Entrega 6: diseño de redes neuronales

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 6 y el proceso de diseño de redes neuronales.

Para esta práctica se usarán los siguientes *imports* vistos en la figura 0.1. Parte del código se reutiliza de la práctica anterior.

```
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 commandline  # Custom command line parser
import os
import sys
```

Figura 0.1: Código de las bibliotecas usadas

También usaremos una serie de constantes para todo el programa (figura 0.2).

```
# Constants
# Path to save the plots
plot_folder = "./memoria/images"
# Path to save the csv files
csv_folder = "./memoria/csv"
# Random state for reproducibility
RANDOM_STATE = 1
```

Figura 0.2: Constantes del programa

El *dataset* para esta práctica lo generamos aleatoriamente con la función *gen_data* (figura 0.4). El dataset se compone de una linea de datos "ideales" y datos con ruido para comprobar la eficacia de la red neuronal.

Para dibujar estos datos usaremos la función *plot_dataset* (figura 0.5).

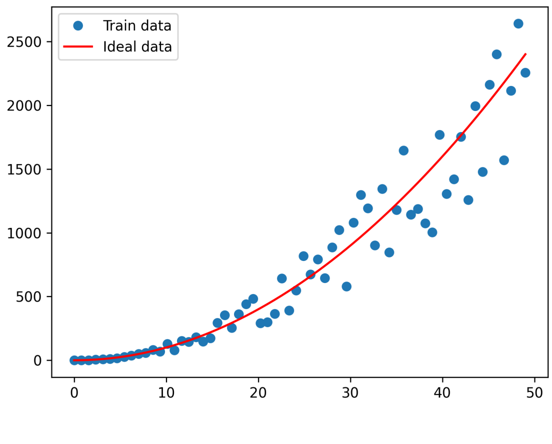

Figura 0.3: Ejemplo de los dígitos del *dataset*

```python
def gen_data(m: int, seed: int = 1, scale: float = 0.7) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """Generates a dataset with noise

    Args:
        m (int): number of samples
        seed (int, optional): random seed. Defaults to 1.
        scale (float, optional): scale of the noise. Defaults to 0.7.

    Returns:
        tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]: x_train, y_train, x_ideal, y_ideal
    """
    c: int = 0
    x_train: np.ndarray = np.linspace(0, 49, m)
    np.random.seed(seed)
    y_ideal: np.ndarray = x_train**2 + c
    y_train: np.ndarray = y_ideal + scale * \
        y_ideal * (np.random.sample((m,)) - 0.5)
    x_ideal: np.ndarray = x_train
    return x_train, y_train, x_ideal, y_ideal
```

Figura 0.4: Función *gen_data*

```python
def plot_dataset(x: np.ndarray, y: np.ndarray, x_ideal: np.ndarray, y_ideal: np.ndarray, name: str) -> None:
    """Plots the dataset and the ideal data

    Args:
        x (np.ndarray): x values of the dataset with noise
        y (np.ndarray): y values of the dataset with noise
        x_ideal (np.ndarray): x ideal values of the dataset
        y_ideal (np.ndarray): y ideal values of the dataset
        name (str): name of the file
    """
    plt.plot(x, y, 'o', label='Train data')
    plt.plot(x_ideal, y_ideal, label='Ideal data', c='red')
    plt.legend()
    plt.savefig(f'{plot_folder}/{name}.png', dpi=300)
    plt.clf()
```

Figura 0.5: Función *plot_dataset*

1. Sobreajuste a los ejemplos de entrenamiento

En este apartado vamos a analizar lo que ocurre si tienes pocos datos de test, haciendo así que el problema se amolde solo a los datos de entrenamiento y no consiga generalizar.

Para ello usaremos la función *overfitting* (figura 1.2). Dentro de esta función primero hacemos una separación de los datos para dejar un porcentaje del 67 % a los datos de entrenamiento y un 33 % a los datos de entrenamiento. Seguido esto, utilizamos las funciones *train*(1.3) para entrenar el modelo lineal y *test*(1.4) para sacar los costes (función *cost*, figura 1.5) de ambos conjuntos de datos. La figura 1.1 nos muestra en gráfica el sobreajuste producido por esta función. Este gráfico se genera con la función *plot_linear_data* (figura 1.6).

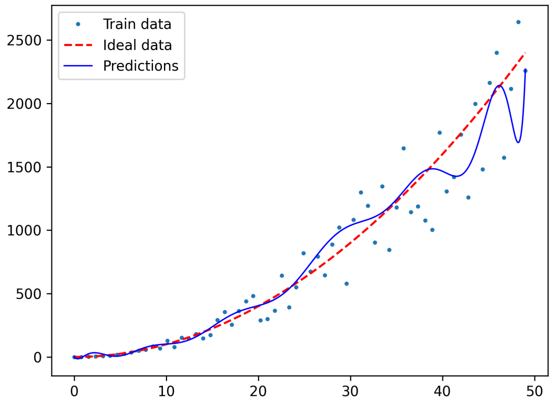

Figura 1.1: Gráfica del sobreajuste

```
def overfitting(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray) -> None:
    """Tests the overfitting of the model

    Args:
        x (np.ndarray): x values of the dataset with noise
        y (np.ndarray): y values of the dataset with noise
        x_i (np.ndarray): x ideal values of the dataset
        y_i (np.ndarray): y ideal values of the dataset
    """
    print("Overfitting")
    x_train, x_test, y_train, y_test = ms.train_test_split(
        x, y, test_size=0.33, random_state=RANDOM_STATE)
    pol, scal, model, x_train_aux = train(x_train, y_train, 15)
    range_x: np.ndarray = np.linspace(np.min(x), np.max(x), 1000)
    range_x = range_x[:, None]
    range_x_p: np.ndarray = pol.transform(range_x)
    range_x_p = scal.transform(range_x_p)
    y_pred: np.ndarray = model.predict(range_x_p)
    plot_linear_data(x, y, x_i, y_i, range_x, y_pred, 'overfitting')
    test_cost, train_cost = test(
        x_test, y_test, x_train_aux, y_train, pol, scal, model)

    print(f"Train cost: {train_cost}")
    print(f"Test cost: {test_cost}")
```

Figura 1.2: Función *overfitting*

```python
def train(x_train: np.ndarray, y_train: np.ndarray, grado: int) -> tuple[sp.PolynomialFeatures
    , sp.StandardScaler, lm.LinearRegression, np.ndarray]:
    """ Trains a model given the training data with polynomial features

    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

    Returns:
        tuple[sp.PolynomialFeatures, sp.StandardScaler, lm.LinearRegression, np.ndarray]:
    _description_
    """
    poly: sp.PolynomialFeatures = sp.PolynomialFeatures(
        degree=grado, include_bias=False)
    x_train = poly.fit_transform(x_train[:, None])
    scal: sp.StandardScaler = sp.StandardScaler()
    x_train = scal.fit_transform(x_train)
    model: lm.LinearRegression = lm.LinearRegression()
    model.fit(x_train, y_train)
    return poly, scal, model, x_train
```

Figura 1.3: Función *train*

```python
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[:, None])
    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
```

Figura 1.4: Función *test*

```python
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

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

Figura 1.5: Función *cost*

```python
def plot_linear_data(x: np.ndarray, y: np.ndarray, x_ideal: np.ndarray, y_ideal: np.ndarray,
    model_range: np.ndarray, model: np.ndarray, name: str) -> None:
    """Plots the dataset, the ideal data and the model

    Args:
        x (np.ndarray): x values of the dataset with noise
        y (np.ndarray): y values of the dataset with noise
        x_ideal (np.ndarray): x ideal values of the dataset
        y_ideal (np.ndarray): y ideal values of the dataset
        model_range (np.ndarray): x values of the model
        model (np.ndarray): y values of the model
        name (str): file name
    """
    plt.plot(x, y, 'o', label='Train data', markersize=2)
    plt.plot(x_ideal, y_ideal, label='Ideal data',
             c='red', linestyle='dashed', linewidth=1.5)
    plt.plot(model_range, model, label='Predictions', c='blue', linewidth=1)
    plt.legend()
    plt.savefig(f'{plot_folder}/{name}.png', dpi=300)
    plt.clf()
```

Figura 1.6: Función *plot_linear_data*

2. Elección del grado del polinomio usando un conjunto de validación

En este apartado vamos a escoger el grado del polinomio basándonos en el menor coste de validación de grados entre 1 y 10. Primero dividimos los datos en 60 % de entrenamiento, 20 % de validación y 20 % de test. Entrenamos el modelo con las funciones del apartado anterior (1.3, 1.4) y escogemos el grado que menos coste de validación nos de.

La función que lleva todo este proceso es *seleccion_grado* (figura 2.2). La figura 2.1 nos muestra en la comparativa entre el modelo del grado escogido y la recta ideal. El grado escogido es 2.

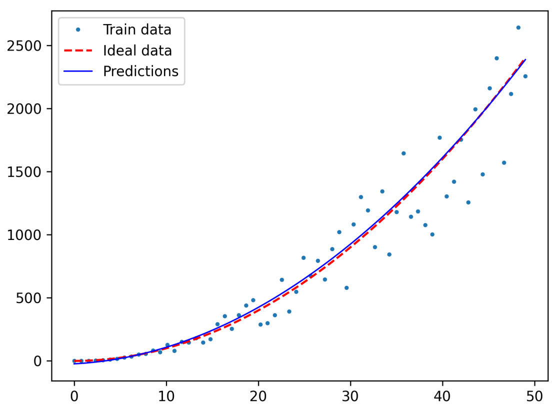

Figura 2.1: Gráfica de la selección del grado

```python
def seleccion_grado(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray) -> None:
    """Selects the best degree for the model

    Args:
        x (np.ndarray): x values of the dataset with noise
        y (np.ndarray): y values of the dataset with noise
        x_i (np.ndarray): x ideal values of the dataset
        y_i (np.ndarray): y ideal values of the dataset
    """
    print("Seleccion de grado")
    x_train, x_test, y_train, y_test = ms.train_test_split(
        x, y, test_size=0.4, random_state=RANDOM_STATE)
    x_test, x_cv, y_test, y_cv = ms.train_test_split(
        x_test, y_test, test_size=0.5, random_state=RANDOM_STATE)
    min_cost: float = 0
    min_grado: float = 0
    models: np.ndarray = np.empty(10, dtype=object)
    for grado in range(10):
        pol, scal, model, x_train_aux = train(x_train, y_train, grado + 1)
        cv_cost, train_cost = test(
            x_cv, y_cv, x_train_aux, y_train, pol, scal, model)
        models[grado] = (pol, scal, model, x_train_aux)
        if min_cost == 0 or cv_cost < min_cost:
            min_cost = cv_cost
            min_grado = grado + 1
    print(f"Grado seleccionado: {min_grado}")

    x_range: np.ndarray = np.linspace(np.min(x), np.max(x), 1000)
    x_range: np.ndarray = x_range[:, None]
    x_range_p: np.ndarray = models[min_grado - 1][0].transform(x_range)
    x_range_p: np.ndarray = models[min_grado - 1][1].transform(x_range_p)
    y_pred: np.ndarray = models[min_grado - 1][2].predict(x_range_p)

    plot_linear_data(x, y, x_i, y_i, x_range, y_pred, 'grado')

    test_cost, train_cost = test(
        x_test, y_test, models[min_grado - 1][3], y_train, models[min_grado - 1][0], models[min_grado - 1][1], models[min_grado - 1][2])
    print(f"Train cost: {train_cost}")
    print(f"CV cost: {min_cost}")
    print(f"Test cost: {test_cost}")
```

Figura 2.2: Función *seleccion_grado*

3. Elección del parámetro λ

Como en este apartado usamos regularización, incluimos la funcion *train_reg* (figura 3.2) para entrenar el modelo con regularización, la función de test se mantiene igual. La función *seleccion_lambda* (figura 3.3) nos ayudará a escoger el mejor λ entre [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100, 300, 600, 900]. La figura 3.1 nos muestra la comparativa entre el modelo con el λ escogido y la recta ideal. El λ escogido es 10.

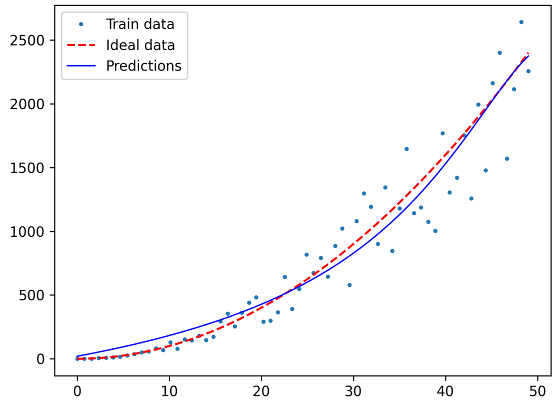

Figura 3.1: Gráfica de la selección de λ

```
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[:, None])
    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
```

Figura 3.2: Función *train_reg*

```
def seleccion_lambda(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray) -> None:
    """Selects the best lambda for the model

    Args:
        x (np.ndarray): x values of the dataset with noise
        y (np.ndarray): y values of the dataset with noise
        x_i (np.ndarray): x ideal values of the dataset
        y_i (np.ndarray): y ideal values of the dataset
    """
    print("Seleccion de lambda")
    lambdas: list[float] = [1e-6, 1e-5, 1e-4, 1e-3,
                            1e-2, 1e-1, 1, 10, 100, 300, 600, 900]
    alpha: float = 0
    min_cost: float = -1
    x_train, x_test, y_train, y_test = ms.train_test_split(
        x, y, test_size=0.4, random_state=RANDOM_STATE)
    x_test, x_cv, y_test, y_cv = ms.train_test_split(
        x_test, y_test, test_size=0.5, random_state=RANDOM_STATE)
    models: np.ndarray = np.empty(len(lambdas), dtype=object)

    for l in lambdas:
        pol, scal, model, x_train_aux = train_reg(x_train, y_train, 15, l)
        test_cost, train_cost = test(
            x_cv, y_cv, x_train_aux, y_train, pol, scal, model)
        models[lambdas.index(l)] = (pol, scal, model, x_train_aux)
        if min_cost == -1 or test_cost < min_cost:
            min_cost = test_cost
            alpha = l
        print(f"Lambda: {l}-> Cost: {test_cost}")
    print(f"Lamda seleccionado: {alpha}")

    x_range: np.ndarray = np.linspace(np.min(x), np.max(x), 1000)
    x_range = x_range[:, None]
    x_range_p: np.ndarray = models[lambdas.index(alpha)][0].transform(x_range)
    x_range_p = models[lambdas.index(alpha)][1].transform(x_range_p)
    y_pred: np.ndarray = models[lambdas.index(alpha)][2].predict(x_range_p)

    plot_linear_data(x, y, x_i, y_i, x_range, y_pred, 'lambda')

    test_cost, train_cost = test(
        x_test, y_test, models[lambdas.index(alpha)][3], y_train, models[lambdas.index(alpha)][0], models[lambdas.index(alpha)][1], models[lambdas.index(alpha)][2])
    print(f"Train cost: {train_cost}")
    print(f"CV cost: {min_cost}")
    print(f"Test cost: {test_cost}")
```

Figura 3.3: Función *seleccion_lambda*

4. Elección de hiperparámetros

En este apartado vamos a escoger grado y regularización mirando el mínimo coste de validación. Para ello usaremos los posibles parámetros de los dos apartados anteriores. La función *seleccion_hiperparametros* (figura 4.2) nos ayudará a escoger el mejor grado y λ. La figura 4.1 nos muestra la comparativa entre el modelo con los hiperparámetros escogidos y la recta ideal. Los hiperparámetros escogidos son grado 12 y λ 1e-6.

Podemos ver los resultados por λ y grado en las tablas 4.1 4.2 y 4.3.

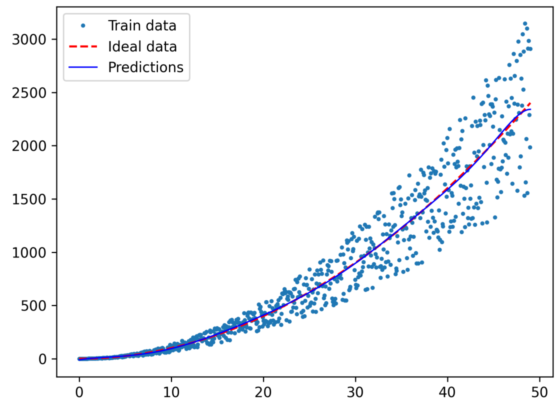

Figura 4.1: Gráfica de la selección de hiperparámetros

0.00000e+00	1.00000e+00	2.00000e+00	3.00000e+00	4.00000e+00
1.00000e-06	6.94186e-310	3.73427e+04	2.00062e+04	2.00069e+04
1.00000e-05	6.94186e-310	3.73427e+04	2.00062e+04	2.00069e+04
1.00000e-04	5.43817e-310	3.73427e+04	2.00062e+04	2.00069e+04
1.00000e-03	5.43817e-310	3.73426e+04	2.00062e+04	2.00069e+04
1.00000e-02	5.88785e+00	3.73419e+04	2.00060e+04	2.00073e+04
1.00000e-01	4.35701e+01	3.73343e+04	2.00046e+04	2.00140e+04
1.00000e+00	1.71402e+01	3.72597e+04	2.00104e+04	2.00877e+04
1.00000e+01	5.49533e+00	3.66353e+04	2.06875e+04	2.01468e+04
1.00000e+02	3.82710e+01	3.87107e+04	2.41588e+04	2.08719e+04
3.00000e+02	5.56075e+00	6.21938e+04	3.48534e+04	2.68364e+04
6.00000e+02	3.70935e+01	9.73603e+04	5.62252e+04	4.08061e+04
9.00000e+02	4.34393e+01	1.23253e+05	7.65306e+04	5.58958e+04

Cuadro 4.1: Resultados de los hiperparámetros

0.00000e+00	5.00000e+00	6.00000e+00	7.00000e+00	8.00000e+00
1.00000e-06	2.00187e+04	2.00036e+04	2.00037e+04	2.00015e+04
1.00000e-05	2.00187e+04	2.00038e+04	2.00029e+04	2.00026e+04
1.00000e-04	2.00185e+04	2.00053e+04	2.00024e+04	2.00020e+04
1.00000e-03	2.00171e+04	2.00116e+04	2.00048e+04	2.00020e+04
1.00000e-02	2.00093e+04	2.00118e+04	2.00104e+04	2.00063e+04
1.00000e-01	2.00036e+04	1.99979e+04	1.99980e+04	2.00001e+04
1.00000e+00	2.00724e+04	2.00344e+04	2.00135e+04	2.00049e+04
1.00000e+01	2.03006e+04	2.03256e+04	2.02652e+04	2.01927e+04
1.00000e+02	2.09946e+04	2.16251e+04	2.20570e+04	2.22388e+04
3.00000e+02	2.51627e+04	2.56445e+04	2.66362e+04	2.75402e+04
6.00000e+02	3.51190e+04	3.36068e+04	3.38446e+04	3.47103e+04
9.00000e+02	4.66041e+04	4.27799e+04	4.16433e+04	4.17849e+04

Cuadro 4.2: Resultados de los hiperparámetros

0.00000e+00	9.00000e+00	1.00000e+01	1.10000e+01	1.20000e+01
1.00000e-06	1.99926e+04	1.99706e+04	1.99346e+04	1.99032e+04
1.00000e-05	1.99986e+04	1.99872e+04	1.99680e+04	1.99431e+04
1.00000e-04	2.00011e+04	1.99973e+04	1.99879e+04	1.99720e+04
1.00000e-03	2.00008e+04	1.99993e+04	1.99964e+04	1.99909e+04
1.00000e-02	2.00023e+04	1.99994e+04	1.99968e+04	1.99938e+04
1.00000e-01	2.00016e+04	2.00013e+04	1.99994e+04	1.99962e+04
1.00000e+00	2.00019e+04	2.00014e+04	2.00021e+04	2.00035e+04
1.00000e+01	2.01386e+04	2.01060e+04	2.00893e+04	2.00820e+04
1.00000e+02	2.22515e+04	2.21748e+04	2.20634e+04	2.19493e+04
3.00000e+02	2.81917e+04	2.85895e+04	2.87844e+04	2.88343e+04
6.00000e+02	3.56910e+04	3.65691e+04	3.72692e+04	3.77821e+04
9.00000e+02	4.24743e+04	4.33353e+04	4.41806e+04	4.49245e+04

Cuadro 4.3: Resultados de los hiperparámetros

```python
def seleccion_hiperparametros(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray)
    -> None:
    """Selects the best hyperparameters for the model

    Args:
        x (np.ndarray): x values of the dataset with noise
        y (np.ndarray): y values of the dataset with noise
        x_i (np.ndarray): x ideal values of the dataset
        y_i (np.ndarray): y ideal values of the dataset
    """
    print("Seleccion de hiperparametros")
    x_train, x_test, y_train, y_test = ms.train_test_split(
        x, y, test_size=0.4, random_state=RANDOM_STATE)
    x_test, x_cv, y_test, y_cv = ms.train_test_split(
        x_test, y_test, test_size=0.5, random_state=RANDOM_STATE)

    lambdas: list[float] = [1e-6, 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
    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}")

    costs = np.append(np.array(lambdas)[None, :], costs, axis=0)
    costs = np.append(np.array(range(0, 17))[None, :], costs.T, axis=0)
    write_csv(costs, 'hiperparametros')

    x_range: np.ndarray = np.linspace(np.min(x), np.max(x), 10000)
    x_range = x_range[:, None]
    x_range_p: np.ndarray = models[eled_grado][lambdas.index(
        elec_lambda)][0].transform(x_range)
    x_range_p = models[eled_grado][lambdas.index(
        elec_lambda)][1].transform(x_range_p)
    y_pred: np.ndarray = models[eled_grado][lambdas.index(
        elec_lambda)][2].predict(x_range_p)

    plot_linear_data(x, y, x_i, y_i, x_range, y_pred,
                     'hiperparametros')

    test_cost, train_cost = test(
        x_test, y_test, models[eled_grado][lambdas.index(elec_lambda)][3], y_train, models[
    eled_grado][lambdas.index(elec_lambda)][0], models[eled_grado][lambdas.index(elec_lambda)
    ][1], models[eled_grado][lambdas.index(elec_lambda)][2])
    print(f"Train cost: {train_cost}")
    print(f"CV cost: {min_cost}")
    print(f"Test cost: {test_cost}")
```

Figura 4.2: Función *seleccion_hiperparametros*

5. Curvas de aprendizaje

En este apartado vamos a hacer una comparativa de coste de validación y de entrenamiento variando el tamaño de muestra de entrenamiento. Para ello usaremos la función *learing_curve* (figura 5.2). La figura 5.1

nos muestra la comparativa entre el coste de validación y de entrenamiento. Para pintar la gráfica se ha usado la función *draw_learning_curve* (figura 5.3).

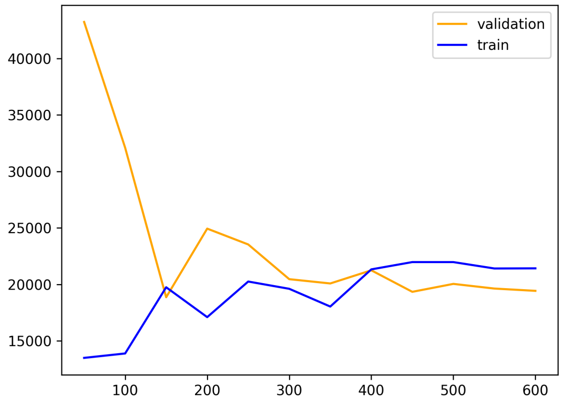

Figura 5.1: Gráfica de las curvas de aprendizaje

```python
def learning_curve():
    """Plots the learning curve of the model based on the number of training sample sizes
    """
    print("Learning curve")
    J_cost_train: list[float] = []
    J_const_val: list[float] = []
    params: list[int] = range(50, 601, 50)
    x, y, x_i, y_i = gen_data(1000)
    x_train_t, x_test, y_train_t, y_test = ms.train_test_split(
        x, y, test_size=0.4, random_state=RANDOM_STATE)
    x_test, x_cv, y_test, y_cv = ms.train_test_split(
        x_test, y_test, test_size=0.5, random_state=RANDOM_STATE)

    for i in range(len(params)):
        indexes = np.linspace(
            0, len(x_train_t) - 1, params[i], dtype=int)
        x_train = x_train_t[indexes]
        y_train = y_train_t[indexes]
        pol, scal, model, x_train_aux = train(x_train, y_train, 16)
        cv_cost, train_cost = test(
            x_cv, y_cv, x_train_aux, y_train, pol, scal, model)
        J_cost_train.append(train_cost)
        J_const_val.append(cv_cost)

    draw_learning_curve(params, J_cost_train, params, J_const_val)

    print(f'Mejor valor de entrenamiento: {np.min(J_cost_train)}')
    print(f'Mejor valor de validacion: {np.min(J_const_val)}')
```

Figura 5.2: Función *learning_curve*

```python
def draw_learning_curve(x: np.ndarray, y: np.ndarray, x_v: np.ndarray, y_v: np.ndarray):
    """Plots the learning curve of the model based on the number of training sample sizes

    Args:
        x (np.ndarray): number of samples
        y (np.ndarray): cost of the training set
        x_v (np.ndarray): number of samples
        y_v (np.ndarray): cost of the validation set
    """
    plt.figure()
    plt.plot(x_v, y_v, c='orange', label='validation')
    plt.plot(x, y, c='blue', label='train')
    plt.legend()
    plt.savefig(f'{plot_folder}/learning_curve.png', dpi=300)
```

Figura 5.3: Función *draw_learning_curve*

6. Utilidades

Algunas utilidades que se han usado en el código son las siguientes:

- *test_overfitting* (figura 6.1): Función que testea el sobreajuste.
- *test_seleccion_grado* (figura 6.2): Función que testea la selección del grado.
- *test_regularización* (figura 6.3): Función que testea la selección de λ.
- *test_hiperparametros* (figura 6.4): Función que testea la selección de hiperparámetros.
- *test_learning_curve* (figura 6.5): Función que testea las curvas de aprendizaje.
- *write_csv* (figura 6.6): Función que escribe los datos en un archivo *csv*.
- *main* (figura 6.7): Función que ejecuta todas las pruebas.
- *CommandLine* (figura **??**): Clase que nos permite añadirle parámetros a la ejecución del programa.

```python
def test_overfitting(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray,
    commandLine: commandline.CommandLine) -> None:
    """Tests the overfitting of the model

    Args:
        x (np.ndarray): x values of the dataset with noise
        y (np.ndarray): y values of the dataset with noise
        x_i (np.ndarray): x ideal values of the dataset
        y_i (np.ndarray): y ideal values of the dataset
        commandLine (commandline.CommandLine): command line arguments
    """
    if (not os.path.exists(f'{plot_folder}/overfitting.png') or commandLine.overfitting or
    commandLine.all):
        overfitting(x, y, x_i, y_i)
    else:
        if (commandLine.interactive):
            answer = ''
            while (answer != 'y' and answer != 'n'):
                print("Recreate overfitting test? [y/n]")
                answer = input()
            if (answer == 'y'):
                overfitting(x, y, x_i, y_i)
```

Figura 6.1: Función *test_overfitting*

```python
def test_seleccion_grado(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray, commandLine: commandline.CommandLine) -> None:
    """Selects the best degree for the model
    """
    if (not os.path.exists(f'{plot_folder}/grado.png') or commandLine.grado or commandLine.all):
        seleccion_grado(x, y, x_i, y_i)
    else:
        if (commandLine.interactive):
            answer = ''
            while (answer != 'y' and answer != 'n'):
                print("Recreate grado test? [y/n]")
                answer = input()
            if (answer == 'y'):
                seleccion_grado(x, y, x_i, y_i)
```

Figura 6.2: Función *test_seleccion_grado*

```python
def test_regularizacion(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray, commandLine: commandline.CommandLine) -> None:
    """Selects the best lambda for the model
    """
    if (not os.path.exists(f'{plot_folder}/lambda.png') or commandLine.reg or commandLine.all):
        seleccion_lambda(x, y, x_i, y_i)
    else:
        if (commandLine.interactive):
            answer = ''
            while (answer != 'y' and answer != 'n'):
                print("Recreate regularizacion test? [y/n]")
                answer = input()
            if (answer == 'y'):
                seleccion_lambda(x, y, x_i, y_i)
```

Figura 6.3: Función *test_regularizacion*

```python
def test_hiperparametros(x: np.ndarray, y: np.ndarray, x_i: np.ndarray, y_i: np.ndarray, commandLine: commandline.CommandLine) -> None:
    """Selects the best hyperparameters for the model
    """
    if (not os.path.exists(f'{plot_folder}/hiperparametros.png') or commandLine.hyperparam or commandLine.all):
        seleccion_hiperparametros(x, y, x_i, y_i)
    else:
        if (commandLine.interactive):
            answer = ''
            while (answer != 'y' and answer != 'n'):
                print("Recreate hiperparametros test? [y/n]")
                answer = input()
            if (answer == 'y'):
                seleccion_hiperparametros(x, y, x_i, y_i)
```

Figura 6.4: Función *test_hiperparametros*

```python
def test_learning_curve(x: np.ndarray, i: np.ndarray, x_i: np.ndarray, y_i: np.ndarray,
    commandLine: commandline.CommandLine) -> None:
    """Plots the learning curve of the model based on the number of training sample sizes
    """
    if (not os.path.exists(f'{plot_folder}/learning_curve.png') or commandLine.learning_curve
    or commandLine.all):
        learning_curve()
    else:
        if (commandLine.interactive):
            answer = ''
            while (answer != 'y' and answer != 'n'):
                print("Recreate learning curve test? [y/n]")
                answer = input()
            if (answer == 'y'):
                learning_curve()
```

Figura 6.5: Función *test_learning_curve*

```python
def write_csv(data: np.ndarray, name: str, split: int = 4) -> None:
    """Writes a numpy array to a csv file

    Args:
        data (np.ndarray): data to write
        name (str): name of the file
        split (int, optional): number of columns to split the data. Defaults to 4.
    """
    np.printoptions(suppress=False)
    # Format string to display numbers in scientific notation with 2 decimal places
    fmt_float = '%.5e'
    for i in range(1, len(data), split):
        data_acc = np.concatenate((data[:, :1], data[:, i:i+split]), axis=1)

        filename = f'{csv_folder}/{name}{i//split}.csv'
        np.savetxt(filename, data_acc, delimiter=',', fmt=fmt_float)
```

Figura 6.6: Función *write_csv*

```python
def main() -> None:
    """Main function
    """
    commandLine = commandline.CommandLine()
    commandLine.parse(sys.argv[1:])
    x, y, x_i, y_i = gen_data(64)
    funcs = [test_overfitting, test_seleccion_grado,
             test_regularizacion, test_hiperparametros, test_learning_curve]

    prepare_folder()
    if not os.path.exists(f'{plot_folder}/dataset.png'):
        plot_dataset(x, y, x_i, y_i, 'dataset')

    for func in funcs:
        if func.__name__ == 'test_hiperparametros':
            x, y, x_i, y_i = gen_data(750)
        func(x, y, x_i, y_i, commandLine)
```

Figura 6.7: Función *main*

```python
def prepare_folder() -> None:
    """Creates the folder to save the plots
    """
    if not os.path.exists(plot_folder):
        os.makedirs(plot_folder)
```

Figura 6.8: Función *prepare_folder*

```python
import argparse


class CommandLine:
    interactive: bool = False
    overfitting: bool = False
    grado: bool = False
    reg: bool = False
    hyperparam: bool = False
    learning_curve: bool = False
    all: bool = False

    def __init__(self):
        self.parser = argparse.ArgumentParser(
            description='Practica 6 - Aprendizaje Automatico')
        self.parser.add_argument('-I', "--Interactive",
                                 help='Interactive mode', required=False, default="", action='
    store_true')
        self.parser.add_argument('-O', "--Overfitting", help='runs the Overfitting test',
                                 required=False, default="", action='store_true')
        self.parser.add_argument(
            '-G', "--Grado", help='Grado', required=False, default="", action='store_true')
        self.parser.add_argument('-R', "--Regularization", help='runs the Regularization test'
    ,
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-HP', "--Hyperparam", help='runs the Hyperparam test',
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-L', "--LearningCurve", help='runs the LearningCurve test',
                                 required=False, default="", action='store_true')
        self.parser.add_argument('-A', "--All", help='runs all tests',
                                 required=False, default="", action='store_true')

    def parse(self, sysargs):
        args = self.parser.parse_args(sysargs)
        if args.Interactive:
            self.interactive = True
        if args.Overfitting:
            self.overfitting = True
        if args.Grado:
            self.grado = True
        if args.Regularization:
            self.reg = True
        if args.Hyperparam:
            self.hyperparam = True
        if args.LearningCurve:
            self.learning_curve = True
        if args.All:
            self.all = True
```

Figura 6.9: Clase *CommandLine*