

# 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:

```

1 import sklearn.svm as svm
2 import scipy.io as sio
3 from sklearn.model_selection import train_test_split
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import os
7 from utils_p7 import email2TokenList, getVocabDict
8 import codecs
9 import SVM_Trainer
10 import Logic_Regression_Trainer
11 import nn_trainer
12 import pytorch_trainer
13 import Poly_trainer
14 import torch.nn as nn
15 import torch.optim as optim
16 import torch
17 from sklearn.metrics import accuracy_score
18 from pytorch_trainer import ComplexModel, train_data, train_model, pred_check, device
19
20 plot_folder: str = 'memoria/images'
21
22
23 def load_data(file: str) -> tuple[np.ndarray, np.ndarray]:
24     """Loads the data from a .mat file
25
26     Args:
27         file (str): name of the file
28
29     Returns:
30         tuple[np.ndarray, np.ndarray]: X and y data
31     """
32     data = sio.loadmat(file)
33     X = data['X']
34     y = data['y']
35     return X, y
36
37
38 def load_data3(file: str) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
39     """Loads the data from a .mat file
40
41     Args:
42         file (str): name of the file
43
44     Returns:
45         tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]: X, y, Xval, yval data
46     """
47     data = sio.loadmat(file)
48     X = data['X']
49     y = data['y']
50     Xval = data['Xval']
51     yval = data['yval']
52     return X, y, Xval, yval
53
54
55 def kernel_linear(X: np.ndarray, y: np.ndarray, C: float) -> None:
56     """Linear kernel
57
58     Args:
59         X (np.ndarray): X train dataa
60         y (np.ndarray): y train data
61         C (float): regularization parameter
62     """
63     svm_lineal: svm.SVC = svm.SVC(kernel='linear', C=C)
64     svm_lineal.fit(X, y.ravel())
65     x1: np.ndarray = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)

```

```

66 x2: np.ndarray = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
67 X1, X2 = np.meshgrid(x1, x2)
68 yp: np.ndarray = svm_lineal.predict(
69     np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape)
70 plt.contour(X1, X2, yp, colors='darkgreen', linewidths=1)
71 plt.scatter(X[y.ravel() == 1, 0], X[y.ravel() == 1, 1], c='b', marker='x')
72 plt.scatter(X[y.ravel() == 0, 0], X[y.ravel() == 0, 1], c='y', marker='o')
73 plt.xticks(np.arange(0, 5.5, 0.5))
74 plt.yticks(np.arange(1.5, 5.5, 0.5))
75 plt.savefig(f'{plot_folder}/SVM_lineal_c{C}.png', dpi=300)
76
77
78 def kerner_gaussiano(X: np.ndarray, y: np.ndarray, C: float, sigma: float) -> None:
79     """Gaussian kernel
80     Args:
81         X (np.ndarray): X train dataa
82         y (np.ndarray): y train data
83         C (float): regularization parameter
84         sigma (float): scale parameter
85     """
86     svm_gauss: svm.SVC = svm.SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
87     svm_gauss.fit(X, y.ravel())
88     x1: np.ndarray = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
89     x2: np.ndarray = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
90     X1, X2 = np.meshgrid(x1, x2)
91     yp: np.ndarray = svm_gauss.predict(
92         np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape)
93     plt.contour(X1, X2, yp, colors='darkgreen', linewidths=1)
94     plt.scatter(X[y.ravel() == 1, 0], X[y.ravel() == 1, 1], c='b', marker='x')
95     plt.scatter(X[y.ravel() == 0, 0], X[y.ravel() == 0, 1], c='y', marker='o')
96     plt.xticks(np.arange(0.0, 1.2, 0.2))
97     plt.yticks(np.arange(0.4, 1.1, 0.1))
98     plt.savefig(f'{plot_folder}/SVM_gauss_c{C}_sigma{sigma}.png', dpi=300)
99
100
101 def seleccion_sigma_C() -> None:
102     """Selects the best C and sigma for the gaussian kernel
103     """
104     X, y, Xval, yval = load_data3('data/ex6data3.mat')
105     C_values: list[float] = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
106     sigma_values: list[float] = C_values
107     best_score: float = 0
108     best_params: tuple[float] = (0, 0)
109     for C in C_values:
110         for sigma in sigma_values:
111             svm_gauss = svm.SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
112             svm_gauss.fit(X, y.ravel())
113             score = svm_gauss.score(Xval, yval)
114             if score > best_score:
115                 best_score = score
116                 best_params = (C, sigma)
117     print(f'Best score: {best_score}')
118     print(f'Best params: {best_params}')
119     svm_gauss: svm.SVC = svm.SVC(
120         kernel='rbf', C=best_params[0], gamma=1/(2*best_params[1]**2))
121     svm_gauss.fit(X, y.ravel())
122     x1: np.ndarray = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
123     x2: np.ndarray = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
124     X1, X2 = np.meshgrid(x1, x2)
125     yp: np.ndarray = svm_gauss.predict(
126         np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape)
127     plt.contour(X1, X2, yp, colors='darkgreen', linewidths=1)
128     plt.scatter(X[y.ravel() == 1, 0], X[y.ravel() == 1, 1], c='b', marker='x')
129     plt.scatter(X[y.ravel() == 0, 0], X[y.ravel() == 0, 1], c='y', marker='o')
130     plt.yticks(np.arange(-0.8, 0.7, 0.2))
131     plt.xticks(np.arange(-0.6, 0.4, 0.1))
132     plt.savefig(f'{plot_folder}/SVM_gauss_best.png', dpi=300)
133
134
135 def apartado_A() -> None:
136     """Apartado A
137     """
138     X, y = load_data('data/ex6data1.mat')
139     print("Linear kernel with C=1")
140     kernel_linear(X, y, 1.0)
141     plt.clf()

```

```

142 print("Linear kernel with C=100")
143 kernel_linear(X, y, 100.0)
144 X, y = load_data('data/ex6data2.mat')
145 plt.clf()
146 print("Gaussian kernel with C=1 and sigma=0.1")
147 kerner_gaussiano(X, y, 1.0, 0.1)
148 plt.clf()
149 print("Selecting C and sigma for gaussian kernel")
150 seleccion_sigma_C()
151
152
153 def load_data_spam() -> list[tuple[list[str], int]]:
154     """Loads the spam data
155     """
156     modes: list[str] = ['spam', 'easy_ham', 'hard_ham']
157     cantidades: list[int] = [500, 2551, 250]
158     spam_flag = [1, 0, 0]
159     correos = []
160     for mode, number, spam in zip(modes, cantidades, spam_flag):
161         progress = 0
162         length = 50
163         for file in range(1, number + 1):
164             file = str(file)
165             with codecs.open(f'./data_spam/spam/{mode}/{file.zfill(4)}.txt', 'r', encoding=
'utf-8', errors='ignore') as f:
166                 progress += 1
167                 bar_length = int(length * progress / number)
168                 bar = '[' + '=' * bar_length + \
169                     ' ' * (length - bar_length) + ']'
170                 print(f'\rLoading {mode} {bar} {progress}/{number}', end='')
171                 email = f.read()
172                 token_list = email2TokenList(email)
173                 correos.append((token_list, spam))
174         print()
175     print(len(correos))
176     return correos
177
178
179 def transform_mail(correos, vocab) -> tuple[np.ndarray, np.ndarray]:
180     """Transforms the emails into a matrix of length of the vocabulary with 1 if the word
181     is in the email
182
183     Args:
184         correos (_type_): mails
185         vocab (_type_): dictionary
186
187     Returns:
188         tuple[np.ndarray, np.ndarray]: transformed emails with label indicating if its spam
189         or not
190     """
191     X = []
192     y = []
193     for c, s in correos:
194         x = np.zeros(len(vocab) + 1)
195         for word in c:
196             if word in vocab:
197                 x[vocab[word]] = 1
198         X.append(x)
199         y.append(s)
200     return np.array(X), np.array(y)
201
202
203 def plot_results(train_scores: list[float], cv_scores: list[float], test_scores: list[float]
, times: list[float]) -> None:
204     """Plots the results
205
206     Args:
207         train_scores (list[float]): train scores
208         cv_scores (list[float]): cv scores
209         test_scores (list[float]): test scores
210         times (list[float]): times
211     """
212     plt.clf()
213     X: np.ndaarray = np.array(
        ['Logistic Regression', 'SVM', 'NN', 'Pytorch', 'Poly'])

```

```

214     x = np.arange(len(X))
215     plt.bar(x-0.2,
216             train_scores, 0.2, label=f'Train')
217     plt.bar(x,
218             cv_scores, 0.2, label=f'CV')
219     plt.bar(x+0.2,
220             test_scores, 0.2, label=f'Test')
221     plt.legend()
222     plt.xticks(x, X)
223     plt.savefig(f'{plot_folder}/results.png', dpi=300)
224     plt.clf()
225     plt.plot(X, times)
226     plt.savefig(f'{plot_folder}/times.png', dpi=300)
227
228
229 def compare_results() -> None:
230     """Compares the results of the different models
231     """
232     lr_data = sio.loadmat('res/logistic_regression.mat')
233     svm_data = sio.loadmat('res/svm.mat')
234     nn_data = sio.loadmat('res/nn.mat')
235     pytorch_data = sio.loadmat('res/pytorch.mat')
236     poly_data = sio.loadmat('res/poly.mat')
237     print('Logistic Regression')
238     print(f"Score: {lr_data['train_score']}")
239     print(f"CV Score: {lr_data['cv_score']}")
240     print(f"Test Score: {lr_data['test_score']}")
241     print(f"Time: {lr_data['time']}")
242     print(f'Best params: {lr_data["best_params"]}')
243     print('SVM')
244     print(f"Score: {svm_data['train_score']}")
245     print(f"CV Score: {svm_data['cv_score']}")
246     print(f"Test Score: {svm_data['test_score']}")
247     print(f"Time: {svm_data['time']}")
248     print(f'Best params: {svm_data["best_params"]}')
249     print('NN')
250     print(f"Score: {nn_data['train_score']}")
251     print(f"CV Score: {nn_data['cv_score']}")
252     print(f"Test Score: {nn_data['test_score']}")
253     print(f"Time: {nn_data['time']}")
254     print(f'Best params: {nn_data["best_params"]}')
255     print('Pytorch')
256     print(f"Score: {pytorch_data['train_score']}")
257     print(f"CV Score: {pytorch_data['cv_score']}")
258     print(f"Test Score: {pytorch_data['test_score']}")
259     print(f"Time: {pytorch_data['time']}")
260     print(f'Best params: {pytorch_data["best_params"]}')
261     print('Poly')
262     print(f"Score: {poly_data['train_score']}")
263     print(f"CV Score: {poly_data['cv_score']}")
264     print(f"Test Score: {poly_data['test_score']}")
265     print(f"Time: {poly_data['time']}")
266     print(f'Best params: {poly_data["best_params"]}')
267
268     train_scores = [lr_data['train_score'][0][0], svm_data['train_score']
269                     [0][0], nn_data['train_score'][0][0], pytorch_data['train_score']
270                     [0][0], poly_data['train_score'][0][0]]
271     cv_scores = [lr_data['cv_score'][0][0], svm_data['cv_score']
272                 [0][0], nn_data['cv_score'][0][0], pytorch_data['cv_score'][0][0],
273                 poly_data['cv_score'][0][0]]
274     test_scores = [lr_data['test_score'][0][0], svm_data['test_score']
275                   [0][0], nn_data['test_score'][0][0], pytorch_data['test_score'][0][0],
276                   poly_data['test_score'][0][0]]
277     times = [lr_data['time'][0][0], svm_data['time'][0][0],
278             nn_data['time'][0][0], pytorch_data['time'][0][0], poly_data['time'][0][0]]
279     print(train_scores)
280     plot_results(train_scores, cv_scores, test_scores, times)
281
282
283 def apartado_B():
284     """Apartado B
285     """
286     correos = load_data_spam()
287     vocab = getVocabDict()
288     X, y = transform_mail(correos, vocab)
289     if not os.path.exists(f'res/svm.mat'):

```

```

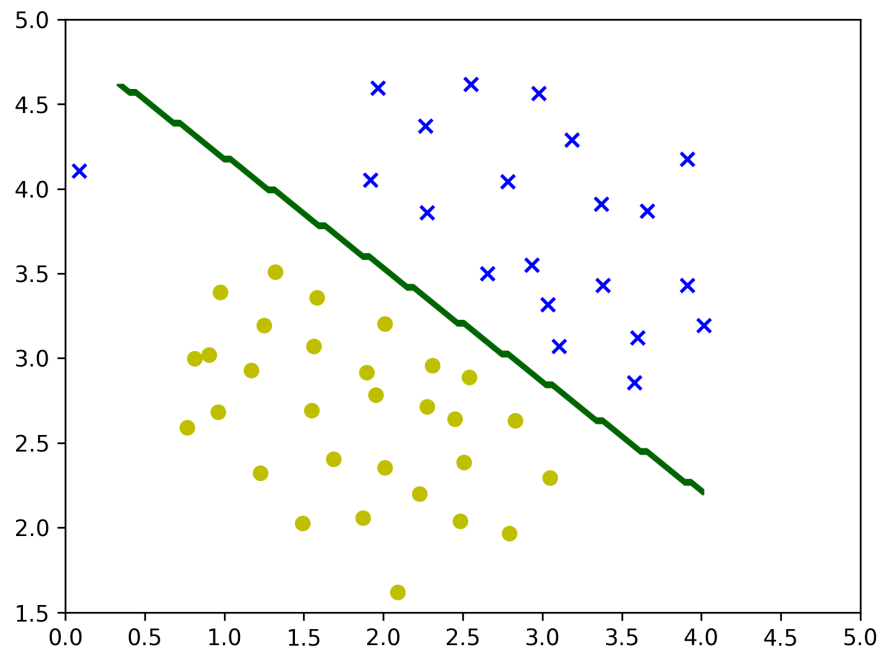
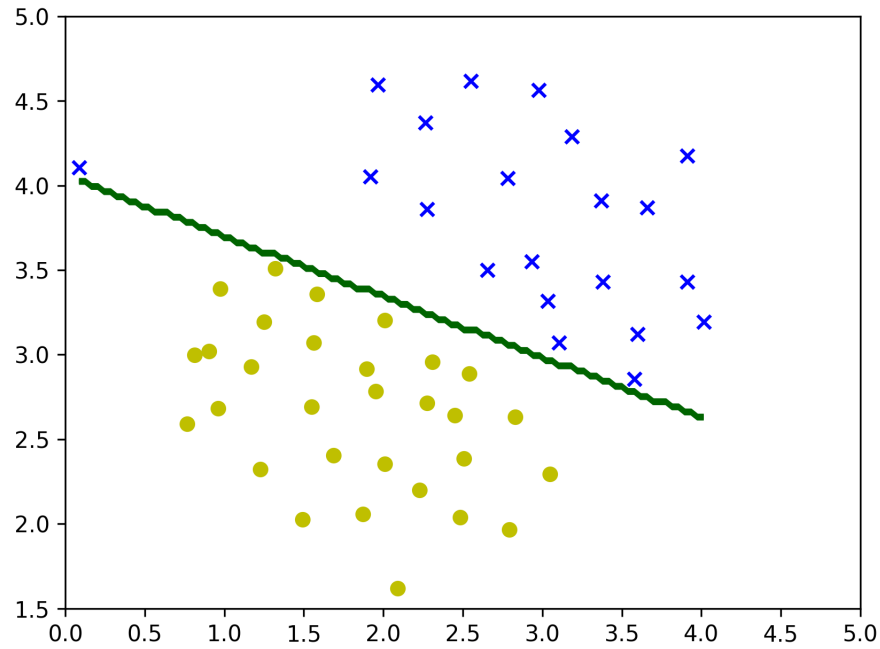
287     print('Training SVM')
288     SVM_Trainer.trainer(X, y)
289 if not os.path.exists(f'res/logistic_regression.mat'):
290     print('Training Logistic Regression')
291     Logic_Regression_Trainer.LR_trainer(X, y)
292 if not os.path.exists(f'res/pytorch.mat'):
293     print('Training Pytorch')
294     pytorch_trainer.trainer(X, y)
295 if not os.path.exists(f'res/nn.mat'):
296     print('Training NN')
297     nn_trainer.trainer(X, y)
298 if not os.path.exists(f'res/poly.mat'):
299     print('Training Poly')
300     Poly_trainer.trainer(X, y)
301
302 compare_results()
303
304
305 def Pruebas():
306     correos = load_data_spam()
307     vocab = getVocabDict()
308     X, y = transform_mail(correos, vocab)
309     lr_data = sio.loadmat('res/logistic_regression.mat')
310     svm_data = sio.loadmat('res/svm.mat')
311     nn_data = sio.loadmat('res/nn.mat')
312     pytorch_data = sio.loadmat('res/pytorch.mat')
313
314     X_train, X_test, y_train, y_test = train_test_split(
315         X, y, test_size=0.3, random_state=22)
316     X_cv, X_test, y_cv, y_test = train_test_split(
317         X_test, y_test, test_size=0.5, random_state=22)
318
319     print('Logistic Regression')
320     w = np.zeros(X.shape[1] + 1)
321     b = 1
322     X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
323     best_params = lr_data['best_params'][0]
324     w, b, _, _ = Logic_Regression_Trainer.gradient_descent(
325         X_train, y_train, w, b, Logic_Regression_Trainer.compute_cost_reg,
326         Logic_Regression_Trainer.compute_gradient_reg, best_params[0], 1000, best_params[1])
327
328     train_score = Logic_Regression_Trainer.predict_check(
329         X_train, y_train, w, b)
330     print(f'Train score: {train_score}')
331     X_cv = np.hstack((np.ones((X_cv.shape[0], 1)), X_cv))
332     cv_score = Logic_Regression_Trainer.predict_check(X_cv, y_cv, w, b)
333     print(f'CV score: {cv_score}')
334     X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
335     test_score = Logic_Regression_Trainer.predict_check(X_test, y_test, w, b)
336     print(f'Test score: {test_score}')
337
338     print('SVM')
339     best_params = svm_data['best_params'][0]
340     svm_gauss = svm.SVC(
341         kernel='rbf', C=best_params[0], gamma=1/(2*best_params[1]**2))
342     svm_gauss.fit(X_train, y_train.ravel())
343
344     test_score = svm_gauss.score(X_test, y_test)
345     cv_score = svm_gauss.score(X_cv, y_cv)
346     train_score = svm_gauss.score(X_train, y_train)
347     print(f'Test score: {test_score}')
348     print(f'CV score: {cv_score}')
349     print(f'Train score: {train_score}')
350
351     print('Pytorch')
352     best_params = pytorch_data['best_params'][0]
353     criterion = nn.CrossEntropyLoss().to(device)
354     model = ComplexModel(X_train.shape[1])
355     optimizer = optim.Adam(model.parameters(), lr=best_params[1],
356                             weight_decay=best_params[0])
357     train_dl = train_data(X_train, y_train)
358     model = train_model(model, train_dl, criterion, optimizer, 20)
359
360     test_score = pred_check(
361         model(torch.tensor(X_test, dtype=torch.float).to(device)), y_test)
362     cv_score = pred_check(

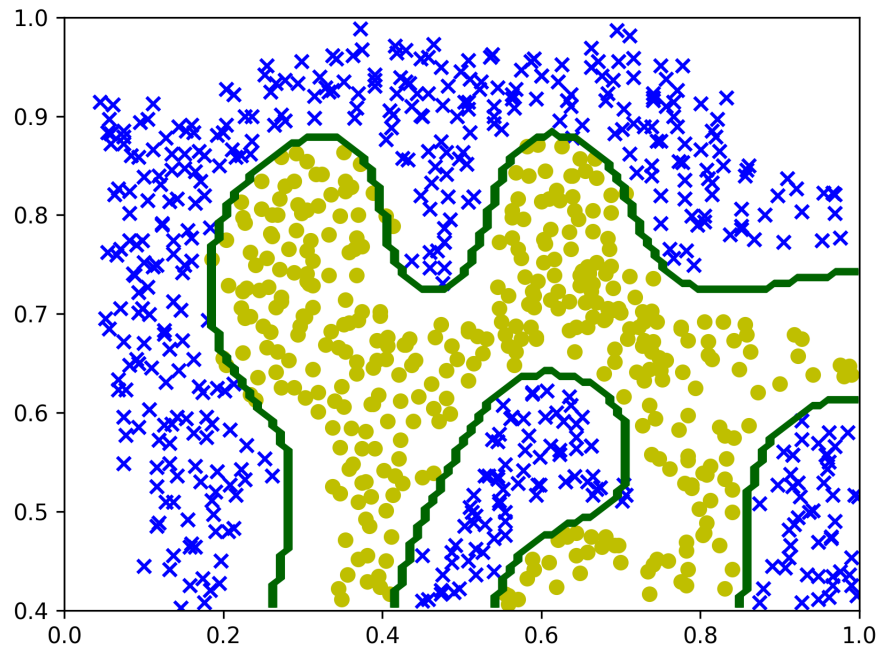
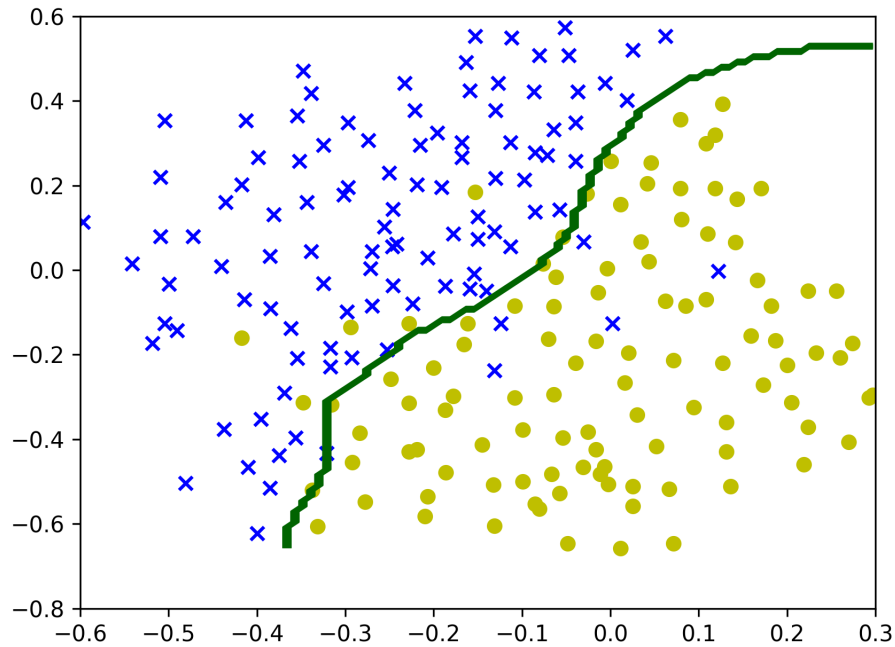
```

```

362     model(torch.tensor(X_cv, dtype=torch.float).to(device)), y_cv)
363     train_score = pred_check(
364         model(torch.tensor(X_train, dtype=torch.float).to(device)), y_train)
365
366     print(f'Test score: {test_score}')
367     print(f'CV score: {cv_score}')
368     print(f'Train score: {train_score}')
369
370     print('NN')
371     best_params = nn_data['best_params'][0]
372     input_layer_size = X.shape[1]
373     hidden_layer_size = 125
374     num_labels = 2
375     yA = [0 if i == 1 else 1 for i in y_train]
376     yB = [1 if i == 1 else 0 for i in y_train]
377     y_encoded = np.array([yA, yB]).T
378     theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
379     theta2 = np.random.rand(num_labels, hidden_layer_size + 1)
380     _, _, _, theta1, theta2, nn_trainer.train_model(X_train, y_encoded, theta1,
381                                                     theta2, best_params[0], best_params[1],
382                                                     1000)
383     yA = [0 if i == 1 else 1 for i in y_test]
384     yB = [1 if i == 1 else 0 for i in y_test]
385     y_encoded = np.array([yA, yB]).T
386     nn_score = nn_trainer.predict_check(X_test, y_encoded, theta1, theta2)
387     print(f'Test score: {nn_score}')
388     yA = [0 if i == 1 else 1 for i in y_cv]
389     yB = [1 if i == 1 else 0 for i in y_cv]
390     y_encoded = np.array([yA, yB]).T
391     nn_score = nn_trainer.predict_check(X_cv, y_encoded, theta1, theta2)
392     print(f'CV score: {nn_score}')
393     yA = [0 if i == 1 else 1 for i in y_train]
394     yB = [1 if i == 1 else 0 for i in y_train]
395     y_encoded = np.array([yA, yB]).T
396     nn_score = nn_trainer.predict_check(X_train, y_encoded, theta1, theta2)
397     print(f'Train score: {nn_score}')
398
399 def main() -> None:
400     apartado_A()
401     # apartado_B()
402     Pruebas()
403
404
405 if __name__ == '__main__':
406     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:** invariable a partir de grado 1, crashea el ordenador porque la matriz de datos es demasiado grande.

Código del entrenador de regresión lógica:

```

1 import numpy as np
2 import copy
3 import time
4 import scipy.io as sio
5 import concurrent.futures
6 from sklearn.model_selection import train_test_split
7
8
9 def compute_cost_reg(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_: float
10 = 1) -> float:
11     """
12     Computes the cost over all examples
13     Args:
14         X : (array_like Shape (m,n)) data, m examples by n features
15         y : (array_like Shape (m,)) target value
16         w : (array_like Shape (n,)) Values of parameters of the model
17         b : (array_like Shape (n,)) Values of bias parameter of the model
18         lambda_ : (scalar, float) Controls amount of regularization
19     Returns:
20         total_cost: (scalar) cost
21     """
22     total_cost = compute_cost(X, y, w, b)
23     total_cost += (lambda_ / (2 * X.shape[0])) * np.sum(w**2)
24
25     return total_cost
26
27
28 def loss(X: np.ndarray, Y: np.ndarray, fun: np.ndarray, w: np.ndarray, b: float) -> float:
29     """loss function for the logistic regression
30
31     Args:
32         X (np.ndarray): X values
33         Y (np.ndarray): Expected y results
34         fun (np.ndarray): logistic regression function
35         w (np.ndarray): weights
36         b (float): bias
37
38     Returns:
39         float: total loss of the regression
40     """
41
42     return (-Y * np.log(fun(X, w, b) + 1e-6)) - ((1 - Y) * np.log(1 - fun(X, w, b) + 1e-6))
43
44
45 def compute_cost(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_=None) ->
46 float:
47     """
48     Computes the cost over all examples
49     Args:
50         X : (ndarray Shape (m,n)) data, m examples by n features
51         y : (array_like Shape (m,)) target value
52         w : (array_like Shape (n,)) Values of parameters of the model
53         b : scalar Values of bias parameter of the model
54         lambda_: unused placeholder
55     Returns:
56         total_cost: (scalar) cost
57     """

```

```

57     # apply the loss function for each element of the x and y arrays
58     loss_v = loss(X, y, function, w, b)
59     total_cost = np.sum(loss_v)
60     total_cost /= X.shape[0]
61
62     return total_cost
63
64
65 def compute_gradient(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_=None)
66     -> tuple[float, np.ndarray]:
67     """
68     Computes the gradient for logistic regression
69
70     Args:
71         X : (ndarray Shape (m,n)) variable such as house size
72         y : (array_like Shape (m,1)) actual value
73         w : (array_like Shape (n,1)) values of parameters of the model
74         b : (scalar) value of parameter of the model
75         lambda_: unused placeholder
76
77     Returns
78         dj_db: (scalar) The gradient of the cost w.r.t. the parameter b.
79         dj_dw: (array_like Shape (n,1)) The gradient of the cost w.r.t. the parameters w.
80     """
81
82     func = function(X, w, b)
83
84     dj_dw = np.dot(func - y, X)
85     dj_dw /= X.shape[0]
86
87     dj_db = np.sum(func - y)
88     dj_db /= X.shape[0]
89
90     return dj_db, dj_dw
91
92 def compute_gradient_reg(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_:
93     float = 1) -> tuple[float, np.ndarray]:
94     """
95     Computes the gradient for linear regression
96
97     Args:
98         X : (ndarray Shape (m,n)) variable such as house size
99         y : (ndarray Shape (m,)) actual value
100        w : (ndarray Shape (n,)) values of parameters of the model
101        b : (scalar) value of parameter of the model
102        lambda_ : (scalar,float) regularization constant
103
104     Returns
105         dj_db: (scalar) The gradient of the cost w.r.t. the parameter b.
106         dj_dw: (ndarray Shape (n,)) The gradient of the cost w.r.t. the parameters w.
107     """
108
109     dj_db, dj_dw = compute_gradient(X, y, w, b)
110     dj_dw += (lambda_ / X.shape[0]) * w
111
112     return dj_db, dj_dw
113
114 def gradient_descent(X: np.ndarray, y: np.ndarray, w_in: np.ndarray, b_in: float,
115     cost_function: float, gradient_function: float, alpha: float, num_iters: int, lambda_:
116     float = None) -> tuple[np.ndarray, float, np.ndarray, np.ndarray]:
117     """
118     Performs batch gradient descent to learn theta. Updates theta by taking
119     num_iters gradient steps with learning rate alpha
120
121     Args:
122         X : (array_like Shape (m, n)) Initial values of parameters of the model
123         y : (array_like Shape (m,)) Initial value of parameter of the model
124         w_in : (array_like Shape (n,)) function to compute cost
125         b_in : (scalar) Learning rate
126         cost_function: (float) number of iterations to run gradient descent
127         alpha : (float) number of iterations to run gradient descent
128         num_iters : (int) regularization constant
129         lambda_ : (scalar, float) regularization constant
130
131     Returns:
132         w : (array_like Shape (n,)) Updated values of parameters of the model after

```

```

129         running gradient descent
130         b : (scalar)             Updated value of parameter of the model after
131         running gradient descent
132         J_history : (ndarray): Shape (num_iters,) J at each iteration,
133         primarily for graphing later
134     """
135
136     w = copy.deepcopy(w_in)
137     b = b_in
138     predict_history = [predict_check(X, y, w, b)]
139     J_history = [cost_function(X, y, w, b, lambda_)]
140
141     for i in range(num_iters):
142         dj_db, dj_dw = gradient_function(X, y, w, b, lambda_)
143         w = w - (alpha * dj_dw)
144         b -= alpha * dj_db
145         J_history.append(cost_function(X, y, w, b, lambda_))
146         predict_history.append(predict_check(X, y, w, b))
147
148     return w, b, np.array(J_history), predict_history
149
150
151 def predict(X, w, b) -> np.ndarray:
152     """
153     Predict whether the label is 0 or 1 using learned logistic
154     regression parameters w and b
155
156     Args:
157     X : (ndarray Shape (m, n))
158     w : (array_like Shape (n,))      Parameters of the model
159     b : (scalar, float)              Parameter of the model
160
161     Returns:
162     p: (ndarray (m,1))
163         The predictions for X using a threshold at 0.5
164     """
165
166     p = np.vectorize(lambda x: 1 if x > 0.5 else 0)(
167         function(X, w, b))
168     return p
169
170
171 def predict_check(X, Z, w, b) -> float:
172     """Gives a percentage of the accuracy of the prediction
173
174     Args:
175     X (_type_): X train data
176     Z (_type_): expected values
177     w (_type_): weights
178     b (_type_): bias
179
180     Returns:
181     float: percentage of accuracy
182     """
183     p = predict(X, w, b)
184     return np.sum(p == Z) / Z.shape[0]
185
186
187 def sigmoid(z: np.ndarray) -> np.ndarray:
188     """
189     Compute the sigmoid of z
190
191     Args:
192     z (ndarray): A scalar, numpy array of any size.
193
194     Returns:
195     g (ndarray): sigmoid(z), with the same shape as z
196
197     """
198
199     g = 1/(1+np.exp(-z))
200
201     return g
202
203
204 def function(x: np.ndarray, w: np.ndarray, b: float) -> np.ndarray:

```

```

205     """Function using 'sigmoid' to calculate the value of y to the given x, w and b
206
207     Args:
208         x (np.ndarray): X data
209         w (np.ndarray): w data
210         b (float): b data
211
212     Returns:
213         np.ndarray: final value after the sigmoid
214     """
215     return sigmoid(np.dot(x, w) + b)
216
217
218 def train_model(X: np.ndarray, y: np.ndarray, x_cv: np.ndarray, y_cv: np.ndarray, alpha:
219 float, lambda_: float, num_iters: int) -> tuple[float, float, float]:
220     """Train the model with the given parameters
221     Args:
222         X (np.ndarray): Training data
223         y (np.ndarray): Training target
224         x_cv (np.ndarray): Cross validation data
225         y_cv (np.ndarray): Cross validation target
226         alpha (float): Learning rate
227         lambda_ (float): Regularization parameter
228         num_iters (int): Number of iterations
229     Returns:
230         tuple[float, float, float]: Learning rate, Regularization parameter, Score
231     """
232     print(f'Alpha: {alpha} Lambda: {lambda_}')
233     m, n = X.shape
234     X = np.hstack((np.ones((m, 1)), X))
235     x_cv = np.hstack((np.ones((x_cv.shape[0], 1)), x_cv))
236     w = np.zeros(X.shape[1])
237     b = 1
238     w, b, _, _ = gradient_descent(
239         X, y, w, b, compute_cost_reg, compute_gradient_reg, alpha, num_iters, lambda_)
240     score = predict_check(x_cv, y_cv, w, b)
241     return (alpha, lambda_, score)
242
243 def LR_trainer(X: np.ndarray, y: np.ndarray) -> None:
244     """Trains the model with the given data
245     Args:
246         X (np.ndarray): Input data
247         y (np.ndarray): Target data
248     """
249     alphas = [0.1, 0.3, 1, 3, 10, 30]
250     lambdas = [0.1, 0.3, 1, 3, 10, 30]
251     num_iters = 1000
252     best_score = 0
253     best_params = (0, 0)
254
255     X_train, X_test, y_train, y_test = train_test_split(
256         X, y, test_size=0.3, shuffle=True, random_state=22)
257     X_cv, X_test, y_cv, y_test = train_test_split(
258         X_test, y_test, test_size=0.5, shuffle=True, random_state=22)
259
260     with concurrent.futures.ProcessPoolExecutor() as executor:
261         futures = []
262         for lambda_ in lambdas:
263             for alpha in alphas:
264                 futures.append(executor.submit(
265                     train_model, X_train, y_train, X_cv, y_cv, alpha, lambda_, num_iters))
266
267         for future in concurrent.futures.as_completed(futures):
268             alpha, lambda_, score = future.result()
269             print(f'Alpha: {alpha} Lambda: {lambda_} Score: {score}')
270             if score > best_score:
271                 best_score = score
272                 best_params = (alpha, lambda_)
273
274     print(f'Best score: {best_score}')
275     print(f'Best params: {best_params}')
276
277     start = time.time()
278     w = np.zeros(X.shape[1] + 1)
279     b = 1

```

```

280 X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
281 w, b, _, _ = gradient_descent(
282     X_train, y_train, w, b, compute_cost_reg, compute_gradient_reg, best_params[0],
    num_iters, best_params[1])
283 end = time.time()
284 print(f'Training time: {end-start}')
285 train_score = predict_check(X_train, y_train, w, b)
286 print(f'Train score: {train_score}')
287 X_cv = np.hstack((np.ones((X_cv.shape[0], 1)), X_cv))
288 cv_score = predict_check(X_cv, y_cv, w, b)
289 print(f'CV score: {cv_score}')
290 X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
291 test_score = predict_check(X_test, y_test, w, b)
292 print(f'Test score: {test_score}')
293 sio.savemat('res/logistic_regression.mat', {'w': w, 'b': b, 'train_score': train_score,
294     'cv_score': cv_score, 'test_score': test_score, 'best_params': best_params,
    'time': end-start})

```

Código del entrenador de SVM gaussiano:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.model_selection import train_test_split
4 import time
5 from sklearn import svm
6 import scipy.io as sio
7 import concurrent.futures
8
9
10 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]:
11     """Train the model with the given parameters
12     Args:
13         C (float): Regularization parameter
14         sigma (float): Gaussian kernel parameter
15         x_train (np.ndarray): Training data
16         y_train (np.ndarray): Training target
17         x_cv (np.ndarray): Cross validation data
18         y_cv (np.ndarray): Cross validation target
19     Returns:
20         tuple[float, float, float]: Regularization parameter, Gaussian kernel parameter,
    Score
21     """
22     print(f'C: {C} sigma: {sigma}')
23     svm_gauss = svm.SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
24     svm_gauss.fit(x_train, y_train.ravel())
25     score = svm_gauss.score(x_cv, y_cv.ravel())
26     return (C, sigma, score)
27
28
29 def trainer(X: np.ndarray, y: np.ndarray) -> None:
30     """Trains the model with the given data
31     Args:
32         X (np.ndarray): Input data
33         y (np.ndarray): Target data
34     """
35     C_values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
36     sigma_values = C_values
37     x_train, x_test, y_train, y_test = train_test_split(
38         X, y, test_size=0.3, random_state=22)
39     x_cv, x_test, y_cv, y_test = train_test_split(
40         x_test, y_test, test_size=0.5, random_state=22)
41     best_score = 0
42     best_params = (0, 0)
43
44     with concurrent.futures.ProcessPoolExecutor() as executor:
45         futures = []
46         for C in C_values:
47             for sigma in sigma_values:
48                 futures.append(executor.submit(
49                     train_model, C, sigma, x_train, y_train, x_cv, y_cv))
50
51         for future in concurrent.futures.as_completed(futures):
52             C, sigma, score = future.result()
53             print(f'C: {C} sigma: {sigma} score: {score}')
54             if score > best_score:

```

```

55         best_score = score
56         best_params = (C, sigma)
57
58     print(f'Best score: {best_score}')
59
60     start = time.time()
61     svm_gauss = svm.SVC(
62         kernel='rbf', C=best_params[0], gamma=1/(2*best_params[1]**2))
63     svm_gauss.fit(x_train, y_train.ravel())
64     end = time.time()
65     print(f'Training time: {end-start}')
66
67     test_score = svm_gauss.score(x_test, y_test)
68     cv_score = svm_gauss.score(x_cv, y_cv)
69     train_score = svm_gauss.score(x_train, y_train)
70     sio.savemat('res/svm.mat', {'train_score': train_score,
71                                'cv_score': cv_score, 'test_score': test_score,
                                'best_params': best_params, 'time': end-start})

```

Código del entrenador de NN:

```

1  import numpy as np
2  import scipy.io as sio
3  import time
4  from sklearn.model_selection import train_test_split
5
6
7  def sigmoid(z: np.ndarray) -> np.ndarray:
8      """
9      Compute the sigmoid of z
10
11      Args:
12          z (ndarray): A scalar, numpy array of any size.
13
14      Returns:
15          g (ndarray): sigmoid(z), with the same shape as z
16
17      """
18
19      g = 1/(1+np.exp(-z))
20
21      return g
22
23
24  def fix_data(X: np.ndarray) -> np.ndarray:
25      """Fixes the data to avoid log(0) errors
26
27      Args:
28          X (np.ndarray): train data
29
30      Returns:
31          np.ndarray: matrix with no 0 or 1 values
32      """
33      return X + 1e-7
34
35
36  def cost(theta1: np.ndarray, theta2: np.ndarray, X: np.ndarray, y: np.ndarray, lambda_:
37           float = 0.0) -> float:
38      """
39      Compute cost for 2-layer neural network.
40
41      Parameters
42      -----
43      theta1 : array_like
44          Weights for the first layer in the neural network.
45          It has shape (2nd hidden layer size x input size + 1)
46
47      theta2: array_like
48          Weights for the second layer in the neural network.
49          It has shape (output layer size x 2nd hidden layer size + 1)
50
51      X : array_like
52          The inputs having shape (number of examples x number of dimensions).
53
54      y : array_like
55          1-hot encoding of labels for the input, having shape

```

```

55         (number of examples x number of labels).
56
57     lambda_ : float
58         The regularization parameter.
59
60     Returns
61     -----
62     J : float
63         The computed value for the cost function.
64
65     """
66     L = 2
67     layers = [theta1, theta2]
68     k: int = y.shape[1]
69     h, z = neural_network(X, [theta1, theta2])
70
71     h = h[-1]
72
73     h = fix_data(h)
74
75     J = y * np.log(h + 1e-7)
76     J += (1 - y) * np.log(1 - h + 1e-7)
77
78     J = -1 / X.shape[0] * np.sum(J)
79
80     if lambda_ != 0:
81         reg = 0
82         for layer in layers:
83             reg += np.sum(layer[:, 1:] ** 2)
84         J += lambda_ / (2 * X.shape[0]) * reg
85     return J
86
87
88 def neural_network(X: np.ndarray, thetas: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
89     """Generate the neural network with a given set of weights
90
91     Args:
92         X (np.ndarray): data
93         thetas (np.ndarray): array containing the weights for each layer
94
95     Returns:
96         tuple[np.ndarray, np.ndarray]: tuple containing the activations and the z values
97         for each layer
98     """
99     a = []
100     z = []
101     a.append(X.copy())
102     for theta in thetas:
103         a[-1] = np.hstack((np.ones((a[-1].shape[0], 1)), a[-1]))
104         z.append(np.dot(a[-1], theta.T))
105         a.append(sigmoid(z[-1]))
106     return a, z
107
108 def backprop(theta1: np.ndarray, theta2: np.ndarray, X: np.ndarray, y: np.ndarray, lambda_:
109             float) -> tuple[float, np.ndarray, np.ndarray]:
110     """
111     Compute cost and gradient for 2-layer neural network.
112
113     Parameters
114     -----
115     theta1 : array_like
116         Weights for the first layer in the neural network.
117         It has shape (2nd hidden layer size x input size + 1)
118
119     theta2: array_like
120         Weights for the second layer in the neural network.
121         It has shape (output layer size x 2nd hidden layer size + 1)
122
123     X : array_like
124         The inputs having shape (number of examples x number of dimensions).
125
126     y : array_like
127         1-hot encoding of labels for the input, having shape
128         (number of examples x number of labels).

```

```

129     lambda_ : float
130         The regularization parameter.
131
132     Returns
133     -----
134     J : float
135         The computed value for the cost function.
136
137     grad1 : array_like
138         Gradient of the cost function with respect to weights
139         for the first layer in the neural network, theta1.
140         It has shape (2nd hidden layer size x input size + 1)
141
142     grad2 : array_like
143         Gradient of the cost function with respect to weights
144         for the second layer in the neural network, theta2.
145         It has shape (output layer size x 2nd hidden layer size + 1)
146
147     """
148     m = X.shape[0]
149     L = 2
150
151     delta = np.empty(2, dtype=object)
152     delta[0] = np.zeros(theta1.shape)
153     delta[1] = np.zeros(theta2.shape)
154
155     a, z = neural_network(X, [theta1, theta2])
156
157     for k in range(m):
158         a1k = a[0][k, :]
159         a2k = a[1][k, :]
160         hk = a[2][k, :]
161         yk = y[k, :]
162
163         d3k = hk - yk
164         d2k = np.dot(theta2.T, d3k) * a2k * (1 - a2k)
165
166         delta[0] = delta[0] + \
167             np.matmul(d2k[1:, np.newaxis], a1k[np.newaxis, :])
168         delta[1] = delta[1] + np.matmul(d3k[:, np.newaxis], a2k[np.newaxis, :])
169
170     grad1 = delta[0] / m
171     grad2 = delta[1] / m
172
173     if lambda_ != 0:
174         grad1[:, 1:] += lambda_ / m * theta1[:, 1:]
175         grad2[:, 1:] += lambda_ / m * theta2[:, 1:]
176
177     J = cost(theta1, theta2, X, y, lambda_)
178
179     return (J, grad1, grad2)
180
181
182 def gradient_descent(X: np.ndarray, y: np.ndarray, theta1: np.ndarray, theta2: np.ndarray,
183                     alpha: float, lambda_: float, num_iters: int) -> tuple[np.ndarray, np.ndarray, np.
184                     ndarray]:
185     """Generates the gradient descent for the neural network
186
187     Args:
188         X (np.ndarray): Train data
189         y (np.ndarray): Expected output in one hot encoding
190         theta1 (np.ndarray): initial weights for the first layer
191         theta2 (np.ndarray): initial weights for the second layer
192         alpha (float): learning rate
193         lambda_ (float): regularization parameter
194         num_iters (int): number of iterations to run
195
196     Returns:
197         tuple[np.ndarray, np.ndarray, np.ndarray]: tuple with the final weights for the
198         first and second layer and the cost history
199     """
200     m = X.shape[0]
201     J_history = np.zeros(num_iters)
202     for i in range(num_iters):
203         print('Iteration: ', i + 1, '/', num_iters, end='\r')
204         J, grad1, grad2 = backprop(theta1, theta2, X, y, lambda_)

```



```

202     theta1 = theta1 - alpha * grad1
203     theta2 = theta2 - alpha * grad2
204     J_history[i] = J
205     print('Gradient descent finished.')
206     return theta1, theta2, J_history
207
208
209 def train_model(X, y, x_cv, y_cv, alpha, lambda_, num_iters):
210     # start = time.time()
211     print(f'Alpha: {alpha} Lambda: {lambda_}')
212     input_layer_size = X.shape[1]
213     hidden_layer_size = 125
214     num_labels = 2
215     yA = [0 if i == 1 else 1 for i in y]
216     yB = [1 if i == 1 else 0 for i in y]
217     y_encoded = np.array([yA, yB]).T
218
219     theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
220     theta2 = np.random.rand(num_labels, hidden_layer_size + 1)
221
222     theta1, theta2, J_history = gradient_descent(
223         X, y_encoded, theta1, theta2, alpha, lambda_, num_iters)
224
225     score = predict_percentage(x_cv, y_cv, theta1, theta2)
226     # time = time.time() - start
227     return (alpha, lambda_, score, theta1, theta2)
228
229
230 def prediction(X: np.ndarray, theta1: np.ndarray, theta2: np.ndarray) -> np.ndarray:
231     """Generates the neural network prediction
232
233     Args:
234         X (np.ndarray): data
235         theta1 (np.ndarray): first layer weight
236         theta2 (np.ndarray): second layer weight
237
238     Returns:
239         np.ndarray: best prediction for each row in 'X'
240     """
241     m = X.shape[0]
242     p = np.zeros(m)
243     a, z = neural_network(X, [theta1, theta2])
244     h = a[-1]
245
246     return np.argmax(h, axis=1)
247
248
249 def predict_percentage(X: np.ndarray, y: np.ndarray, theta1: np.ndarray, theta2: np.ndarray) -> float:
250     """Gives the accuracy of the neural network
251
252     Args:
253         X (ndarray): Train data
254         y (ndarray): Expected output
255         theta1 (ndarray): First layer weights
256         theta2 (ndarray): Second layer weights
257
258     Returns:
259         float: Accuracy of the neural network
260     """
261     m = X.shape[0]
262     p = prediction(X, theta1, theta2)
263
264     return p[p == y].size / m
265
266
267 def trainer(X: np.ndarray, y: np.ndarray) -> None:
268     lambdas = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
269     alphas = lambdas
270     num_iters = 100
271     best_score = 0
272     best_params = (0, 0)
273     input_layer_size = X.shape[1]
274     hidden_layer_size = 125
275     num_labels = 2
276     best_time = 0

```

```

277 X_train, X_test, y_train, y_test = train_test_split(
278     X, y, test_size=0.3, random_state=22)
279 X_cv, X_test, y_cv, y_test = train_test_split(
280     X_test, y_test, test_size=0.5, random_state=22)
281 model = (np.array([]), np.array([]))
282
283 for alpha in alphas:
284     for lambda_ in lambdas:
285
286         start = time.time()
287         print(f'Alpha: {alpha} Lambda: {lambda_}')
288         input_layer_size = X.shape[1]
289         hidden_layer_size = 125
290         num_labels = 2
291         yA = [0 if i == 1 else 1 for i in y]
292         yB = [1 if i == 1 else 0 for i in y]
293         y_encoded = np.array([yA, yB]).T
294         theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
295         theta2 = np.random.rand(num_labels, hidden_layer_size + 1)
296
297         theta1, theta2, J_history = gradient_descent(
298             X, y_encoded, theta1, theta2, alpha, lambda_, num_iters)
299
300         score = predict_percentage(X_cv, y_cv, theta1, theta2)
301         print(f'Score: {score}')
302         aux_time = time.time() - start
303         if score > best_score:
304             best_score = score
305             best_params = (alpha, lambda_)
306             model = (theta1, theta2)
307             best_time = aux_time
308         print(f'Best score: {best_score}')
309         print(f'Best params: {best_params}')
310
311         theta1 = np.random.rand(hidden_layer_size, input_layer_size + 1)
312         theta2 = np.random.rand(num_labels, hidden_layer_size + 1)
313         yA = [0 if i == 1 else 1 for i in y_train]
314         yB = [1 if i == 1 else 0 for i in y_train]
315         y_encoded = np.array([yA, yB]).T
316
317         theta1, theta2, = model
318         print(f'Training time: {best_time}')
319
320         train_score = predict_percentage(X_train, y_train, theta1, theta2)
321         print(f'Train score: {train_score}')
322         cv_score = predict_percentage(X_cv, y_cv, theta1, theta2)
323         print(f'CV score: {cv_score}')
324         test_score = predict_percentage(X_test, y_test, theta1, theta2)
325         print(f'Test score: {test_score}')
326         sio.savemat('res/nn.mat',
327             {'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:

```

1 import torch.nn as nn
2 import torch.optim as optim
3 import numpy as np
4 from sklearn.model_selection import train_test_split
5 import scipy.io as sio
6 import time
7 import torch
8
9 # Select cuda device if available to speed up training
10 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
11
12 if torch.cuda.is_available():
13     print(f'Using GPU {torch.cuda.get_device_name()}')
14 else:
15     print('Using CPU')
16
17
18 def train_data(x: np.ndarray, y: np.ndarray) -> torch.utils.data.DataLoader:
19     """ Create a DataLoader object from the input data
20     Args:
21         x (np.ndarray): Input data

```

```

22     y (np.ndarray): Target data
23     """
24     return torch.utils.data.DataLoader(torch.utils.data.TensorDataset(
25         torch.tensor(x, dtype=torch.float).to(device), torch.tensor(y).to(device)),
26         batch_size=2, shuffle=True)
27
28 def train_model(model: nn.Sequential, train_dl: torch.utils.data.DataLoader, criterion: nn.
29     CrossEntropyLoss, optimizer: optim.Adam, epochs: int) -> nn.Sequential:
30     """ Train the model with the given data
31     Args:
32         model (nn.Sequential): Model to train
33         train_dl (torch.utils.data.DataLoader): DataLoader object with the training data
34         criterion (nn.CrossEntropyLoss): Loss function
35         optimizer (optim.Adam): Optimizer
36         epochs (int): Number of epochs to train the model
37     Returns:
38         nn.Sequential: Trained model
39     """
40     for epoch in range(epochs):
41         model.train()
42         for x, y in train_dl:
43             optimizer.zero_grad()
44             y_pred = model(x)
45             loss = criterion(y_pred, y)
46             loss.backward()
47             optimizer.step()
48             print(f'Epoch: {epoch}, Loss: {loss.item()}')
49     return model
50
51 def ComplexModel(input_size: int) -> nn.Sequential:
52     """Creates a Sequential model with 3 layers
53
54     Args:
55         input_size (int): input size of the model
56
57     Returns:
58         nn.Sequential: base model
59     """
60     return nn.Sequential(
61         nn.Linear(input_size, 512),
62         nn.ReLU(),
63         nn.Linear(512, 10),
64         nn.ReLU(),
65         nn.Linear(10, 2),
66         nn.Sigmoid()
67     ).to(device)
68
69
70 def pred_check(pred: torch.Tensor, y: np.ndarray) -> float:
71     """Gives the accuracy of the model in percentage
72
73     Args:
74         pred (torch.Tensor): predictions made by the model
75         y (np.ndarray): target data
76
77     Returns:
78         float: predict percentage
79     """
80     return (pred.argmax(dim=1) == torch.tensor(y).to(device)).sum().item() / len(y)
81
82
83 def trainer(X: np.ndarray, y: np.ndarray) -> None:
84     """Trains the model with the given data
85     Args:
86         X (np.ndarray): Input data
87         y (np.ndarray): Target data
88     """
89
90     # device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
91
92     # y = np.array([[0, 1] if i == 1 else [1, 0] for i in y])
93
94     x_train, x_test, y_train, y_test = train_test_split(
95         X, y, test_size=0.3, random_state=22)

```

```

96 x_cv, x_test, y_cv, y_test = train_test_split(
97     x_test, y_test, test_size=0.5, random_state=22)
98 lambdas = np.array([0.001, 0.01, 0.05, 0.1, 0.2, 0.3])
99 learning_rates = np.array([0.01, 0.1, 0.5, 1])
100 best_score = 0
101 best_params = (0, 0)
102
103 for lambda_ in lambdas:
104     for learning_rate in learning_rates:
105         criterion = nn.CrossEntropyLoss().to(device)
106         print(f'Lambda: {lambda_}, Learning rate: {learning_rate}')
107         model = ComplexModel(x_train.shape[1])
108         optimizer = optim.Adam(
109             model.parameters(), lr=learning_rate, weight_decay=lambda_)
110         train_dl = train_data(x_train, y_train)
111         model = train_model(
112             model, train_dl, criterion, optimizer, 20)
113         pred = model(torch.tensor(x_cv, dtype=torch.float).to(device))
114
115         score = pred_check(pred, y_cv)
116         print(score)
117         if score > best_score:
118             best_score = score
119             best_params = (lambda_, learning_rate)
120
121 print(f'Best score: {best_score}')
122 print(f'Best params: {best_params}')
123
124 start = time.time()
125 criterion = nn.CrossEntropyLoss().to(device)
126 model = ComplexModel(X.shape[1])
127 optimizer = optim.Adam(model.parameters(), lr=best_params[1],
128                         weight_decay=best_params[0])
129 train_dl = train_data(x_train, y_train)
130 model = train_model(model, train_dl, criterion, optimizer, 20)
131 end = time.time()
132
133 test_score = pred_check(
134     model(torch.tensor(x_test, dtype=torch.float).to(device)), y_test)
135 cv_score = pred_check(
136     model(torch.tensor(x_cv, dtype=torch.float).to(device)), y_cv)
137 train_score = pred_check(
138     model(torch.tensor(x_train, dtype=torch.float).to(device)), y_train)
139
140 print(f'Test score: {test_score}')
141 print(f'CV score: {cv_score}')
142 print(f'Train score: {train_score}')
143 print(f'Time: {end-start}')
144
145 sio.savemat('res/pytorch.mat', {
146     'test_score': test_score,
147     'cv_score': cv_score,
148     'train_score': train_score,
149     'best_params': best_params,
150     'time': end-start
151 })

```

Código del entrenador de PolynomialTransformer:

```

1 import time
2 from typing import Union
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import sklearn.linear_model as lm
6 import sklearn.preprocessing as sp
7 import sklearn.model_selection as ms
8 import scipy.io as sio
9
10
11 def cost(y: np.ndarray, y_hat: np.ndarray) -> float:
12     """Calculates the cost of the model
13
14     Args:
15         y (np.ndarray): real values
16         y_hat (np.ndarray): predicted values
17

```

```

18     Returns:
19         float: cost of the model
20     """
21     return np.mean((y_hat - y)**2) / 2
22
23
24 def train_reg(x_train: np.ndarray, y_train: np.ndarray, grado: int, l: float) -> tuple[sp.
    PolynomialFeatures, sp.StandardScaler, lm.Ridge, np.ndarray]:
25     """ Trains a model given the training data with polynomial features and regularization
26
27     Args:
28         x_train (np.ndarray): x values of the training data
29         y_train (np.ndarray): y values of the training data
30         grado (int): degree of the polynomial
31         l (float): lambda value for the regularization
32
33     Returns:
34         tuple[sp.PolynomialFeatures, sp.StandardScaler, lm.Ridge, np.ndarray]:
35         _description_
36     """
37     poly: sp.PolynomialFeatures = sp.PolynomialFeatures(
38         degree=grado, include_bias=False)
39     x_train = poly.fit_transform(x_train)
40     scal: sp.StandardScaler = sp.StandardScaler()
41     # x_train = scal.fit_transform(x_train)
42     model: lm.Ridge = lm.Ridge(alpha=l)
43     model.fit(x_train, y_train)
44     return poly, scal, model, x_train
45
46 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]:
47     """Tests the model with the test data
48
49     Args:
50         x_test (np.ndarray): x values of the test data
51         y_test (np.ndarray): y values of the test data
52         x_train_aux (np.ndarray): x values of the training data
53         y_train (np.ndarray): y values of the training data
54         poly (sp.PolynomialFeatures): polynomial features
55         scal (sp.StandardScaler): standard scaler
56         model (Union[lm.LinearRegression, lm.Ridge]): model to test
57
58     Returns:
59         tuple[float, float]: test cost, train cost
60     """
61     x_test = poly.transform(x_test)
62     # x_test = scal.transform(x_test)
63
64     y_pred_test: np.ndarray = model.predict(x_test)
65     test_cost: float = cost(y_test, y_pred_test)
66
67     y_pred_train: np.ndarray = model.predict(x_train_aux)
68     train_cost: float = cost(y_train, y_pred_train)
69
70     return test_cost, train_cost
71
72
73 def trainer(X: np.ndarray, y: np.ndarray) -> None:
74     """Trains the model with the given data
75
76     Args:
77         X (np.ndarray): Input data
78         y (np.ndarray): Target data
79     """
80     x_train, x_test, y_train, y_test = ms.train_test_split(
81         X, y, test_size=0.3, random_state=22)
82     x_cv, x_test, y_cv, y_test = ms.train_test_split(
83         x_test, y_test, test_size=0.5, random_state=22)
84
85     lambdas: list[float] = [1e-5, 1e-4, 1e-3,
86                             1e-2, 1e-1, 1, 10, 100, 300, 600, 900]
87
88     models: np.ndarray = np.empty((16, len(lambdas)), dtype=object)
89
90     min_cost: float = -1

```

```

90     elec_lambda: float = 0
91     eled_grado: int = 0
92     costs = np.empty((16, len(lambdas)))
93
94     for i in range(1, 16):
95         for l in lambdas:
96             pol, scal, model, x_train_aux = train_reg(x_train, y_train, i, l)
97             models[i][lambdas.index(l)] = (pol, scal, model, x_train_aux)
98             cv_cost, train_cost = test(
99                 x_cv, y_cv, x_train_aux, y_train, pol, scal, model)
100             # costs[i][lambdas.index(l)] = cv_cost
101             if min_cost == -1 or cv_cost < min_cost:
102                 min_cost = cv_cost
103                 elec_lambda = l
104                 eled_grado = i
105             print(f"Grado: {i} Lambda: {l}-> Cost: {cv_cost}")
106     print(f"Grado seleccionado: {eled_grado}")
107     print(f"Lambda seleccionado: {elec_lambda}")
108
109     start = time.time()
110     pol, scal, model, x_train_aux = train_reg(
111         x_train, y_train, eled_grado, elec_lambda)
112     end = time.time()
113
114     print(f"Tiempo de entrenamiento: {end-start}")
115     X_train_aux = pol.transform(x_train)
116     # X_train_aux = scal.transform(X_train_aux)
117     y_pred = model.predict(X_train_aux)
118     train_pred = (y_pred == y_train).sum() / len(y_train)
119     print(f"Train pred: {train_pred}")
120     X_cv_aux = pol.transform(x_cv)
121     # X_cv_aux = scal.transform(X_cv_aux)
122     y_pred = model.predict(X_cv_aux)
123     cv_pred = (y_pred == y_cv).sum() / len(y_cv)
124     print(f"CV pred: {cv_pred}")
125     X_test_aux = pol.transform(x_test)
126     # X_test_aux = scal.transform(X_test_aux)
127     y_pred = model.predict(X_test_aux)
128     test_pred = (y_pred == y_test).sum() / len(y_test)
129     print(f"Test pred: {test_pred}")
130     sio.savemat('res/poly.mat', {'train_score': train_pred,
131                                  'cv_score': cv_pred, 'test_score': test_pred, 'best_params': (eled_grado,
132                                  elec_lambda), 'time': end-start})

```

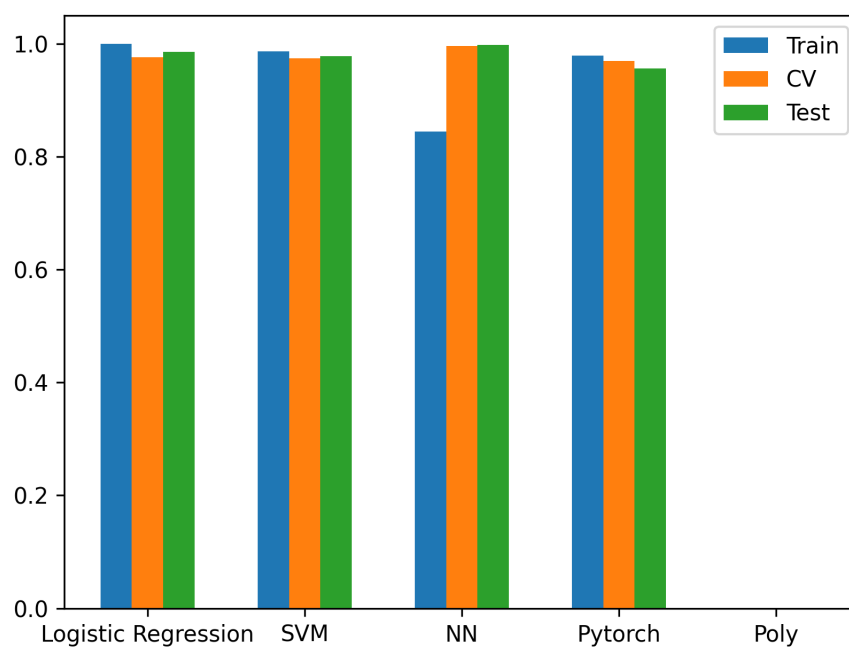


Figura 2.1: Resultados de los distintos modelos en precisión

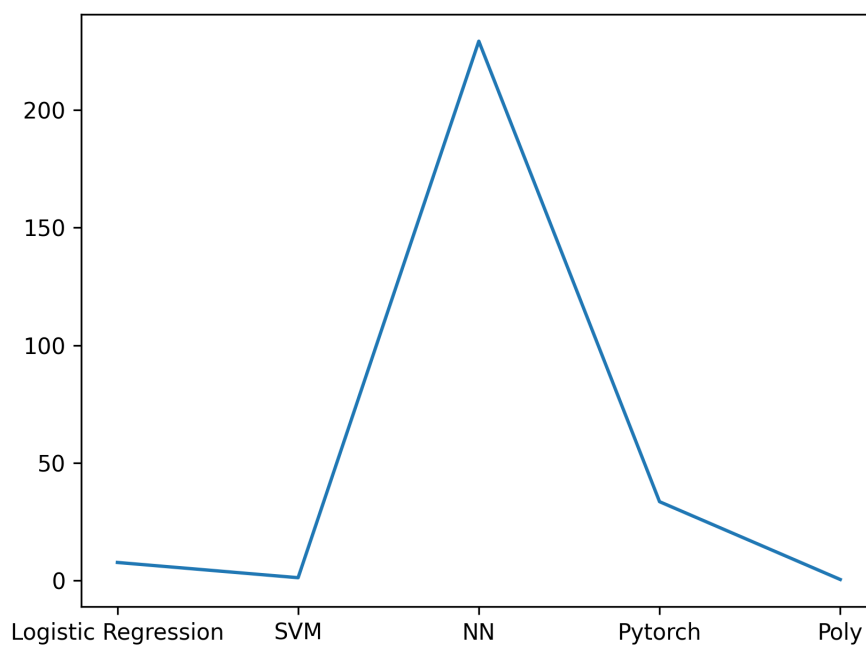


Figura 2.2: Tiempo de entrenamiento de los distintos modelos