Proyecto final: Predicción de bancarrota

Álvar Domingo Fernández y Pablo Jurado López

Introducción y preparación inicial

Para este proyecto hemos usado un set de datos que contiene distintas características de miles de empresas acompañadas de una columna que indica si esas empresas acabaron o no en bancarrota.

Los datos en cuestión han sido extraídos de https://www.kaggle.com/fedesoriano/company-bankruptcy-prediction

Sin embargo, este dataset tiene una peculiaridad: solo cuenta con 220 ejemplos de empresas caídas en bancarrota y unos 6600 que no lo hicieron, por lo que está muy desiquilibrado. Esto puede supone un problema bastante grande para aplicar métodos de aprendizaje automático. Es por ello que hemos recurrido a la técnica SMOTE (Synthetic Minority Oversampling Technique) para poder hacer un estudio más certero de los datos.

En este proyecto aplicaremos tres métodos de aprendizaje automático: regresión logística multiclase, redes neuronales y SVM, y compararemos los resultados obtenidos para el dataset original, un dataset equilibrado manualmente (eliminando gran parte de los ejemplos mayoritarios para que haya igualdad en número) y un dataset equilibrado mediante SMOTE.

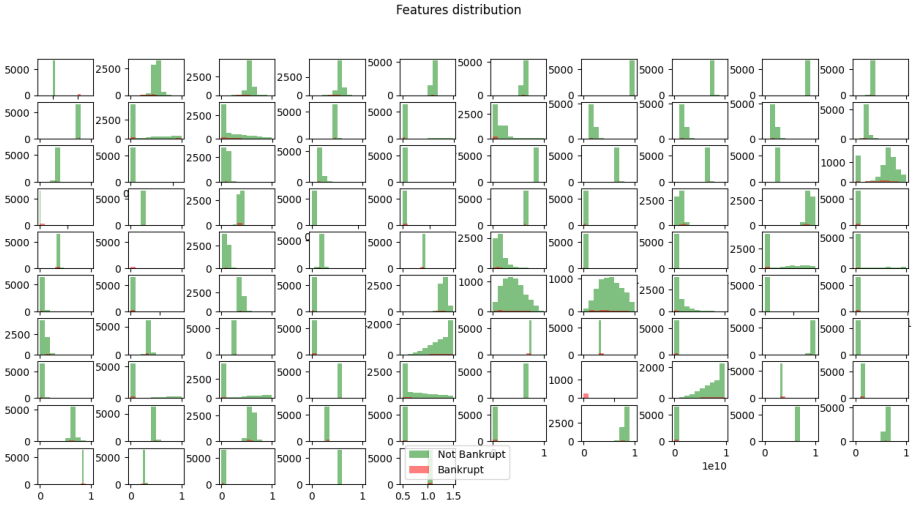
Para programar el proyecto se han utilizado las siguientes librerías y métodos:

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from pandas.io.parsers import read_csv
from mpl_toolkits.mplot3d import Axes3D
import scipy.optimize as opt
import operator
import checkNNGradients as cnn
from sklearn.model_selection import train_test_split
import pandas as pd
import scipy.special as special
from imblearn.over_sampling import SMOTE
from sklearn import preprocessing
from sklearn.svm import SVC
import time
import seaborn as sns
from sklearn.metrics import confusion_matrix
```

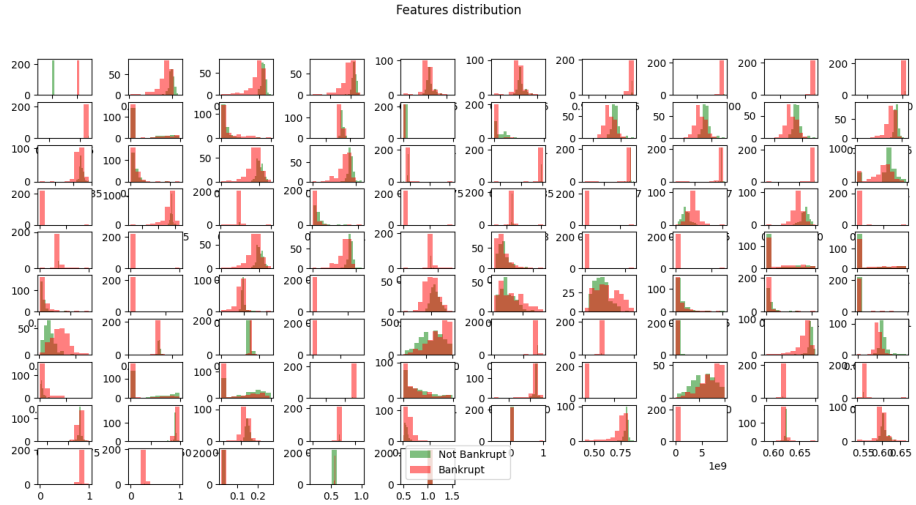
A continuación se muestran tres gráficas que muestran la distribución de los datos en cada dataset que hemos utilizado. Los datos representados en verde son aquellos de empresas que no cayeron

en bancarrota, y los rojos son los datos de empresas que sí lo hicieron.

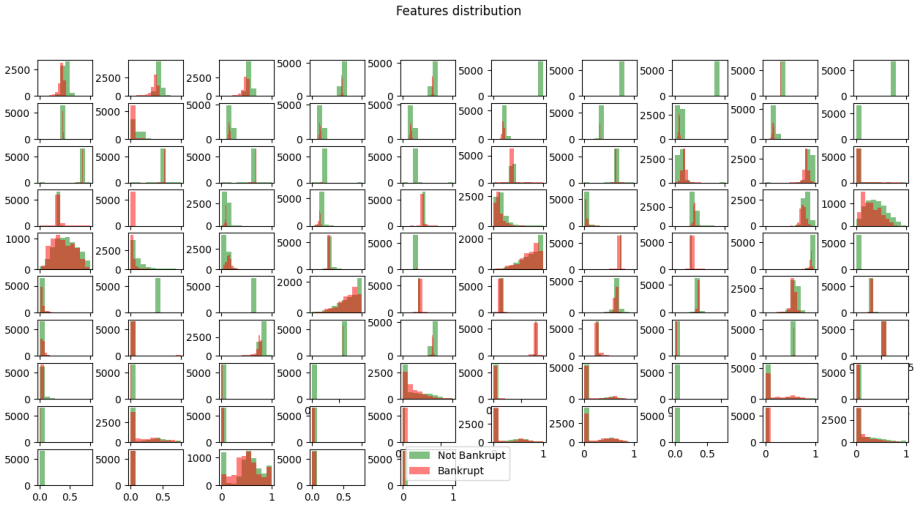
Esta es la gráfica de los datos originales:



A continuación se muestra la gráfica con los datos equilibrados a mano:



Y por último la gráfica con los datos equilibrados con SMOTE



0.1 Equilibrado de datos con SMOTE

Hemos hecho uso de la librería imblearn para aplicar SMOTE a nuestro dataset. Hemos inicializado un obeto sm y luego lo hemos usado para equilibrar el tamaño de las muestras en X e y. Antes de eso hemos tenido que normalizar los datos, para lo cual hemos tenido que separar las columnas de números que están entre 0 y 1 y las que no, para que los números resultantes no estén también desequilibrados:

[ ]:
```
def get_fraction_valued_columns(df):
    my_columns  = []
    for col in df.columns:
        if (df[col].max()<=1) & (df[col].min() >= 0):
            my_columns.append(col)
    return(my_columns)

df = pd.read_csv('data.csv')

cname = 'Bankrupt?'

fractional_columns = get_fraction_valued_columns(df=df.drop([cname],axis=1))
non_fractional_columns = df.drop([cname],axis=1).columns.
 ↪difference(fractional_columns)
norm = preprocessing.normalize(df[non_fractional_columns])
normalized_df = pd.DataFrame(norm, columns=df[non_fractional_columns].columns)

scaled_data = pd.concat([df.
 ↪drop(non_fractional_columns,axis=1),normalized_df],axis = 1)

X = scaled_data.drop(cname, axis = 1)
y = scaled_data[cname]
```

```
sm = SMOTE(random_state=123)
X_sm , y_sm = sm.fit_resample(X,y)
```

Regresión logística multiclase

Para aplicar este método hemos hecho uso del siguiente método:

[ ]:
```
def logistic_regression(X_train, y_train, X_test, y_test, fileName):

        lambdas = np.linspace(1, 3, 20)

        accuracy = []

        for i in lambdas:
                clasificadores = oneVsAll(X_train, y_train, 2, i)
                y_pred, acc = prediccion(X_test, y_test, clasificadores)
                accuracy.append(acc)
                print(acc)
                cm = confusion_matrix(y_test, y_pred)
                plt.figure()
                fig = sns.heatmap(cm, annot=True,fmt="",cmap='Blues').
↪get_figure()
                fig.savefig(fileName + "iter" + str(i) + ".png", dpi=400)

        plt.figure()
        plt.plot(lambdas, accuracy)
        plt.savefig(fileName + "lambdaAccuracyLogistic")
```

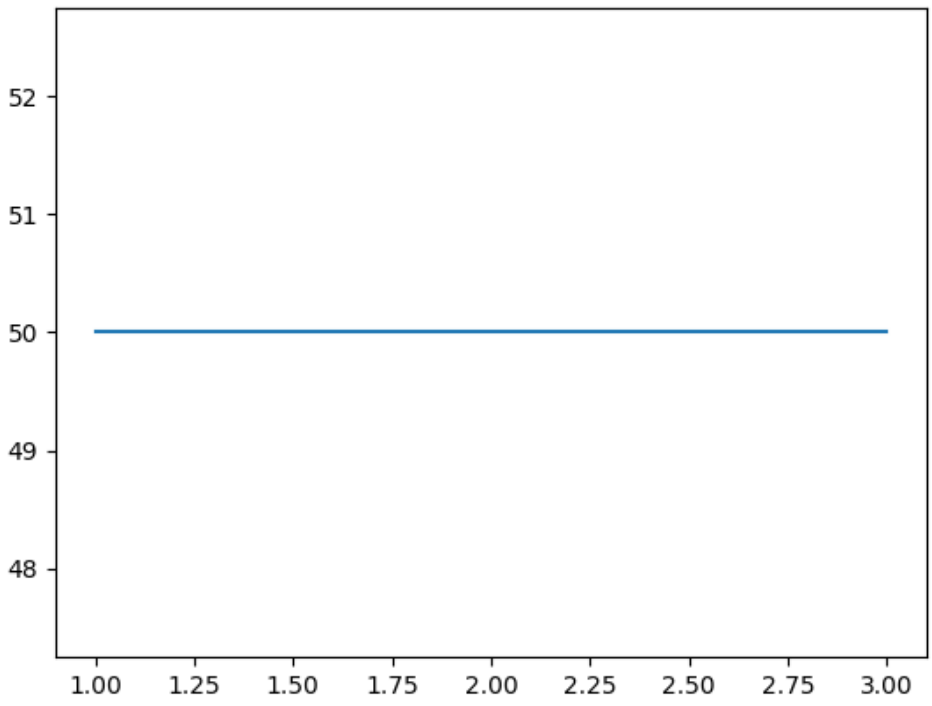
Que utiliza los siguientes métodos auxiliares:

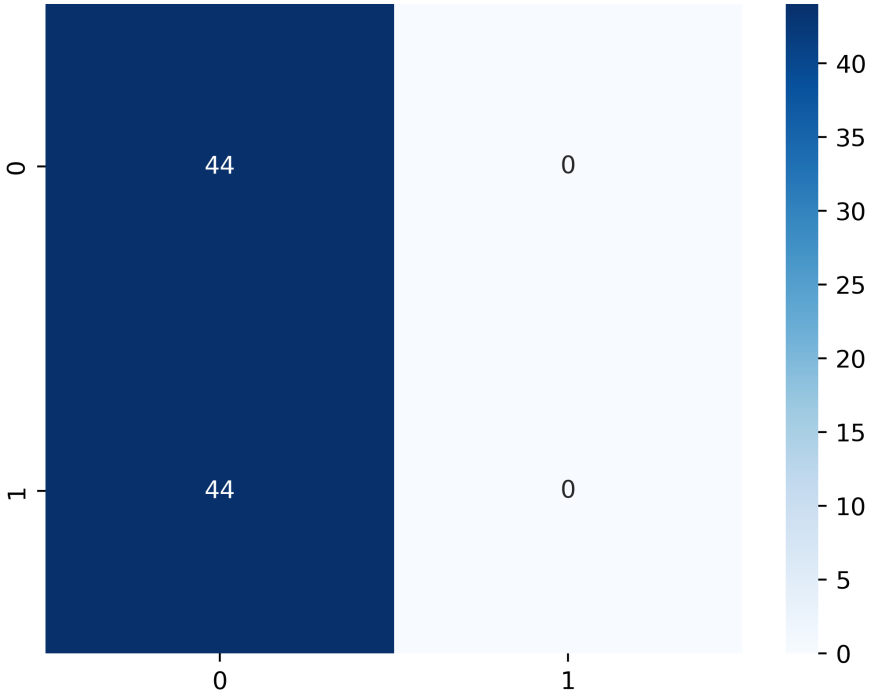
[ ]:
```
def oneVsAll(X, y, num_etiquetas, reg):
        clasificadores = np.zeros(shape=(num_etiquetas, X.shape[1]))

        for i in range(num_etiquetas):
                filtrados = (y==i) * 1
                thetas = np.zeros(np.shape(X)[1])
                clasificadores[i] = opt.fmin_tnc(func=costeReg, x0=thetas,␣
↪fprime=gradienteReg, args=(X, filtrados, reg), messages=0)[0]

        return clasificadores

def prediccion(X, Y, clasificadores):
        predicciones = {}
        Y_pred = []
        for imagen in range(np.shape(X)[0]):
                for i in range(clasificadores.shape[0]):
                        theta_opt = clasificadores[i]
                        prediccion = sigmoide(
```

```
                                np.matmul(np.transpose(theta_opt), X[imagen]))

                        predicciones[i] = prediccion
                Y_pred.append(max(predicciones.items(), key=operator.
 ↪itemgetter(1))[0])
        return Y_pred, np.sum((Y == np.array(Y_pred)))/np.shape(X)[0] * 100

def costeReg(thetas, x, y, lamb):
        sigXT = sigmoide(np.dot(x, thetas))
        a1 = (-1/np.shape(x)[0])
        a2 = np.dot(np.log(sigXT), y)
        a3 = np.dot(np.log(1-sigXT+1e-6), 1-y)
        a = a1 * (a2+a3)
        b = ((lamb/(2*np.shape(x)[0])) * np.sum(thetas ** 2))
        return a + b

def gradienteReg(thetas, x, y, lamb):
        sigXT = sigmoide(np.matmul(x, thetas))
        a = ((1/np.shape(x)[0]) * np.matmul(np.transpose(x), (sigXT - y))) +␣
 ↪((lamb/np.shape(x)[0]) * thetas)
        return a

def sigmoide(x):
        return special.expit(x)
```

Los resultados son los siguientes:

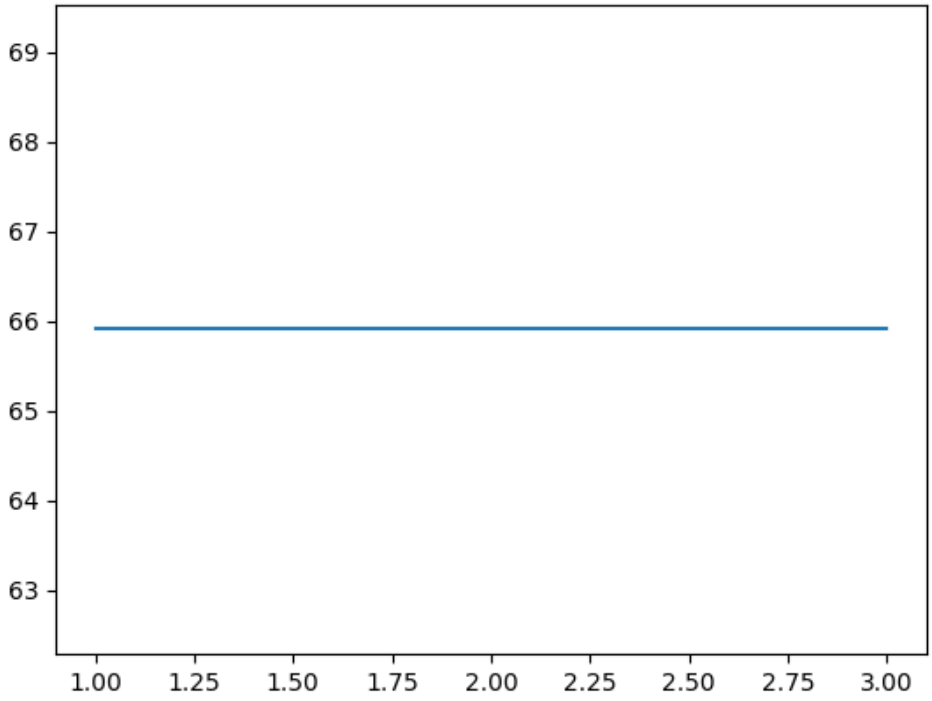
Dataset original:

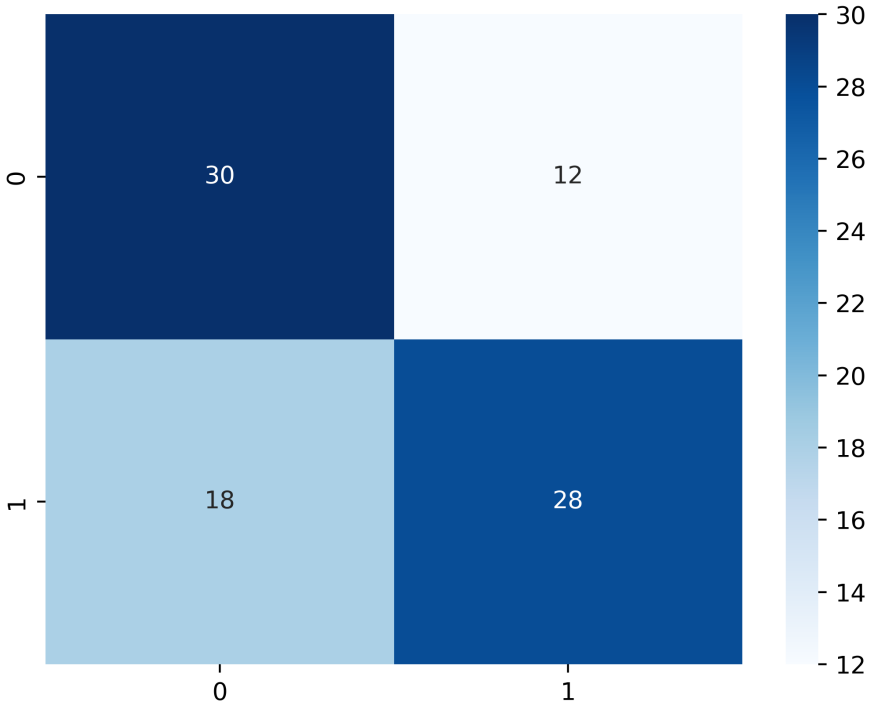

52
51
50
49
48
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00


44
0
44
0
0
1
0
1
40
35
30
25
20
15
10
5
0

Podemos observar que con este entrenamiento siempre se predice que la compañía no está en bancarrota, ya que al haber muchos más ejemplos de esto que del caso contrario, las predicciones están muy sesgadas hacia el caso negativo. De hecho, si utilizáramos una porción del dataset original como casos de prueba, obtendríamos una precisión muy alta, de alrededor del 97%, pero esto se debe a que siempre se predice el caso negativo y el dataset consta de 97% casos negativos. Para resaltar que la predicción no es buena, se han elegido casos de prueba que sean 50% positivos y 50% negativos. El hecho de que la precisión no cambie respecto a lambda posiblemente se debe a que el sesgo es tan grande que siempre se predice el caso negativo, además de que hay muy pocos casos de prueba.

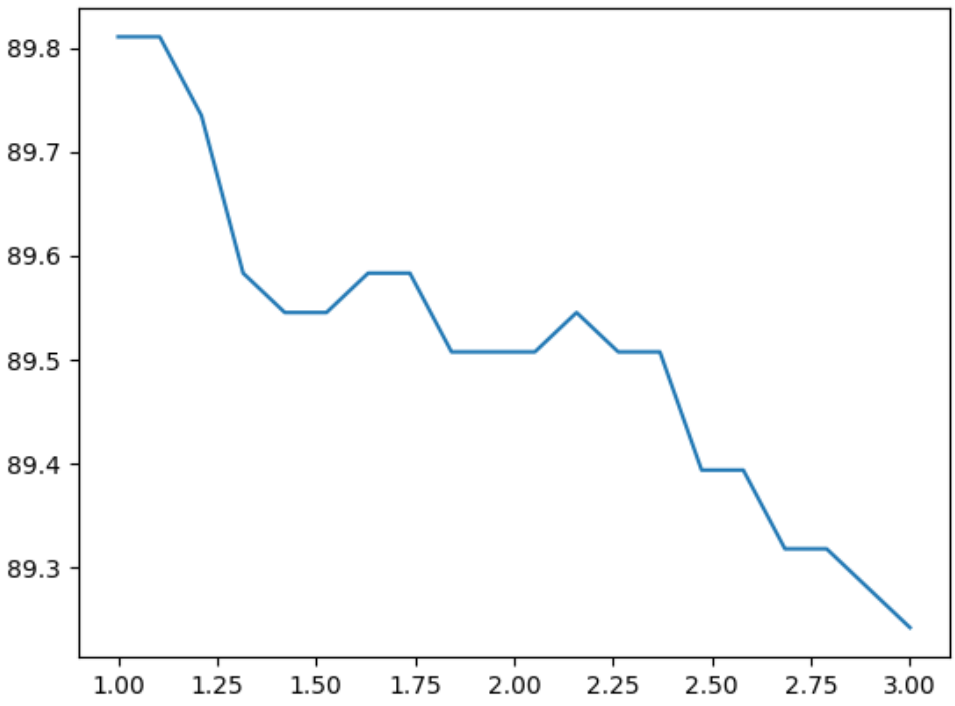
Dataset balanceado a mano:

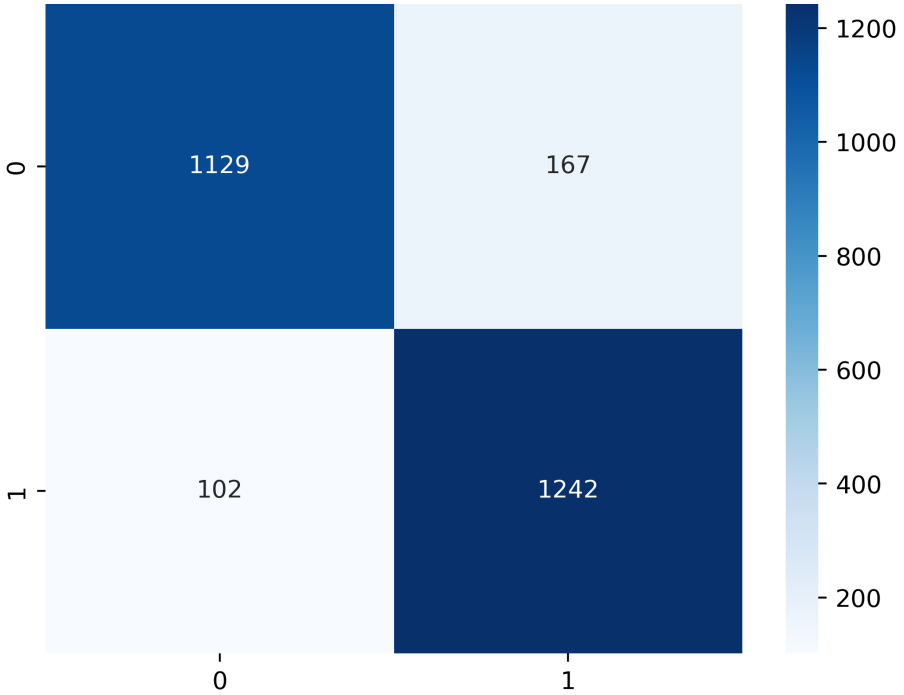




La precisión ha mejorado bastante, estando ahora alrededor del 66% y por la matriz de confusión podemos ver que ahora a veces predice que sí y a veces que no. Sin embargo, sigue sin ser una probabilidad muy alta ya que, por cortar el dataset para que haya la misma proporción de 0s y 1s, tan solo hay unos 400 ejemplos de entrenamiento, que no son suficientes para alcanzar una mayor precisión. También podemos observar que la precisión no cambia respecto a lambda, probablemente debido al pequeño número de casos de prueba, ya que con tan pocos, la precisión subiría por más de 1% por cada caso acertado, y estos valores de lambda no parecen ser suficientes para crear un cambio tan grande.

Dataset balanceado con SMOTE:


89.8
89.7
89.6
89.5
89.4
89.3
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00


1129
167
102
1242
0
1
0
1
1200
1000
800
600
400
200

Finalmente, los resultados con el dataset balanceado con SMOTE son mucho mejores. La precisión llega casi a 90% gracias a contar ahora con muchos más ejemplos para el entrenamiento, y además, como tiene más casos de prueba, el efecto de lambda sí es apreciable. En general vemos que lambdas más pequeñas dan mayor precisión.

Redes neuronales

Hemos aplicado el método de la red neuronal con el siguiente código:

[ ]:
```
def neural_network(X, y, fileName):

        m = len(y)
        input_size = X.shape[1]
        num_labels = 10
        num_ocultas = 25

        y_onehot = np.zeros((m, num_labels))

        for i in range(m):
                y_onehot[i][int(y[i])] = 1

        theta1 = random_weights(num_ocultas, input_size + 1)
        theta2 = random_weights(num_labels, num_ocultas +1)
        params_rn = np.concatenate((np.ravel(theta1), np.ravel(theta2)))
        reg_param = 1

        cost, grad = backprop(params_rn, input_size, num_ocultas,
                                                num_labels, X, y_onehot,␣
↪reg_param)
        a = cnn.checkNNGradients(backprop, 1)
        print(a)

        fmin = opt.minimize(fun=backprop, x0=params_rn, args=(input_size,␣
↪num_ocultas, num_labels,
                                                X, y_onehot, reg_param),␣
↪method='TNC', jac=True, options={'maxiter': 70})

        theta1_opt = np.reshape(
                fmin.x[:num_ocultas * (input_size + 1)], (num_ocultas,␣
↪(input_size + 1)))
        theta2_opt = np.reshape(
                fmin.x[num_ocultas * (input_size + 1):], (num_labels,␣
↪(num_ocultas + 1)))

        a1, a2, h = propagar(X, theta1_opt, theta2_opt)

        print("El porcentaje de acierto del modelo es: {}%".format(
```

```
                np.sum((y == predict_nn(X, h)))/X.shape[0] * 100))

        lambdas = np.linspace(0, 1, 10)

        accuracy = []

        for lamb in lambdas:
                print("Voy por ", lamb, " de ", len(lambdas))
                reg_param = lamb
                fmin = opt.minimize(fun=backprop, x0=params_rn,␣
↪args=(input_size, num_ocultas, num_labels,
                                                        X, y_onehot, reg_param),␣
↪method='TNC', jac=True, options={'maxiter': 70})

                theta1_opt = np.reshape(
                        fmin.x[:num_ocultas * (input_size + 1)], (num_ocultas,␣
↪(input_size + 1)))
                theta2_opt = np.reshape(
                        fmin.x[num_ocultas * (input_size + 1):], (num_labels,␣
↪(num_ocultas + 1)))

                a1, a2, h = propagar(X, theta1_opt, theta2_opt)
                accuracy.append((np.sum((y == predict_nn(X, h)))/X.shape[0] *␣
↪100))

        plt.plot(lambdas, accuracy)
        plt.savefig(fileName + "lambdaAccuracy")

        lambdas = np.linspace(10, 70, 7)

        accuracy = []

        for lamb in lambdas:
                reg_param = lamb
                fmin = opt.minimize(fun=backprop, x0=params_rn,␣
↪args=(input_size, num_ocultas, num_labels,
                                                        X, y_onehot, reg_param),␣
↪method='TNC', jac=True, options={'maxiter': int(lamb)})

                theta1_opt = np.reshape(
                        fmin.x[:num_ocultas * (input_size + 1)], (num_ocultas,␣
↪(input_size + 1)))
                theta2_opt = np.reshape(
                        fmin.x[num_ocultas * (input_size + 1):], (num_labels,␣
↪(num_ocultas + 1)))
```

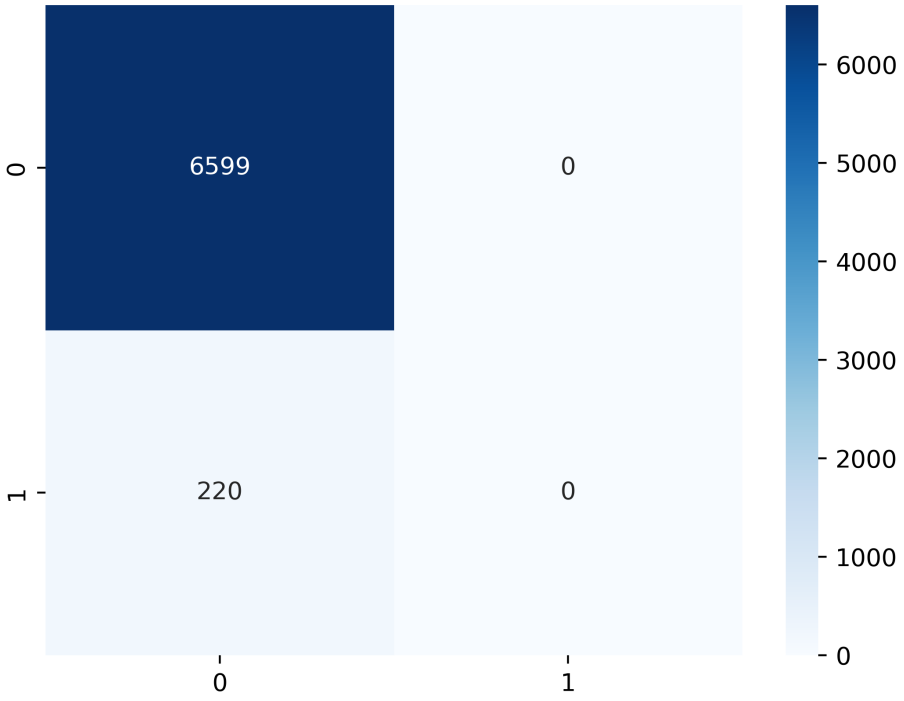
```
                a1, a2, h = propagar(X, theta1_opt, theta2_opt)
                accuracy.append((np.sum((y == predict_nn(X, h)))/X.shape[0] *␣
↪100))
        plt.figure()
        plt.plot(lambdas, accuracy)
        plt.savefig(fileName + "iterationAccuracy")
```

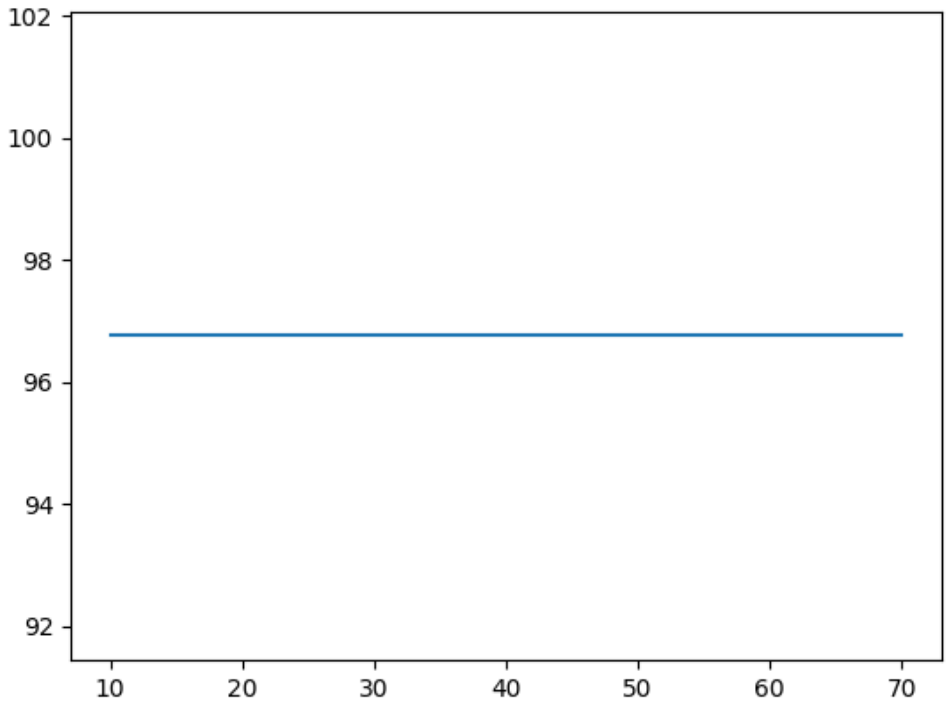
Que utiliza los siguientes métodos auxiliares:

[ ]:
```
def sigmoide_prima(x):
        return sigmoide(x) / (1 - sigmoide(x))

def propagar(X, theta1, theta2):
        m = np.shape(X)[0]

        a1 = np.hstack([np.ones([m, 1]), X])
        z2 = np.dot(a1, theta1.T)
        a2 = np.hstack([np.ones([m, 1]), sigmoide(z2)])
        z3 = np.dot(a2, theta2.T)
        a3 = sigmoide(z3)
        return a1, a2, a3


def coste_neuronal(X, y, theta1, theta2, reg):
        a1, a2, h = propagar(X, theta1, theta2)
        m = X.shape[0]

        J = 0
        for i in range(m):
                J += np.sum(-y[i]*np.log(h[i]) - (1 - y[i])*np.log(1-h[i]+1e-9))
        J = J/m

        sum_theta1 = np.sum(np.square(theta1[:, 1:]))
        sum_theta2 = np.sum(np.square(theta2[:, 1:]))

        reg_term = (sum_theta1 + sum_theta2) * reg / (2*m)

        return J + reg_term


def gradiente(X, y, Theta1, Theta2, reg):
        m = X.shape[0]

        delta1 = np.zeros(Theta1.shape)
        delta2 = np.zeros(Theta2.shape)

        a1, a2, h = propagar(X, Theta1, Theta2)
```

```
        for t in range(m):
                a1t = a1[t, :]
                a2t = a2[t, :]
                ht = h[t, :]
                yt = y[t]
                d3t = ht - yt
                d2t = np.dot(Theta2.T, d3t) * (a2t * (1 - a2t))

                delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])
                delta1[:, 1:] += Theta1[:, 1:] * reg/m
                delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])
                delta2[:, 1:] += Theta2[:, 1:] * reg/m

        return np.concatenate((np.ravel(delta1/m), np.ravel(delta2/m)))


def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
        Theta1 = np.reshape(
                params_rn[:num_ocultas * (num_entradas + 1)], (num_ocultas,␣
 ↪(num_entradas + 1)))
        Theta2 = np.reshape(
                params_rn[num_ocultas * (num_entradas + 1):], (num_etiquetas,␣
 ↪(num_ocultas+1)))
        return coste_neuronal(X, y, Theta1, Theta2, reg), gradiente(X, y,␣
 ↪Theta1, Theta2, reg)


def random_weights(L_in, L_out):
        epsilon = np.sqrt(6)/np.sqrt(L_in + L_out)
        return np.random.random((L_in, L_out)) * epsilon - epsilon/2


def predict_nn(X, h):
        return [(np.argmax(h[image])) for image in range(X.shape[0])]
```

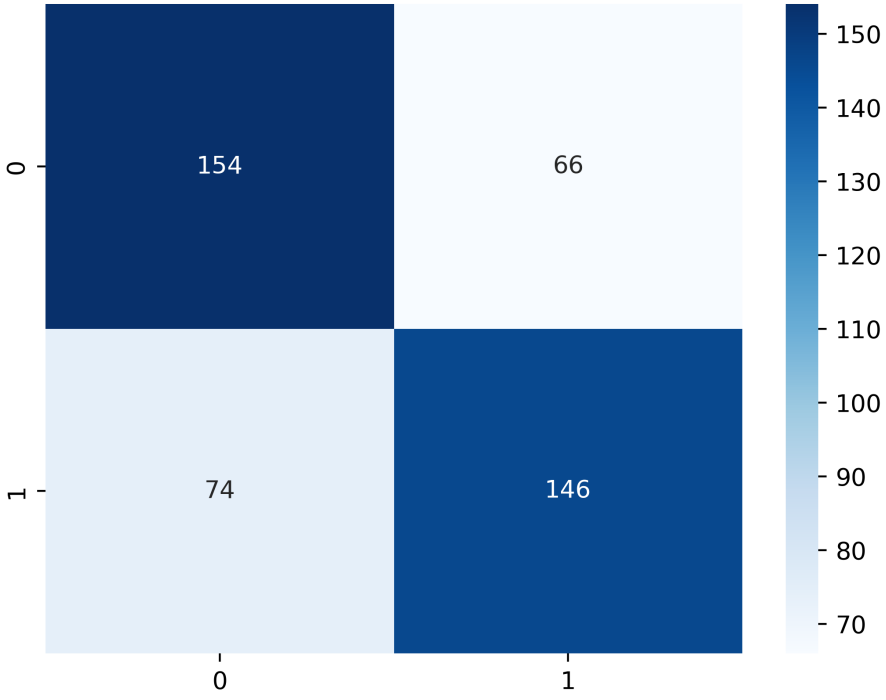
Dataset original:

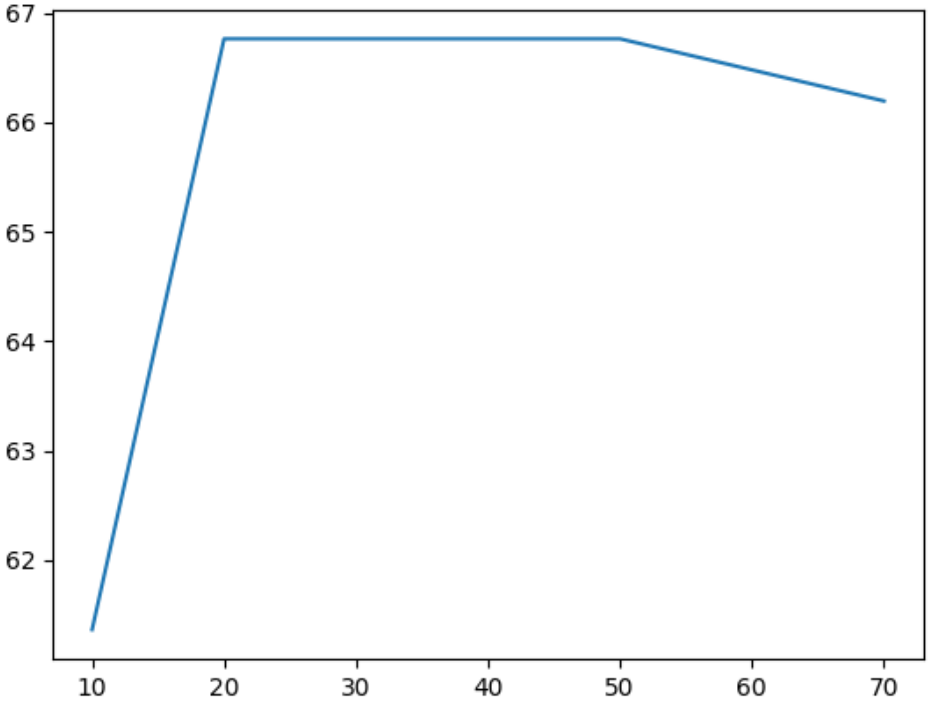

6599
0
220
0
0
1
0
1
6000
5000
4000
3000
2000
1000
0

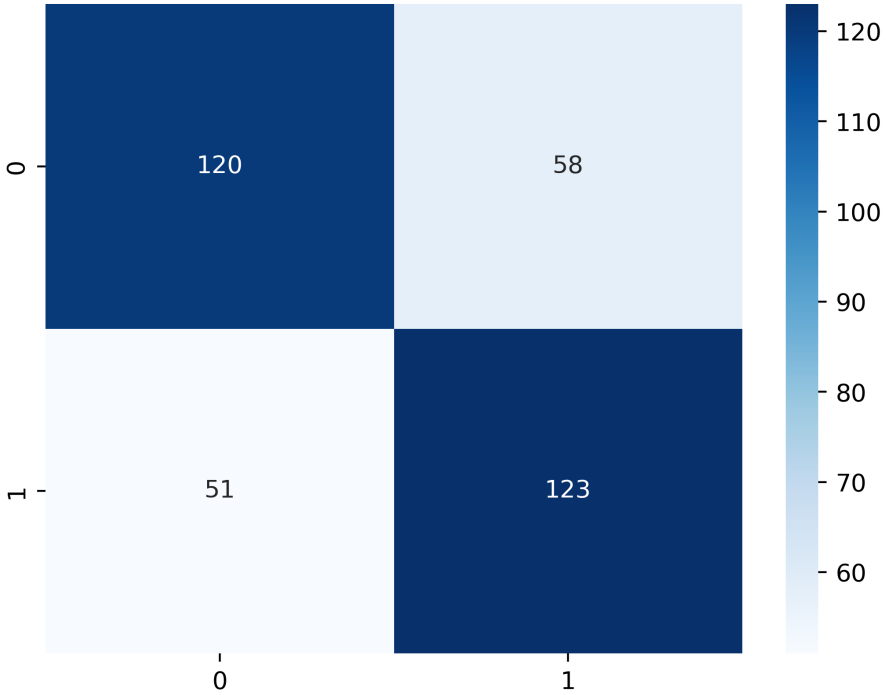

102
100
98
96
94
92
10
20
30
40
50
60
70

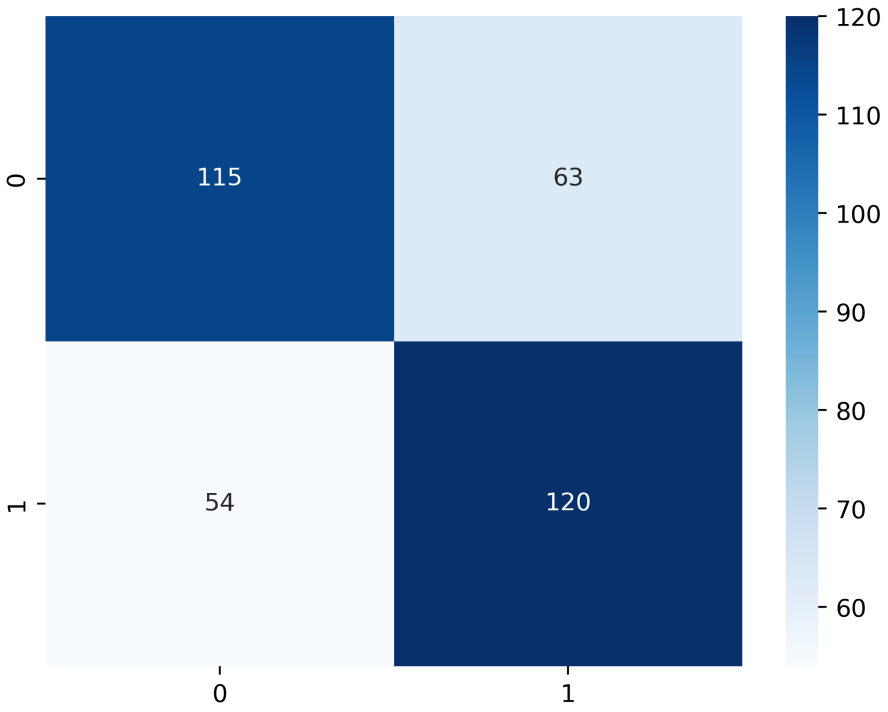
Aquí se da el mismo caso que en la regresión logística. Sin embargo, al no ser tan fácil cambiar a nuestro gusto la cantidad de datos de prueba de cada clase, podemos observar lo que pasa si todo se deja tal cual. La proporción de aciertos es muy alta, pero esto se debe únicamente a que hay muchos más ejemplos de empresas que no han caído en bancarrota, y nuestro modelo está tan sesgado que siempre dice que la empresa no está en bancarrota y acierta la mayoría de las veces.

Dataset balanceado a mano:



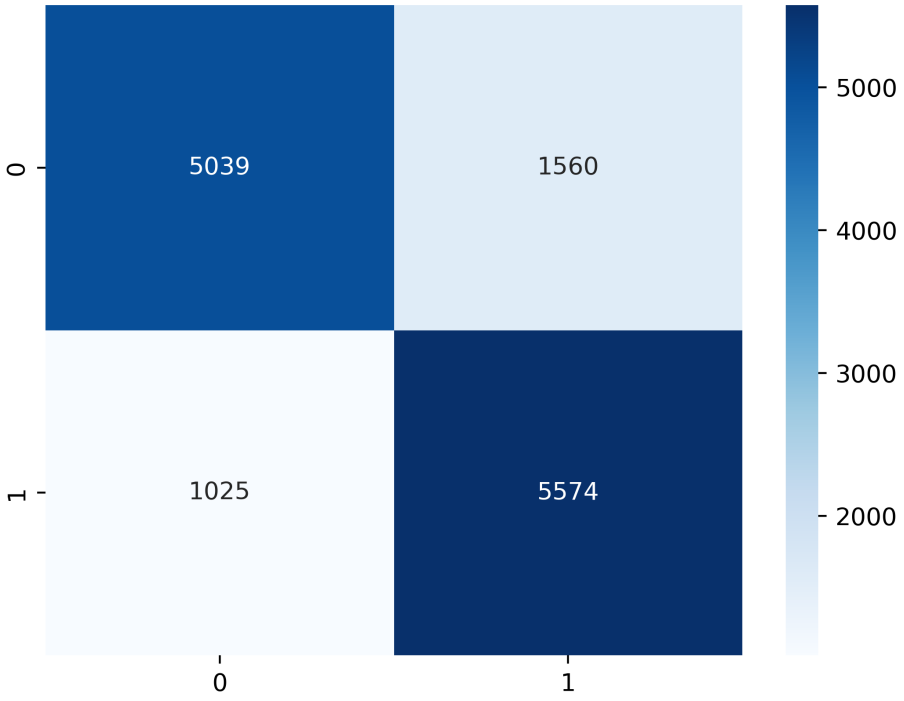

67
66
65
64
63
62
10
20
30
40
50
60
70


120
58
51
123
0
1
0
1
120
110
100
90
80
70
60



Al igual que ocurrió con la regresión logística, el dataset balanceado predice realmente en vez de decir siempre que no, pero no llega a ser muy preciso. Un gráfico muestra la mejoría en precisión según el número de iteraciones que no consigue llegar a 70% de precisión. Las otras matrices de confusión son las de la mejor lambda y el mejor número de iteraciones, respectivamente.

Dataset balanceado con SMOTE:



Al igual que pasó antes, los datos balanceados con Smote dan resultados mucho mejores que incluso rozan el 90% de precisión.

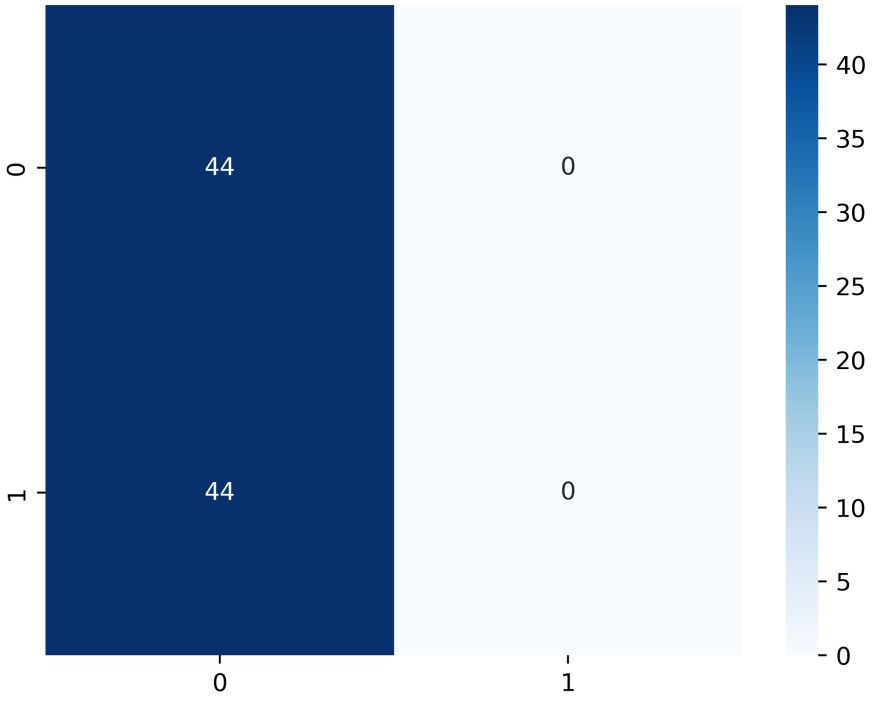
Support Vector Machine

Para aplicar la técnica de las SVM, hemos hecho uso del siguiente código:

[ ]:
```
def svm(X, y, fileName):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=1)
        X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
 ↪test_size=0.25, random_state=1)

        values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300]
        n = len(values)
        scores = np.zeros((n, n))

        startTime = time.process_time()

        for i in range(n):
                C = values[i]
                print("Voy por el i ", i, "de " , n)
                for j in range(n):
```

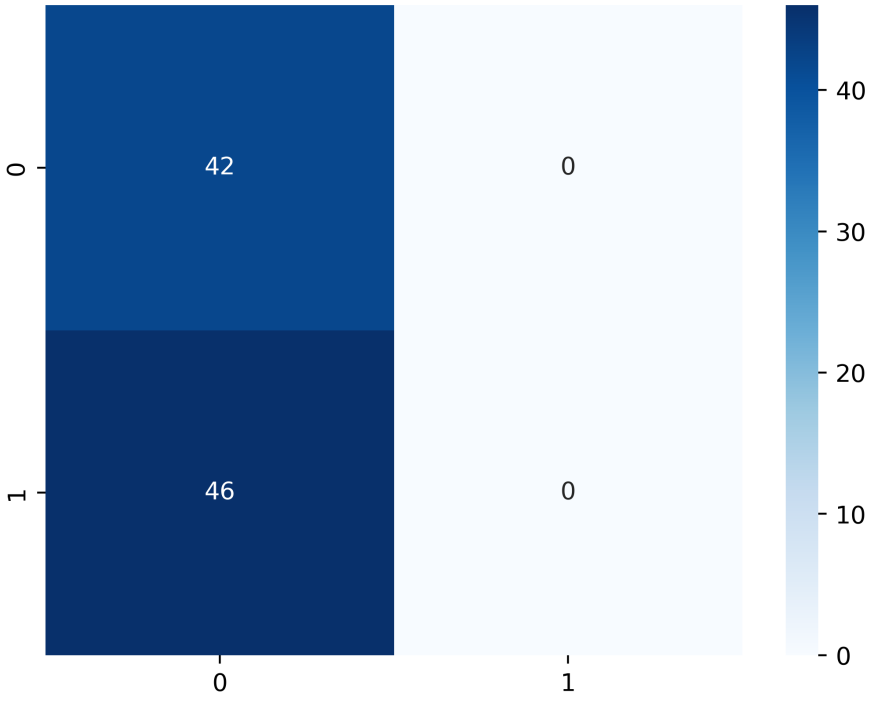
```
                        print("Voy por el j ", j, "de " , n)
                        sigma = values[j]
                        svm = SVC(kernel='rbf', C = C, gamma= 1 / (2 * sigma␣
↪**2))
                        svm.fit(X_train, y_train)
                        scores[i, j] = svm.score(X_val, y_val)

        print("Error mínimo: {}".format(1 - scores.max()))
        C_opt = values[scores.argmax()//n]
        sigma_opt = values[scores.argmax()%n]
        print("C óptimo: {}, sigma óptimo: {}".format(C_opt, sigma_opt))

        svm = SVC(kernel='rbf', C= C_opt, gamma=1 / (2 * sigma_opt)**2)
        svm.fit(X_train, y_train)
        y_pred = svm.predict(X_test)
        endTime = time.process_time()

        score = svm.score(X_test, y_test)
        totalTime = endTime - startTime

        print('Precisión: {:.3f}%'.format(score*100))
        print('Tiempo de ejecución: {}'.format(totalTime))
        print('Matriz de confusión: ')
        cm = confusion_matrix(y_test, y_pred)
        fig = sns.heatmap(cm, annot=True,fmt="",cmap='Blues').get_figure()
        fig.savefig(fileName, dpi=400)
```

Dataset original:

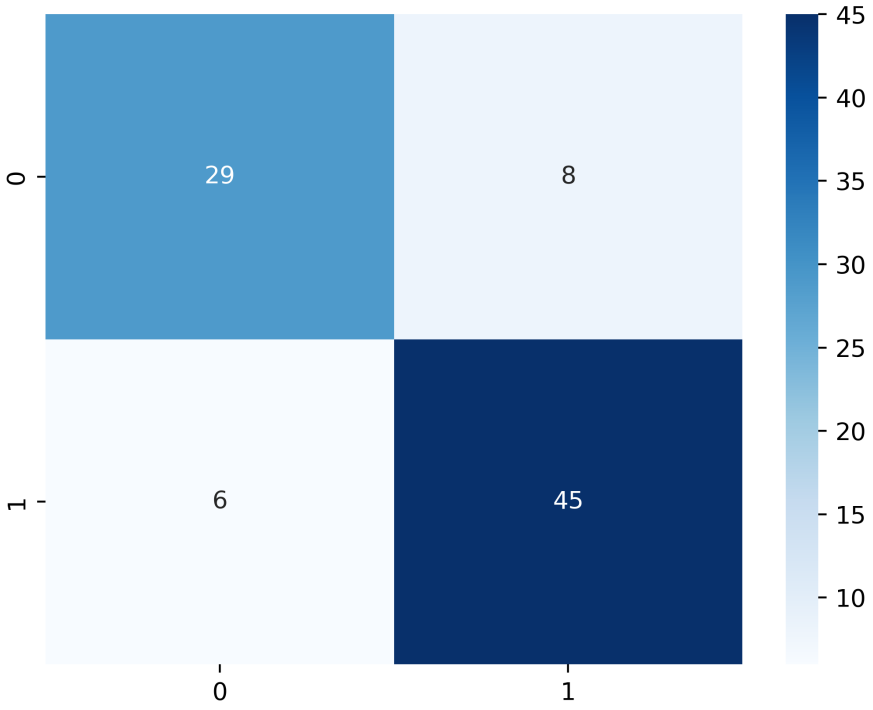


Pasa lo mismo que en los dos casos anteriores. El modelo ha recibido tantos datos de empresas que no están en bancarrota que siempre clasifica a todos los ejemplos como 'no en bancarrota' (0).

Dataset balanceado a mano:

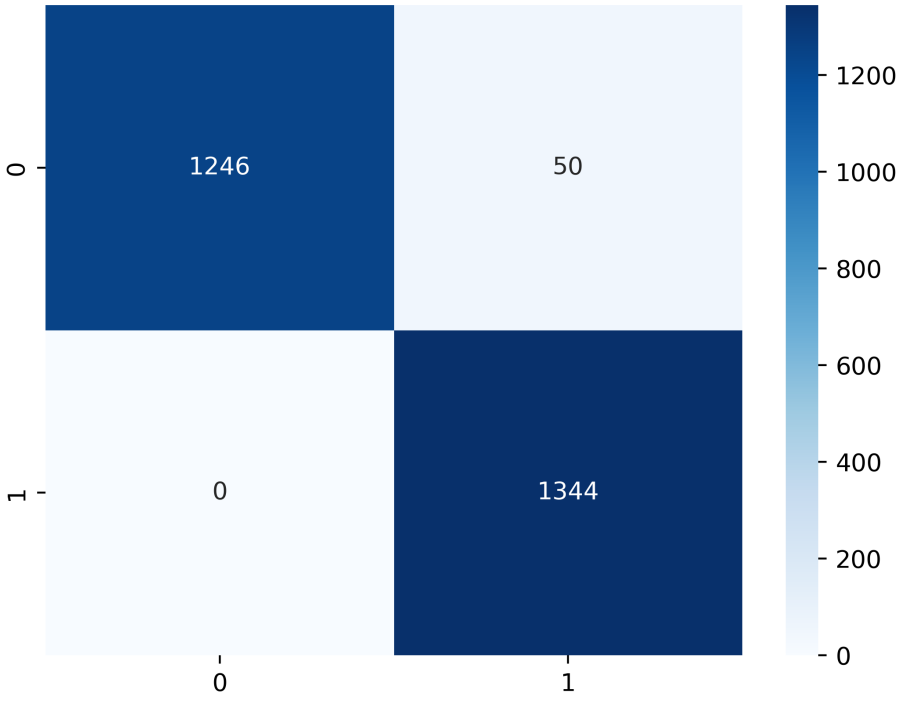


Este es un caso curioso porque no actúa de forma similar a los anteriores casos de estudio con este dataset, sino que también elige siempre 0. Esto se debe a que no se han normalizado los datos y por lo tanto los cálculos no son correctos. A continuación se muestra el gráfico resultante cuando los datos sí están normalizados.



Tras normalizar, incluso con solo 440 ejemplos conseguimos una precisión de alrededor del 84%. Esto demuestra la importancia de normalizar los datasets.

Dataset balanceado con SMOTE:



El modelo smote sigue siendo el más eficaz, teniendo un porcentaje de aciertos incluso mayor que la red neuronal (alrededor del 98%)

Conclusión

Como hemos podido observar, las predicciones son mejores cuando los datos de aprendizaje son abundantes, normalizados y bien equilibrados. Cuando hay muy pocos datos, los resultados son menos precisos, y cuando no están equilibrados, están muy sesgados a favor del elemento mayoritario en los datos de aprendizaje.

También podemos observar que la técnica más precisa en general ha sido SVM (aunque era la más dependiente de una buena normalización), seguida de redes neuronales y finalmente regresión logística.