Entrega 7: Detección de spam

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

1. Apartado A

Siguiendo las instrucciones del enunciado, el código queda tal que:

```python
import sklearn.svm as svm
import scipy.io as sio
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import os
from utils_p7 import email2TokenList, getVocabDict
import codecs
import SVM_Trainer
import Logic_Regression_Trainer
import nn_trainer
import pytorch_trainer
import Poly_trainer
import torch.nn as nn
import torch.optim as optim
import torch
from sklearn.metrics import accuracy_score
from pytorch_trainer import ComplexModel, train_data, train_model, pred_check, device

plot_folder: str = 'memoria/images'


def load_data(file: str) -> tuple[np.ndarray, np.ndarray]:
    """Loads the data from a .mat file

    Args:
        file (str): name of the file

    Returns:
        tuple[np.ndarray, np.ndarray]: X and y data
    """
    data = sio.loadmat(file)
    X = data['X']
    y = data['y']
    return X, y


def load_data3(file: str) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """Loads the data from a .mat file

    Args:
        file (str): name of the file

    Returns:
        tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]: X, y, Xval, yval data
    """
    data = sio.loadmat(file)
    X = data['X']
    y = data['y']
    Xval = data['Xval']
    yval = data['yval']
    return X, y, Xval, yval


def kernel_linear(X: np.ndarray, y: np.ndarray, C: float) -> None:
    """Linear kernel

    Args:
        X (np.ndarray): X train dataa
        y (np.ndarray): y train data
        C (float): regularization parameter
    """
    svm_lineal: svm.SVC = svm.SVC(kernel='linear',  C=C)
    svm_lineal.fit(X, y.ravel())
    x1: np.ndarray = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
```

```python
    x2: np.ndarray = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
    X1, X2 = np.meshgrid(x1, x2)
    yp: np.ndarray = svm_lineal.predict(
        np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape)
    plt.contour(X1, X2, yp, colors='darkgreen', linewidths=1)
    plt.scatter(X[y.ravel() == 1, 0], X[y.ravel() == 1, 1], c='b', marker='x')
    plt.scatter(X[y.ravel() == 0, 0], X[y.ravel() == 0, 1], c='y', marker='o')
    plt.xticks(np.arange(0, 5.5, 0.5))
    plt.yticks(np.arange(1.5, 5.5, 0.5))
    plt.savefig(f'{plot_folder}/SVM_lineal_c{C}.png',  dpi=300)


def kerner_gaussiano(X: np.ndarray, y: np.ndarray, C: float, sigma: float) -> None:
    """Gaussian kernel
    Args:
        X (np.ndarray): X train dataa
        y (np.ndarray): y train data
        C (float): regularization parameter
        sigma (float): scale parameter
    """
    svm_gauss: svm.SVC = svm.SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
    svm_gauss.fit(X, y.ravel())
    x1: np.ndarray = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    x2: np.ndarray = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
    X1, X2 = np.meshgrid(x1, x2)
    yp: np.ndarray = svm_gauss.predict(
        np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape)
    plt.contour(X1, X2, yp, colors='darkgreen', linewidths=1)
    plt.scatter(X[y.ravel() == 1, 0], X[y.ravel() == 1, 1], c='b', marker='x')
    plt.scatter(X[y.ravel() == 0, 0], X[y.ravel() == 0, 1], c='y', marker='o')
    plt.xticks(np.arange(0.0, 1.2, 0.2))
    plt.yticks(np.arange(0.4, 1.1, 0.1))
    plt.savefig(f'{plot_folder}/SVM_gauss_c{C}_sigma{sigma}.png', dpi=300)


def seleccion_sigma_C() -> None:
    """Selects the best C and sigma for the gaussian kernel
    """
    X, y, Xval, yval = load_data3('data/ex6data3.mat')
    C_values: list[float] = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
    sigma_values: list[float] = C_values
    best_score: float = 0
    best_params: tuple[float] = (0, 0)
    for C in C_values:
        for sigma in sigma_values:
            svm_gauss = svm.SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
            svm_gauss.fit(X, y.ravel())
            score = svm_gauss.score(Xval, yval)
            if score > best_score:
                best_score = score
                best_params = (C, sigma)
    print(f'Best score: {best_score}')
    print(f'Best params: {best_params}')
    svm_gauss: svm.SVC = svm.SVC(
        kernel='rbf', C=best_params[0], gamma=1/(2*best_params[1]**2))
    svm_gauss.fit(X, y.ravel())
    x1: np.ndarray = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    x2: np.ndarray = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
    X1, X2 = np.meshgrid(x1, x2)
    yp: np.ndarray = svm_gauss.predict(
        np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape)
    plt.contour(X1, X2, yp, colors='darkgreen', linewidths=1)
    plt.scatter(X[y.ravel() == 1, 0], X[y.ravel() == 1, 1], c='b', marker='x')
    plt.scatter(X[y.ravel() == 0, 0], X[y.ravel() == 0, 1], c='y', marker='o')
    plt.yticks(np.arange(-0.8, 0.7, 0.2))
    plt.xticks(np.arange(-0.6, 0.4, 0.1))
    plt.savefig(f'{plot_folder}/SVM_gauss_best.png', dpi=300)


def apartado_A() -> None:
    """Apartado A
    """
    X, y = load_data('data/ex6data1.mat')
    print("Linear kernel with C=1")
    kernel_linear(X, y, 1.0)
    plt.clf()
```

```
    print("Linear kernel with C=100")
    kernel_linear(X, y, 100.0)
    X, y = load_data('data/ex6data2.mat')
    plt.clf()
    print("Gaussian kernel with C=1 and sigma=0.1")
    kerner_gaussiano(X, y, 1.0, 0.1)
    plt.clf()
    print("Selecting C and sigma for gaussian kernel")
    seleccion_sigma_C()


def load_data_spam() -> list[tuple[list[str], int]]:
    """Loads the spam data
    """
    modes:  list[str] = ['spam', 'easy_ham', 'hard_ham']
    cantidades: list[int] = [500, 2551, 250]
    spam_flag = [1, 0, 0]
    correos = []
    for mode, number, spam in zip(modes, cantidades, spam_flag):
        progress = 0
        length = 50
        for file in range(1, number + 1):
            file = str(file)
            with codecs.open(f'./data_spam/spam/{mode}/{file.zfill(4)}.txt', 'r', encoding=
    'utf-8', errors='ignore') as f:
                progress += 1
                bar_length = int(length * progress / number)
                bar = '[' + '=' * bar_length + \
                    ' ' * (length - bar_length) + ']'
                print(f'\rLoading {mode} {bar} {progress}/{number}', end='')
                email = f.read()
                token_list = email2TokenList(email)
                correos.append((token_list, spam))
        print()
    print(len(correos))
    return correos


def transform_mail(correos, vocab) -> tuple[np.ndarray, np.ndarray]:
    """Transforms the emails into a matrix of length of the vocabulary with 1 if the word
    is in the email

    Args:
        correos (_type_): mails
        vocab (_type_): dictionary

    Returns:
        tuple[np.ndarray, np.ndarray]: transformed emails with label indicating if its spam
     or not
    """
    X = []
    y = []

    for c, s in correos:
        x = np.zeros(len(vocab) + 1)
        for word in c:
            if word in vocab:
                x[vocab[word]] = 1
        X.append(x)
        y.append(s)

    return np.array(X), np.array(y)


def plot_results(train_scores: list[float], cv_scores: list[float], test_scores: list[float
    ], times: list[float]) -> None:
    """Plots the results
    Args:
        train_scores (list[float]): train scores
        cv_scores (list[float]): cv scores
        test_scores (list[float]): test scores
        times (list[float]): times
    """
    plt.clf()
    X: np.ndaarray = np.array(
        ['Logistic Regression', 'SVM', 'NN', 'Pytorch', 'Poly'])
```

```
    x = np.arange(len(X))
    plt.bar(x-0.2,
            train_scores, 0.2, label=f'Train')
    plt.bar(x,
            cv_scores, 0.2, label=f'CV')
    plt.bar(x+0.2,
            test_scores, 0.2, label=f'Test')
    plt.legend()
    plt.xticks(x, X)
    plt.savefig(f'{plot_folder}/results.png', dpi=300)
    plt.clf()
    plt.plot(X, times)
    plt.savefig(f'{plot_folder}/times.png', dpi=300)


def compare_results() -> None:
    """Compares the results of the different models
    """
    lr_data = sio.loadmat('res/logistic_regression.mat')
    svm_data = sio.loadmat('res/svm.mat')
    nn_data = sio.loadmat('res/nn.mat')
    pytorch_data = sio.loadmat('res/pytorch.mat')
    poly_data = sio.loadmat('res/poly.mat')
    print('Logistic Regression')
    print(f"Score: {lr_data['train_score']}")
    print(f"CV Score: {lr_data['cv_score']}")
    print(f"Test Score: {lr_data['test_score']}")
    print(f"Time: {lr_data['time']}")
    print(f'Best params: {lr_data["best_params"]}')
    print('SVM')
    print(f"Score: {svm_data['train_score']}")
    print(f"CV Score: {svm_data['cv_score']}")
    print(f"Test Score: {svm_data['test_score']}")
    print(f"Time: {svm_data['time']}")
    print(f'Best params: {svm_data["best_params"]}')
    print('NN')
    print(f"Score: {nn_data['train_score']}")
    print(f"CV Score: {nn_data['cv_score']}")
    print(f"Test Score: {nn_data['test_score']}")
    print(f"Time: {nn_data['time']}")
    print(f'Best params: {nn_data["best_params"]}')
    print('Pytorch')
    print(f"Score: {pytorch_data['train_score']}")
    print(f"CV Score: {pytorch_data['cv_score']}")
    print(f"Test Score: {pytorch_data['test_score']}")
    print(f"Time: {pytorch_data['time']}")
    print(f'Best params: {pytorch_data["best_params"]}')
    print('Poly')
    print(f"Score: {poly_data['train_score']}")
    print(f"CV Score: {poly_data['cv_score']}")
    print(f"Test Score: {poly_data['test_score']}")
    print(f"Time: {poly_data['time']}")
    print(f'Best params: {poly_data["best_params"]}')

    train_scores = [lr_data['train_score'][0][0], svm_data['train_score']
                    [0][0], nn_data['train_score'][0][0], pytorch_data['train_score'
    ][0][0], poly_data['train_score'][0][0]]
    cv_scores = [lr_data['cv_score'][0][0], svm_data['cv_score']
                 [0][0], nn_data['cv_score'][0][0], pytorch_data['cv_score'][0][0],
    poly_data['cv_score'][0][0]]
    test_scores = [lr_data['test_score'][0][0], svm_data['test_score']
                   [0][0], nn_data['test_score'][0][0], pytorch_data['test_score'][0][0],
    poly_data['test_score'][0][0]]
    times = [lr_data['time'][0][0], svm_data['time'][0][0],
             nn_data['time'][0][0], pytorch_data['time'][0][0], poly_data['time'][0][0]]
    print(train_scores)
    plot_results(train_scores, cv_scores, test_scores, times)


def apartado_B():
    """Apartado B
    """
    correos = load_data_spam()
    vocab = getVocabDict()
    X, y = transform_mail(correos, vocab)
    if not os.path.exists(f'res/svm.mat'):
```

```python
        print('Training SVM')
        SVM_Trainer.trainer(X, y)
    if not os.path.exists(f'res/logistic_regression.mat'):
        print('Training Logistic Regression')
        Logic_Regression_Trainer.LR_trainer(X, y)
    if not os.path.exists(f'res/pytorch.mat'):
        print('Training Pytorch')
        pytorch_trainer.trainer(X, y)
    if not os.path.exists(f'res/nn.mat'):
        print('Training NN')
        nn_trainer.trainer(X, y)
    if not os.path.exists(f'res/poly.mat'):
        print('Training Poly')
        Poly_trainer.trainer(X, y)

    compare_results()


def Pruebas():
    correos = load_data_spam()
    vocab = getVocabDict()
    X, y = transform_mail(correos, vocab)
    lr_data = sio.loadmat('res/logistic_regression.mat')
    svm_data = sio.loadmat('res/svm.mat')
    nn_data = sio.loadmat('res/nn.mat')
    pytorch_data = sio.loadmat('res/pytorch.mat')

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=22)
    X_cv, X_test, y_cv, y_test = train_test_split(
        X_test, y_test, test_size=0.5, random_state=22)

    print('Logistic Regression')
    w = np.zeros(X.shape[1] + 1)
    b = 1
    X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
    best_params = lr_data['best_params'][0]
    w, b, _, _ = Logic_Regression_Trainer.gradient_descent(
        X_train, y_train, w, b, Logic_Regression_Trainer.compute_cost_reg, Logic_Regression_Trainer.compute_gradient_reg, best_params[0], 1000, best_params[1])

    train_score = Logic_Regression_Trainer.predict_check(
        X_train, y_train, w, b)
    print(f'Train score: {train_score}')
    X_cv = np.hstack((np.ones((X_cv.shape[0], 1)), X_cv))
    cv_score = Logic_Regression_Trainer.predict_check(X_cv, y_cv, w, b)
    print(f'CV score: {cv_score}')
    X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
    test_score = Logic_Regression_Trainer.predict_check(X_test, y_test, w, b)
    print(f'Test score: {test_score}')

    print('SVM')
    best_params = svm_data['best_params'][0]
    svm_gauss = svm.SVC(
        kernel='rbf', C=best_params[0], gamma=1/(2*best_params[1]**2))
    svm_gauss.fit(X_train, y_train.ravel())

    test_score = svm_gauss.score(X_test, y_test)
    cv_score = svm_gauss.score(X_cv, y_cv)
    train_score = svm_gauss.score(X_train, y_train)
    print(f'Test score: {test_score}')
    print(f'CV score: {cv_score}')
    print(f'Train score: {train_score}')

    print('Pytorch')
    best_params = pytorch_data['best_params'][0]
    criterion = nn.CrossEntropyLoss().to(device)
    model = ComplexModel(X_train.shape[1])
    optimizer = optim.Adam(model.parameters(), lr=best_params[1],
                           weight_decay=best_params[0])
    train_dl = train_data(X_train, y_train)
    model = train_model(model, train_dl, criterion, optimizer, 20)

    test_score = pred_check(
        model(torch.tensor(X_test, dtype=torch.float).to(device)), y_test)
    cv_score = pred_check(
```

```
        model(torch.tensor(X_cv, dtype=torch.float).to(device)), y_cv)
    train_score = pred_check(
        model(torch.tensor(X_train, dtype=torch.float).to(device)), y_train)

    print(f'Test score: {test_score}')
    print(f'CV score: {cv_score}')
    print(f'Train score: {train_score}')

    print('NN')
    best_params = nn_data['best_params'][0]
    input_layer_size = X.shape[1]
    hidden_layer_size = 125
    num_labels = 2
    yA = [0 if i == 1 else 1 for i in y_train]
    yB = [1 if i == 1 else 0 for i in y_train]
    y_encoded = np.array([yA, yB]).T
    theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
    theta2 = np.random.rand(num_labels, hidden_layer_size + 1)
    _, _, _, theta1, theta2, nn_trainer.train_model(X_train, y_encoded, theta1,
                                                    theta2, best_params[0], best_params[1],
     1000)
    yA = [0 if i == 1 else 1 for i in y_test]
    yB = [1 if i == 1 else 0 for i in y_test]
    y_encoded = np.array([yA, yB]).T
    nn_score = nn_trainer.predict_check(X_test, y_encoded, theta1, theta2)
    print(f'Test score: {nn_score}')
    yA = [0 if i == 1 else 1 for i in y_cv]
    yB = [1 if i == 1 else 0 for i in y_cv]
    y_encoded = np.array([yA, yB]).T
    nn_score = nn_trainer.predict_check(X_cv, y_encoded, theta1, theta2)
    print(f'CV score: {nn_score}')
    yA = [0 if i == 1 else 1 for i in y_train]
    yB = [1 if i == 1 else 0 for i in y_train]
    y_encoded = np.array([yA, yB]).T
    nn_score = nn_trainer.predict_check(X_train, y_encoded, theta1, theta2)
    print(f'Train score: {nn_score}')


def main() -> None:
    apartado_A()
    # apartado_B()
    Pruebas()


if __name__ == '__main__':
    main()
```

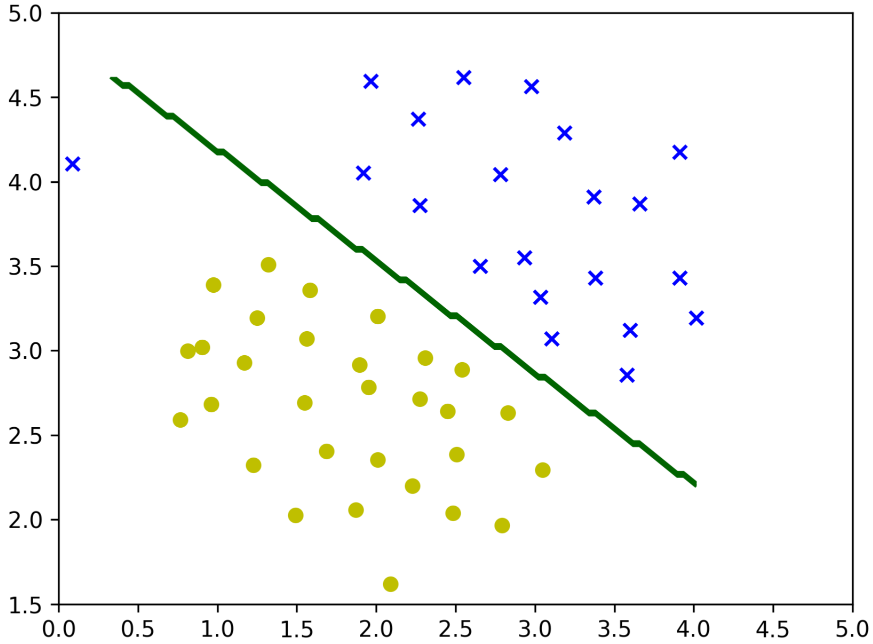

Figura 1.1: SVM lineal con C=1.0

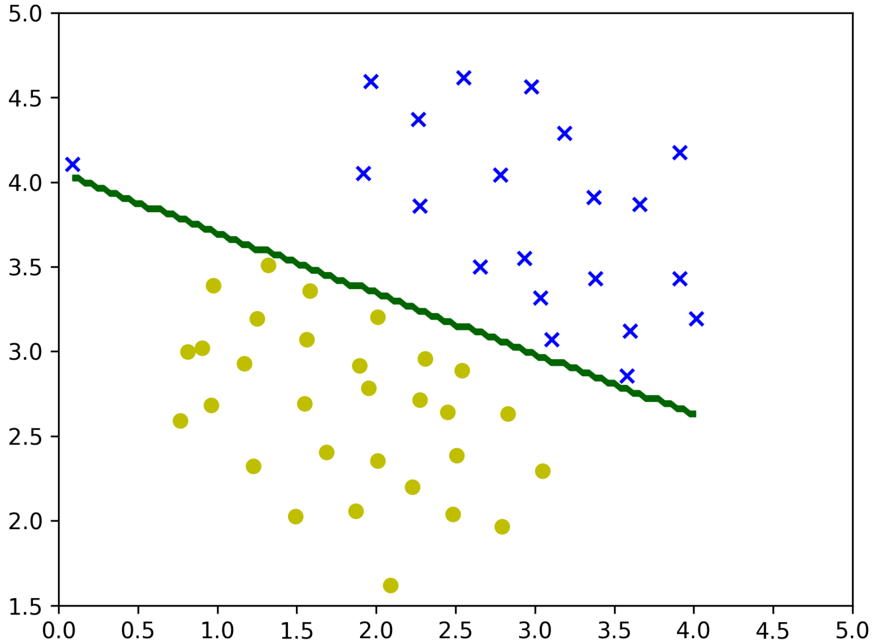

Figura 1.2: SVM lineal con C=100.0

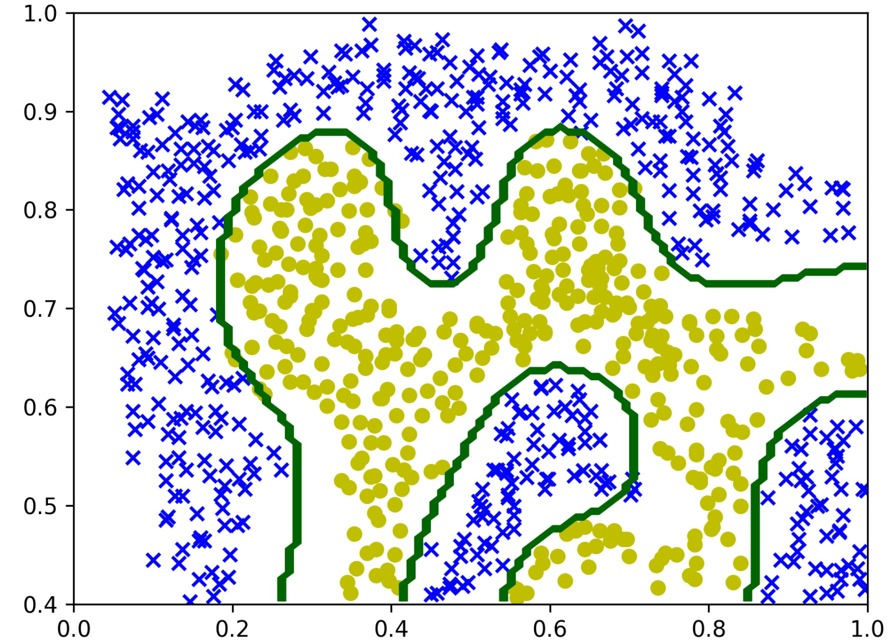

Figura 1.3: SVM gaussiano con C=1.0 y sigma=0.1

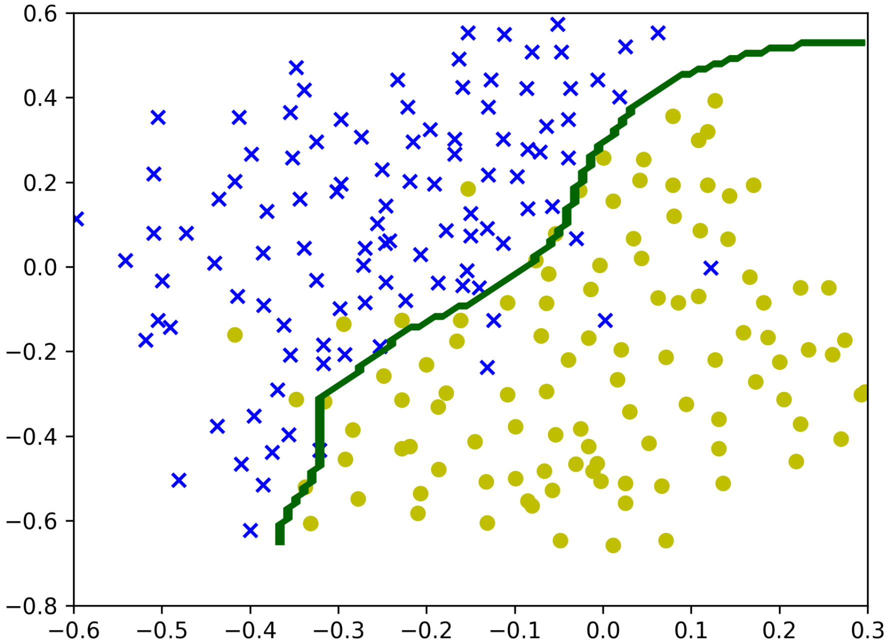

Figura 1.4: SVM gaussiano con C=1.0 y sigma=0.1, mejor configuración para este problema

2. Apartado B

Para este problema usaremos distintos modelos:

- **Regresión lógica:** parece que sobreentrena en train, tiene como resultados 100, 97, 98 en train, validación y test respectivamente. Tarda 7.73 segundos en su mejor modelo con parámetros (10, 0.1)
- **SVM gaussiano:** mejor modelo de todos, tiene 98, 97, 98 en train, validación y test respectivamente. Tarda 1 segundos en su mejor modelo con parámetros (1.0, 10.)
- **NN:** el modelo que más tarda de todos (posiblemente porque está implementado en python a mano y no con una biblioteca hecha en un lenguaje competente) con resultados 96, 96 y 95 en train, validación y test respectivamente. Tarda 214 segundos en su mejor modelo con parámetros (3, 30)
- **Pytorch:** modelo entrenado en GPU con resultados 97, 97, 96 en train, validación y test respectivamente. Tarda 33 segundos en su mejor modelo con parámetros (0.001, 0.01)
- **PolynomialTransformer:** inviable a partir de grado 1, crashea el ordenador porque la matriz de datos es demasiado grande.

Código del entrenador de regresión lógica:

```python
import numpy as np
import copy
import time
import scipy.io as sio
import concurrent.futures
from sklearn.model_selection import train_test_split


def compute_cost_reg(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_: float
    = 1) -> float:
    """
    Computes the cost over all examples
    Args:
      X : (array_like Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : (array_like Shape (n,)) Values of bias parameter of the model
      lambda_ : (scalar, float)    Controls amount of regularization
    Returns:
      total_cost: (scalar)         cost
    """

    total_cost = compute_cost(X, y, w, b)
    total_cost += (lambda_ / (2 * X.shape[0])) * np.sum(w**2)

    return total_cost


def loss(X: np.ndarray, Y: np.ndarray, fun: np.ndarray, w: np.ndarray, b: float) -> float:
    """loss function for the logistic regression

    Args:
        X (np.ndarray): X values
        Y (np.ndarray): Expected y results
        fun (np.ndarray): logistic regression function
        w (np.ndarray): weights
        b (float): bias

    Returns:
        float: total loss of the regression
    """

    return (-Y * np.log(fun(X, w, b) + 1e-6)) - ((1 - Y) * np.log(1 - fun(X, w, b) + 1e-6))


def compute_cost(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_=None) ->
    float:
    """
    Computes the cost over all examples
    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : scalar Values of bias parameter of the model
      lambda_: unused placeholder
    Returns:
      total_cost: (scalar)         cost
    """
```

```python
    # apply the loss function for each element of the x and y arrays
    loss_v = loss(X, y, function, w, b)
    total_cost = np.sum(loss_v)
    total_cost /= X.shape[0]

    return total_cost


def compute_gradient(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_=None)
    -> tuple[float, np.ndarray]:
    """
    Computes the gradient for logistic regression

    Args:
        X : (ndarray Shape (m,n)) variable such as house size
        y : (array_like Shape (m,1)) actual value
        w : (array_like Shape (n,1)) values of parameters of the model
        b : (scalar)                 value of parameter of the model
        lambda_: unused placeholder
    Returns
        dj_db: (scalar)                The gradient of the cost w.r.t. the parameter b.
        dj_dw: (array_like Shape (n,1)) The gradient of the cost w.r.t. the parameters w.
    """

    func = function(X, w, b)

    dj_dw = np.dot(func - y, X)
    dj_dw /= X.shape[0]

    dj_db = np.sum(func - y)
    dj_db /= X.shape[0]

    return dj_db, dj_dw


def compute_gradient_reg(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_:
    float = 1) -> tuple[float, np.ndarray]:
    """
    Computes the gradient for linear regression

    Args:
      X : (ndarray Shape (m,n))   variable such as house size
      y : (ndarray Shape (m,))    actual value
      w : (ndarray Shape (n,))    values of parameters of the model
      b : (scalar)                value of parameter of the model
      lambda_ : (scalar,float)    regularization constant
    Returns
      dj_db: (scalar)             The gradient of the cost w.r.t. the parameter b.
      dj_dw: (ndarray Shape (n,)) The gradient of the cost w.r.t. the parameters w.

    """
    dj_db, dj_dw = compute_gradient(X, y, w, b)
    dj_dw += (lambda_ / X.shape[0]) * w

    return dj_db, dj_dw


def gradient_descent(X: np.ndarray, y: np.ndarray, w_in: np.ndarray, b_in: float,
    cost_function: float, gradient_function: float, alpha: float, num_iters: int, lambda_:
    float = None) -> tuple[np.ndarray, float, np.ndarray, np.ndarray]:
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X :    (array_like Shape (m, n)
      y :    (array_like Shape (m,))
      w_in : (array_like Shape (n,))  Initial values of parameters of the model
      b_in : (scalar)                 Initial value of parameter of the model
      cost_function:                  function to compute cost
      alpha : (float)                 Learning rate
      num_iters : (int)               number of iterations to run gradient descent
      lambda_ (scalar, float)         regularization constant

    Returns:
      w : (array_like Shape (n,)) Updated values of parameters of the model after
```

```
            running gradient descent
      b : (scalar)                Updated value of parameter of the model after
            running gradient descent
      J_history : (ndarray): Shape (num_iters,) J at each iteration,
            primarily for graphing later
    """

    w = copy.deepcopy(w_in)
    b = b_in
    predict_history = [predict_check(X, y, w, b)]
    J_history = [cost_function(X, y, w, b, lambda_)]

    for i in range(num_iters):
        dj_db, dj_dw = gradient_function(X, y, w, b, lambda_)
        w = w - (alpha * dj_dw)
        b -= alpha * dj_db
        J_history.append(cost_function(X, y, w, b, lambda_))
        predict_history.append(predict_check(X, y, w, b))

    return w, b, np.array(J_history), predict_history


def predict(X, w, b) -> np.ndarray:
    """
    Predict whether the label is 0 or 1 using learned logistic
    regression parameters w and b

    Args:
    X : (ndarray Shape (m, n))
    w : (array_like Shape (n,))      Parameters of the model
    b : (scalar, float)              Parameter of the model

    Returns:
    p: (ndarray (m,1))
        The predictions for X using a threshold at 0.5
    """

    p = np.vectorize(lambda x: 1 if x > 0.5 else 0)(
        function(X, w, b))
    return p


def predict_check(X, Z, w, b) -> float:
    """Gives a percentage of the accuracy of the prediction

    Args:
        X (_type_): X train data
        Z (_type_): expected values
        w (_type_): weights
        b (_type_): bias

    Returns:
        float: percentage of accuracy
    """
    p = predict(X, w, b)
    return np.sum(p == Z) / Z.shape[0]


def sigmoid(z: np.ndarray) -> np.ndarray:
    """
    Compute the sigmoid of z

    Args:
        z (ndarray): A scalar, numpy array of any size.

    Returns:
        g (ndarray): sigmoid(z), with the same shape as z

    """

    g = 1/(1+np.exp(-z))

    return g


def function(x: np.ndarray, w: np.ndarray, b: float) -> np.ndarray:
```

```python
    """Function using ''sigmoid'' to calculate the value of y to the given x, w and b

    Args:
        x (np.ndarray): X data
        w (np.ndarray): w data
        b (float): b data

    Returns:
        np.ndarray: final value after the sigmoid
    """
    return sigmoid(np.dot(x, w) + b)


def train_model(X: np.ndarray, y: np.ndarray, x_cv: np.ndarray, y_cv: np.ndarray, alpha:
    float, lambda_: float, num_iters: int) -> tuple[float, float, float]:
    """Train the model with the given parameters
    Args:
        X (np.ndarray): Training data
        y (np.ndarray): Training target
        x_cv (np.ndarray): Cross validation data
        y_cv (np.ndarray): Cross validation target
        alpha (float): Learning rate
        lambda_ (float): Regularization parameter
        num_iters (int): Number of iterations
    Returns:
        tuple[float, float, float]: Learning rate, Regularization parameter, Score
    """
    print(f'Alpha: {alpha} Lambda: {lambda_}')
    m, n = X.shape
    X = np.hstack((np.ones((m, 1)), X))
    x_cv = np.hstack((np.ones((x_cv.shape[0], 1)), x_cv))
    w = np.zeros(X.shape[1])
    b = 1
    w, b, _, _ = gradient_descent(
        X, y, w, b, compute_cost_reg, compute_gradient_reg, alpha, num_iters, lambda_)
    score = predict_check(x_cv, y_cv, w, b)
    return (alpha, lambda_, score)


def LR_trainer(X: np.ndarray, y: np.ndarray) -> None:
    """Trains the model with the given data
    Args:
        X (np.ndarray): Input data
        y (np.ndarray): Target data
    """
    alphas = [0.1, 0.3, 1, 3, 10, 30]
    lambdas = [0.1, 0.3, 1, 3, 10, 30]
    num_iters = 1000
    best_score = 0
    best_params = (0, 0)

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, shuffle=True, random_state=22)
    X_cv, X_test, y_cv, y_test = train_test_split(
        X_test, y_test, test_size=0.5, shuffle=True, random_state=22)

    with concurrent.futures.ProcessPoolExecutor() as executor:
        futures = []
        for lambda_ in lambdas:
            for alpha in alphas:
                futures.append(executor.submit(
                    train_model, X_train, y_train, X_cv, y_cv, alpha, lambda_, num_iters))

        for future in concurrent.futures.as_completed(futures):
            alpha, lambda_, score = future.result()
            print(f'Alpha: {alpha} Lambda: {lambda_} Score: {score}')
            if score > best_score:
                best_score = score
                best_params = (alpha, lambda_)

    print(f'Best score: {best_score}')
    print(f'Best params: {best_params}')

    start = time.time()
    w = np.zeros(X.shape[1] + 1)
    b = 1
```

```python
    X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
    w, b, _, _ = gradient_descent(
        X_train, y_train, w, b, compute_cost_reg, compute_gradient_reg, best_params[0], num_iters, best_params[1])
    end = time.time()
    print(f'Training time: {end-start}')
    train_score = predict_check(X_train, y_train, w, b)
    print(f'Train score: {train_score}')
    X_cv = np.hstack((np.ones((X_cv.shape[0], 1)), X_cv))
    cv_score = predict_check(X_cv, y_cv, w, b)
    print(f'CV score: {cv_score}')
    X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
    test_score = predict_check(X_test, y_test, w, b)
    print(f'Test score: {test_score}')
    sio.savemat('res/logistic_regression.mat', {'w': w, 'b': b, 'train_score': train_score,
                'cv_score': cv_score, 'test_score': test_score, 'best_params': best_params, 'time': end-start})
```

Código del entrenador de SVM gaussiano:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import time
from sklearn import svm
import scipy.io as sio
import concurrent.futures


def train_model(C: float, sigma: float, x_train: np.ndarray, y_train: np.ndarray, x_cv: np.ndarray, y_cv: np.ndarray) -> tuple[float, float, float]:
    """Train the model with the given parameters
    Args:
        C (float): Regularization parameter
        sigma (float): Gaussian kernel parameter
        x_train (np.ndarray): Training data
        y_train (np.ndarray): Training target
        x_cv (np.ndarray): Cross validation data
        y_cv (np.ndarray): Cross validation target
    Returns:
        tuple[float, float, float]: Regularization parameter, Gaussian kernel parameter, Score
    """
    print(f'C: {C} sigma: {sigma}')
    svm_gauss = svm.SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
    svm_gauss.fit(x_train, y_train.ravel())
    score = svm_gauss.score(x_cv, y_cv.ravel())
    return (C, sigma, score)


def trainer(X: np.ndarray, y: np.ndarray) -> None:
    """Trains the model with the given data
    Args:
        X (np.ndarray): Input data
        y (np.ndarray): Target data
    """
    C_values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
    sigma_values = C_values
    x_train, x_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3,  random_state=22)
    x_cv, x_test, y_cv, y_test = train_test_split(
        x_test, y_test, test_size=0.5, random_state=22)
    best_score = 0
    best_params = (0, 0)

    with concurrent.futures.ProcessPoolExecutor() as executor:
        futures = []
        for C in C_values:
            for sigma in sigma_values:
                futures.append(executor.submit(
                    train_model, C, sigma, x_train, y_train, x_cv, y_cv))

        for future in concurrent.futures.as_completed(futures):
            C, sigma, score = future.result()
            print(f'C: {C} sigma: {sigma} score: {score}')
            if score > best_score:
```

```python
                best_score = score
                best_params = (C, sigma)

    print(f'Best score: {best_score}')

    start = time.time()
    svm_gauss = svm.SVC(
        kernel='rbf', C=best_params[0], gamma=1/(2*best_params[1]**2))
    svm_gauss.fit(x_train, y_train.ravel())
    end = time.time()
    print(f'Training time: {end-start}')

    test_score = svm_gauss.score(x_test, y_test)
    cv_score = svm_gauss.score(x_cv, y_cv)
    train_score = svm_gauss.score(x_train, y_train)
    sio.savemat('res/svm.mat', {'train_score': train_score,
                                'cv_score': cv_score, 'test_score': test_score, 'best_params': best_params, 'time': end-start})
```

Código del entrenador de NN:

```python
import numpy as np
import scipy.io as sio
import time
from sklearn.model_selection import train_test_split


def sigmoid(z: np.ndarray) -> np.ndarray:
    """
    Compute the sigmoid of z

    Args:
        z (ndarray): A scalar, numpy array of any size.

    Returns:
        g (ndarray): sigmoid(z), with the same shape as z

    """

    g = 1/(1+np.exp(-z))

    return g


def fix_data(X: np.ndarray) -> np.ndarray:
    """Fixes the data to avoid log(0) errors

    Args:
        X (np.ndarray): train data

    Returns:
        np.ndarray: matrix with no 0 or 1 values
    """
    return X + 1e-7


def cost(theta1: np.ndarray, theta2: np.ndarray, X: np.ndarray, y: np.ndarray, lambda_: float = 0.0) -> float:
    """
    Compute cost for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
```

```
        (number of examples x number of labels).

    lambda_ : float
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.

    """
    L = 2
    layers = [theta1, theta2]
    k: int = y.shape[1]
    h, z = neural_network(X, [theta1, theta2])

    h = h[-1]

    h = fix_data(h)

    J = y * np.log(h + 1e-7)
    J += (1 - y) * np.log(1 - h + 1e-7)

    J = -1 / X.shape[0] * np.sum(J)

    if lambda_ != 0:
        reg = 0
        for layer in layers:
            reg += np.sum(layer[:, 1:] ** 2)
        J += lambda_ / (2 * X.shape[0]) * reg
    return J


def neural_network(X: np.ndarray, thetas: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    """Generate the neural network with a given set of weights

    Args:
        X (np.ndarray): data
        thetas (np.ndarray): array containing the weights for each layer

    Returns:
        tuple[np.ndarray, np.ndarray]: tuple containing the activations and the z values
    for each layer
    """
    a = []
    z = []
    a.append(X.copy())
    for theta in thetas:
        a[-1] = np.hstack((np.ones((a[-1].shape[0], 1)), a[-1]))
        z.append(np.dot(a[-1], theta.T))
        a.append(sigmoid(z[-1]))
    return a, z


def backprop(theta1: np.ndarray, theta2: np.ndarray, X: np.ndarray, y: np.ndarray, lambda_:
 float) -> tuple[float, np.ndarray, np.ndarray]:
    """
    Compute cost and gradient for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

```

```
    lambda_ : float
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.

    grad1 : array_like
        Gradient of the cost function with respect to weights
        for the first layer in the neural network, theta1.
        It has shape (2nd hidden layer size x input size + 1)

    grad2 : array_like
        Gradient of the cost function with respect to weights
        for the second layer in the neural network, theta2.
        It has shape (output layer size x 2nd hidden layer size + 1)

    """
    m = X.shape[0]
    L = 2

    delta = np.empty(2, dtype=object)
    delta[0] = np.zeros(theta1.shape)
    delta[1] = np.zeros(theta2.shape)

    a, z = neural_network(X, [theta1, theta2])

    for k in range(m):
        a1k = a[0][k, :]
        a2k = a[1][k, :]
        hk = a[2][k, :]
        yk = y[k, :]

        d3k = hk - yk
        d2k = np.dot(theta2.T, d3k) * a2k * (1 - a2k)

        delta[0] = delta[0] + \
            np.matmul(d2k[1:, np.newaxis], a1k[np.newaxis, :])
        delta[1] = delta[1] + np.matmul(d3k[:, np.newaxis], a2k[np.newaxis, :])

    grad1 = delta[0] / m
    grad2 = delta[1] / m

    if lambda_ != 0:
        grad1[:, 1:] += lambda_ / m * theta1[:, 1:]
        grad2[:, 1:] += lambda_ / m * theta2[:, 1:]

    J = cost(theta1, theta2, X, y, lambda_)

    return (J, grad1, grad2)


def gradient_descent(X: np.ndarray, y: np.ndarray, theta1: np.ndarray, theta2: np.ndarray, alpha: float, lambda_: float, num_iters: int) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Generates the gradient descent for the neural network

    Args:
        X (np.ndarray): Train data
        y (np.ndarray): Expected output in one hot encoding
        theta1 (np.ndarray): initial weights for the first layer
        theta2 (np.ndarray): initial weights for the second layer
        alpha (float): learning rate
        lambda_ (float): regularization parameter
        num_iters (int): number of iterations to run

    Returns:
        tuple[np.ndarray, np.ndarray, np.ndarray]: tuple with the final weights for the first and second layer and the cost history
    """
    m = X.shape[0]
    J_history = np.zeros(num_iters)
    for i in range(num_iters):
        print('Iteration: ', i + 1, '/', num_iters, end='\r')
        J, grad1, grad2 = backprop(theta1, theta2, X, y, lambda_)
```

```
        theta1 = theta1 - alpha * grad1
        theta2 = theta2 - alpha * grad2
        J_history[i] = J
    print('Gradient descent finished.')
    return theta1, theta2, J_history


def train_model(X, y, x_cv, y_cv, alpha, lambda_, num_iters):
    # start = time.time()
    print(f'Alpha: {alpha} Lambda: {lambda_}')
    input_layer_size = X.shape[1]
    hidden_layer_size = 125
    num_labels = 2
    yA = [0 if i == 1 else 1 for i in y]
    yB = [1 if i == 1 else 0 for i in y]
    y_encoded = np.array([yA, yB]).T

    theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
    theta2 = np.random.rand(num_labels, hidden_layer_size + 1)

    theta1, theta2, J_history = gradient_descent(
        X, y_encoded, theta1, theta2, alpha, lambda_, num_iters)

    score = predict_percentage(x_cv, y_cv, theta1, theta2)
    # time = time.time() - start
    return (alpha, lambda_, score, theta1, theta2)


def prediction(X: np.ndarray, theta1: np.ndarray, theta2: np.ndarray) -> np.ndarray:
    """Generates the neural network prediction

    Args:
        X (np.ndarray): data
        theta1 (np.ndarray): first layer weight
        theta2 (np.ndarray): second layer weight

    Returns:
        np.ndarray: best prediction for each row in `X`
    """
    m = X.shape[0]
    p = np.zeros(m)
    a, z = neural_network(X, [theta1, theta2])
    h = a[-1]

    return np.argmax(h, axis=1)


def predict_percentage(X: np.ndarray, y: np.ndarray, theta1: np.ndarray, theta2: np.ndarray
    ) -> float:
    """Gives the accuracy of the neural network

    Args:
        X (ndarray): Train data
        y (ndarray): Expected output
        theta1 (ndarray): First layer weights
        theta2 (ndarray): Second layer weights

    Returns:
        float: Accuracy of the neural network
    """
    m = X.shape[0]
    p = prediction(X, theta1, theta2)

    return p[p == y].size / m


def trainer(X: np.ndarray, y: np.ndarray) -> None:
    lambdas = [0.01, 0.03,  0.1, 0.3, 1, 3, 10, 30]
    alphas = lambdas
    num_iters = 100
    best_score = 0
    best_params = (0, 0)
    input_layer_size = X.shape[1]
    hidden_layer_size = 125
    num_labels = 2
    best_time = 0
```

```python
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3,  random_state=22)
    X_cv, X_test, y_cv, y_test = train_test_split(
        X_test, y_test, test_size=0.5,  random_state=22)
    model = (np.array([]), np.array([]))

    for alpha in alphas:
        for lambda_ in lambdas:

            start = time.time()
            print(f'Alpha: {alpha} Lambda: {lambda_}')
            input_layer_size = X.shape[1]
            hidden_layer_size = 125
            num_labels = 2
            yA = [0 if i == 1 else 1 for i in y]
            yB = [1 if i == 1 else 0 for i in y]
            y_encoded = np.array([yA, yB]).T
            theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
            theta2 = np.random.rand(num_labels, hidden_layer_size + 1)

            theta1, theta2, J_history = gradient_descent(
                X, y_encoded, theta1, theta2, alpha, lambda_, num_iters)

            score = predict_percentage(X_cv, y_cv, theta1, theta2)
            print(f'Score: {score}')
            aux_time = time.time() - start
            if score > best_score:
                best_score = score
                best_params = (alpha, lambda_)
                model = (theta1, theta2)
                best_time = aux_time
    print(f'Best score: {best_score}')
    print(f'Best params: {best_params}')

    theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
    theta2 = np.random.rand(num_labels, hidden_layer_size + 1)
    yA = [0 if i == 1 else 1 for i in y_train]
    yB = [1 if i == 1 else 0 for i in y_train]
    y_encoded = np.array([yA, yB]).T

    theta1, theta2, = model
    print(f'Training time: {best_time}')

    train_score = predict_percentage(X_train, y_train, theta1, theta2)
    print(f'Train score: {train_score}')
    cv_score = predict_percentage(X_cv, y_cv, theta1, theta2)
    print(f'CV score: {cv_score}')
    test_score = predict_percentage(X_test, y_test, theta1, theta2)
    print(f'Test score: {test_score}')
    sio.savemat('res/nn.mat',
                {'theta1': theta1, 'theta2': theta2, 'train_score': train_score, 'cv_score'
    : cv_score, 'test_score': test_score, 'best_params': best_params, 'time': best_time})
```

Código del entrenador de Pytorch:

```python
import torch.nn as nn
import torch.optim as optim
import numpy as np
from sklearn.model_selection import train_test_split
import scipy.io as sio
import time
import torch

# Select cuda device if available to speed up training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

if torch.cuda.is_available():
    print(f'Using GPU {torch.cuda.get_device_name()}')
else:
    print('Using CPU')


def train_data(x: np.ndarray, y: np.ndarray) -> torch.utils.data.DataLoader:
    """ Create a DataLoader object from the input data
    Args:
        x (np.ndarray): Input data
```

```python
        y (np.ndarray): Target data
    """
    return torch.utils.data.DataLoader(torch.utils.data.TensorDataset(
        torch.tensor(x, dtype=torch.float).to(device), torch.tensor(y).to(device)),
    batch_size=2, shuffle=True)


def train_model(model: nn.Sequential, train_dl: torch.utils.data.DataLoader, criterion: nn.
    CrossEntropyLoss, optimizer: optim.Adam, epochs: int) -> nn.Sequential:
    """ Train the model with the given data
    Args:
        model (nn.Sequential): Model to train
        train_dl (torch.utils.data.Dataloader): DataLoader object with the training data
        criterion (nn.CrossEntropyLoss): Loss function
        optimizer (optim.Adam): Optimizer
        epochs (int): Number of epochs to train the model
    Returns:
        nn.Sequential: Trained model
    """
    for epoch in range(epochs):
        model.train()
        for x, y in train_dl:
            optimizer.zero_grad()
            y_pred = model(x)
            loss = criterion(y_pred, y)
            loss.backward()
            optimizer.step()
        print(f'Epoch: {epoch}, Loss: {loss.item()}')
    return model


def ComplexModel(input_size: int) -> nn.Sequential:
    """Creates a Sequential model with 3 layers

    Args:
        input_size (int): input size of the model

    Returns:
        nn.Sequential: base model
    """
    return nn.Sequential(
        nn.Linear(input_size, 512),
        nn.ReLU(),
        nn.Linear(512, 10),
        nn.ReLU(),
        nn.Linear(10, 2),
        nn.Sigmoid()
    ).to(device)


def pred_check(pred: torch.Tensor, y: np.ndarray) -> float:
    """Gives the accuracy of the model in percentage

    Args:
        pred (torch.Tensor): predictions made by  the model
        y (np.ndarray): target data

    Returns:
        float: predict percentage
    """
    return (pred.argmax(dim=1) == torch.tensor(y).to(device)).sum().item() / len(y)


def trainer(X: np.ndarray, y: np.ndarray) -> None:
    """Trains the model with the given data
    Args:
        X (np.ndarray): Input data
        y (np.ndarray): Target data
    """

    # device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # y = np.array([[0, 1] if i == 1 else [1, 0] for i in y])

    x_train, x_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=22)
```

```python
    x_cv, x_test, y_cv, y_test = train_test_split(
        x_test, y_test, test_size=0.5, random_state=22)
    lambdas = np.array([0.001, 0.01, 0.05, 0.1, 0.2, 0.3])
    learning_rates = np.array([0.01, 0.1, 0.5, 1])
    best_score = 0
    best_params = (0, 0)

    for lambda_ in lambdas:
        for learning_rate in learning_rates:
            criterion = nn.CrossEntropyLoss().to(device)
            print(f'Lambda: {lambda_}, Learning rate: {learning_rate}')
            model = ComplexModel(x_train.shape[1])
            optimizer = optim.Adam(
                model.parameters(), lr=learning_rate, weight_decay=lambda_)
            train_dl = train_data(x_train, y_train)
            model = train_model(
                model, train_dl, criterion, optimizer, 20)
            pred = model(torch.tensor(x_cv, dtype=torch.float).to(device))

            score = pred_check(pred, y_cv)
            print(score)
            if score > best_score:
                best_score = score
                best_params = (lambda_, learning_rate)

    print(f'Best score: {best_score}')
    print(f'Best params: {best_params}')

    start = time.time()
    criterion = nn.CrossEntropyLoss().to(device)
    model = ComplexModel(X.shape[1])
    optimizer = optim.Adam(model.parameters(), lr=best_params[1],
                           weight_decay=best_params[0])
    train_dl = train_data(x_train, y_train)
    model = train_model(model, train_dl, criterion, optimizer, 20)
    end = time.time()

    test_score = pred_check(
        model(torch.tensor(x_test, dtype=torch.float).to(device)), y_test)
    cv_score = pred_check(
        model(torch.tensor(x_cv, dtype=torch.float).to(device)), y_cv)
    train_score = pred_check(
        model(torch.tensor(x_train, dtype=torch.float).to(device)), y_train)

    print(f'Test score: {test_score}')
    print(f'CV score: {cv_score}')
    print(f'Train score: {train_score}')
    print(f'Time: {end-start}')

    sio.savemat('res/pytorch.mat', {
        'test_score': test_score,
        'cv_score': cv_score,
        'train_score': train_score,
        'best_params': best_params,
        'time': end-start
    })
```

Código del entrenador de PolynomialTransformer:

```python
import time
from typing import Union
import numpy as np
import matplotlib.pyplot as plt
import sklearn.linear_model as lm
import sklearn.preprocessing as sp
import sklearn.model_selection as ms
import scipy.io as sio


def cost(y: np.ndarray, y_hat: np.ndarray) -> float:
    """Calculates the cost of the model

    Args:
        y (np.ndarray): real values
        y_hat (np.ndarray): predicted values

```

```python
    Returns:
        float: cost of the model
    """
    return np.mean((y_hat - y)**2) / 2


def train_reg(x_train: np.ndarray, y_train: np.ndarray, grado: int, l: float) -> tuple[sp.PolynomialFeatures, sp.StandardScaler, lm.Ridge, np.ndarray]:
    """ Trains a model given the training data with polynomial features and regularization

    Args:
        x_train (np.ndarray): x values of the training data
        y_train (np.ndarray): y values of the training data
        grado (int): degree of the polynomial
        l (float): lambda value for the regularization

    Returns:
        tuple[sp.PolynomialFeatures, sp.StandardScaler, lm.Ridge, np.ndarray]: _description_
    """
    poly: sp.PolynomialFeatures = sp.PolynomialFeatures(
        degree=grado, include_bias=False)
    x_train = poly.fit_transform(x_train)
    scal: sp.StandardScaler = sp.StandardScaler()
    # x_train = scal.fit_transform(x_train)
    model: lm.Ridge = lm.Ridge(alpha=l)
    model.fit(x_train, y_train)
    return poly, scal, model, x_train


def test(x_test: np.ndarray, y_test: np.ndarray, x_train_aux: np.ndarray, y_train: np.ndarray, poly: sp.PolynomialFeatures, scal: sp.StandardScaler, model: Union[lm.LinearRegression, lm.Ridge]) -> tuple[float, float]:
    """Tests the model with the test data

    Args:
        x_test (np.ndarray): x values of the test data
        y_test (np.ndarray): y values of the test data
        x_train_aux (np.ndarray): x values of the training data
        y_train (np.ndarray): y values of the training data
        poly (sp.PolynomialFeatures): polynomial features
        scal (sp.StandardScaler): standard scaler
        model (Union[lm.LinearRegression, lm.Ridge]): model to test

    Returns:
        tuple[float, float]: test cost, train cost
    """
    x_test = poly.transform(x_test)
    # x_test = scal.transform(x_test)

    y_pred_test: np.ndarray = model.predict(x_test)
    test_cost: float = cost(y_test, y_pred_test)

    y_pred_train: np.ndarray = model.predict(x_train_aux)
    train_cost: float = cost(y_train, y_pred_train)

    return test_cost, train_cost


def trainer(X: np.ndarray, y: np.ndarray) -> None:
    """Trains the model with the given data
    Args:
        X (np.ndarray): Input data
        y (np.ndarray): Target data
    """
    x_train, x_test, y_train, y_test = ms.train_test_split(
        X, y, test_size=0.3, random_state=22)
    x_cv, x_test, y_cv, y_test = ms.train_test_split(
        x_test, y_test, test_size=0.5, random_state=22)

    lambdas: list[float] = [1e-5, 1e-4, 1e-3,
                            1e-2, 1e-1, 1, 10, 100, 300, 600, 900]

    models: np.ndarray = np.empty((16, len(lambdas)), dtype=object)

    min_cost: float = -1
```

```python
    elec_lambda: float = 0
    eled_grado: int = 0
    costs = np.empty((16, len(lambdas)))

    for i in range(1, 16):
        for l in lambdas:
            pol, scal, model, x_train_aux = train_reg(x_train, y_train, i, l)
            models[i][lambdas.index(l)] = (pol, scal, model, x_train_aux)
            cv_cost, train_cost = test(
                x_cv, y_cv, x_train_aux, y_train, pol, scal, model)
            # costs[i][lambdas.index(l)] = cv_cost
            if min_cost == -1 or cv_cost < min_cost:
                min_cost = cv_cost
                elec_lambda = l
                eled_grado = i
            print(f"Grado: {i} Lambda: {l}-> Cost: {cv_cost}")
    print(f"Grado seleccionado: {eled_grado}")
    print(f"Lambda seleccionado: {elec_lambda}")

    start = time.time()
    pol, scal, model, x_train_aux = train_reg(
        x_train, y_train, eled_grado, elec_lambda)
    end = time.time()

    print(f"Tiempo de entrenamiento: {end-start}")
    X_train_aux = pol.transform(x_train)
    # X_train_aux = scal.transform(X_train_aux)
    y_pred = model.predict(X_train_aux)
    train_pred = (y_pred == y_train).sum() / len(y_train)
    print(f"Train pred: {train_pred}")
    X_cv_aux = pol.transform(x_cv)
    # X_cv_aux = scal.transform(X_cv_aux)
    y_pred = model.predict(X_cv_aux)
    cv_pred = (y_pred == y_cv).sum() / len(y_cv)
    print(f"CV pred: {cv_pred}")
    X_test_aux = pol.transform(x_test)
    # X_test_aux = scal.transform(X_test_aux)
    y_pred = model.predict(X_test_aux)
    test_pred = (y_pred == y_test).sum() / len(y_test)
    print(f"Test pred: {test_pred}")
    sio.savemat('res/poly.mat', {'train_score': train_pred,
                'cv_score': cv_pred, 'test_score': test_pred, 'best_params': (eled_grado,
    elec_lambda), 'time': end-start})
```

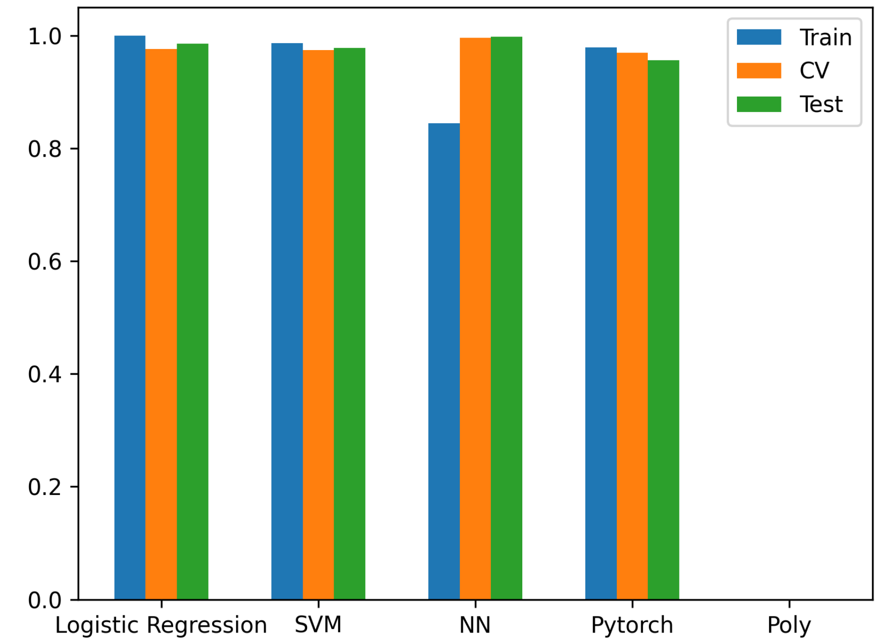

Figura 2.1: Resultados de los distintos modelos en precisión

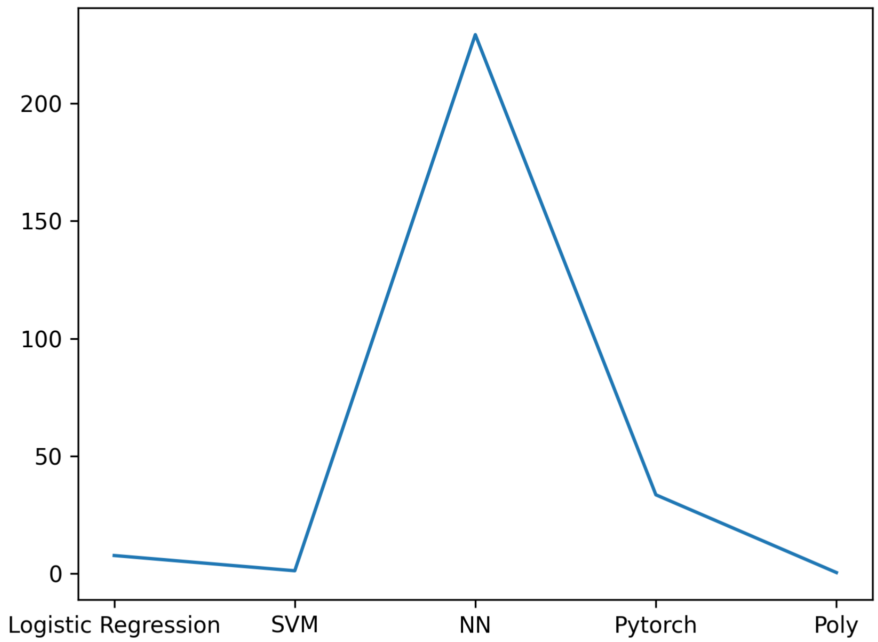

Figura 2.2: Tiempo de entrenamiento de los distintos modelos