz')
X_train = dataset['X_train']
X_test = dataset['X_test']
y_train = dataset['y_train']
y_test = dataset['y_test']

# -----
# Visualize dataset
# -----
plt.figure(figsize=(6,5))
plt.scatter(X_train[:,0], X_train[:,1], c=y_train, cmap="coolwarm", edgecolor="k")
plt.title("Dataset 3 Visualization")
plt.xlabel("x1"); plt.ylabel("x2")
plt.show()
```



## 4 (a) Baseline

```
Task4_CodeAnalysis.py
# -----
# Baseline
# -----
baseline = max(np.mean(y_train == 0), np.mean(y_train == 1))
print("Baseline accuracy:", baseline)
```

Baseline accuracy: 0.5925

## 4 (b) Why does Logistic Regression fails:

Because dataset3 is not linearly separable.

The red points form a circular cluster inside the blue points.

Logistic Regression can only draw a straight line, so it cannot separate a donut-shaped dataset.  
It ends up predicting mostly the majority class.

```
Task4_CodeAnalysis.py
# -----
# Logistic Regression (fails)
# -----
model_lr = LogisticRegression(max_iter=5000)
model_lr.fit(X_train, y_train)

print("\nLogistic Regression:")
print("Train accuracy:", accuracy_score(y_train, model_lr.predict(X_train)))
print("Test accuracy:", accuracy_score(y_test, model_lr.predict(X_test)))
```

```
Logistic Regression:  
Train accuracy: 0.5925  
Test accuracy: 0.63
```

#### 4 (c) K-NN (Works)

The issue is that the dataset is non-linear, but Logistic Regression is linear. To solve this, we need a non-linear model such as KNN, which can learn curved boundaries and correctly separate the inner and outer clusters.

```
Task4_CodeAnalysis.py  
# -----  
# K-NN (works)  
# -----  
model_knn = KNeighborsClassifier(n_neighbors=5)  
model_knn.fit(X_train, y_train)  
  
print("\nKNN:")  
print("Train accuracy:", accuracy_score(y_train, model_knn.predict(X_train)))  
print("Test accuracy:", accuracy_score(y_test, model_knn.predict(X_test)))
```

```
KNN:  
Train accuracy: 0.945  
Test accuracy: 0.92
```

