ARCENE DATASET - PROYECTO FINAL DE MACHINE LEARNING AND BIG DATA

January 24, 2019

ÍNDICE

1 TRATAMIENTO DE LOS EJEMPLOS DE ENTRADA

1.1 EXTRACCIÓN DE LOS DATOS OFRECIDOS

ARCENE es un proyecto científico dedicado a la detección del cancer de ovarios y prósrata.La clasificación de potenciales pacientes se realiza según unos datos de espectrometría de masa de los mismos, recogidos en el Data Set que se nos facilitó.

El *Data Set* ARCENE,con el que vamos a trabajar en este proyecto, es uno de los cinco *Data Set* del "NIPS 2003 feature selection challenge"; preparado por Isabelle Guyon. El *Data Set* se divide en tres a su vez: - ***arcene_train.data***: conjunto de ejemplos de entrenamiento para diseñar nuestro clasificador de conjuntos de personas con cáncer y sin cáncer - ***arcene_valid.data***: conjunto de ejemplos de validación que nos permite calcular el porcentaje de acierto de nuestro clasificador y evaluar las hipótesis y decisiones iniciales tomadas relativas a las técnicas de entrenamiento, parámetros usados para funciones y cualquier decisión que influya en la precisión de nuestro sistema de aprendizaje automático. - ***arcene_test.data***: Se ofrece como un conjunto de datos de entrada reales para los que nuestro clasificador tendría que predecir en qué conjunto se encasilla cada ejemplo de entrada.

Por otro lado, disponemos de las etiquetas o salidas esperadas (variables $\boldsymbol{y}$) del Training Set y Validation Set. - ***arcene_train.labels*** - ***arcene_valid.labels***

Dado que este Data Set estaba orientado a ser usado por los participantes de en la competición, no disponemos del archivo ***"arcene_test.labels"***. Solo nos valdremos de los arhivos anteriores mencionados para diseñar el sistema de aprendizaje automático. En su lugar hemos decido escoger un 50% de los ejemplos de validaión y convertirlos en nuestros ejemplos de test final.

ESTRUCTURA

De los 200 registros útiles de pacientes o ejemplos de los que disoponemos(descontando de los 900 registros totales, los 700 ejemplos del test a los que no podemos acceder a su salida esperada), los estructuramos de la siguiente forma: - 100 son destinados a ser ejemplos de entrenamiento para nuestro clasificador - 50 corresponden con los ejemplos de validación - 50 se reservan para el test final

Cada uno de los registros de personas están formados por 10.000 características o atributos, de las cuales 3000 se añadieron de forma adicional y no tienen valor para la predcción de nuestro sistema.

Una buena idea si disponemos de una CPU que soporte la operación, sería realizar pruebas sobre el entrenamiento del sistema escogiendo diferentes permutaciones de los atributos, subconjuntos del conjunto total de 100000 características. Sin embargo, en nuestro caso no disponemos de tal y continuaremos con todos los atributos analizando los resultados que obtengamos.

Enlace a la descripción de las matrices con los datos de entrada arcene.param

1.1.1 Paso 1- lectura de los data-set ofrecidos.

```
In [32]: #Para la visualización de gráficos
         import matplotlib.pyplot as plt

         #Procesamiento de datos
         import numpy as np
         from pandas.io.parsers import read_csv
         import scipy.optimize as opt
         from sklearn.preprocessing import PolynomialFeatures as pf
         from scipy.io import loadmat
         from sklearn import decomposition
         #Libreria para support vector machine
         from sklearn.svm import SVC
         from sklearn.model_selection import train_test_split

         #Documentación del Código
         #para la traducción del código latex
         #from IPython.display import display, Math
         #import pdb; pdb.set_trace()
```

```
In [2]: def loadMatrix(file):
            return np.loadtxt(file)
```

```
In [3]: def lectura_Train_Set():
            X = loadMatrix('arcene_train.data')
            y = loadMatrix('arcene_train.labels').astype(int)
            return (X,y)
```

```
In [4]: def lectura_Valid_Set():
            cvX = loadMatrix('arcene_valid.data')
            cvY = loadMatrix('arcene_valid.labels').astype(int)
            return (cvX[:50],cvY[:50])

In [5]: def lectura_Test_Set():
            x_test = loadMatrix('arcene_valid.data')
            y_test = loadMatrix('arcene_valid.labels').astype(int)
            return (x_test[50:],y_test[50:])

In [6]: lectura_Train_Set()
        # A continuación se aprecia el tamaño de la matriz de datos de entrada y los valores d
        #labels correspondientes

Out[6]: (array([[  0.,  71.,   0., ...,   0.,   0., 524.],
                [  0.,  41.,  82., ...,   0., 284., 423.],
                [  0.,   0.,   1., ...,   0.,  34., 508.],
                ...,
                [  2.,  15.,  48., ...,   0.,   0., 453.],
                [  8.,   0.,  38., ...,   0., 189., 403.],
                [  0.,   0.,   0., ...,   0.,  10., 365.]]),
         array([ 1, -1,  1,  1, -1, -1,  1, -1, -1, -1, -1,  1, -1,  1, -1,  1, -1,
                -1, -1, -1, -1, -1, -1,  1, -1, -1,  1, -1,  1, -1,  1,  1,  1, -1,
                -1,  1, -1, -1,  1, -1,  1, -1, -1,  1, -1, -1, -1, -1,  1,  1, -1,
                 1, -1, -1,  1, -1,  1,  1,  1, -1,  1,  1, -1,  1, -1, -1, -1, -1,
                 1,  1, -1,  1, -1, -1,  1, -1, -1,  1, -1,  1,  1,  1, -1,  1,  1,
                -1,  1,  1, -1, -1,  1, -1,  1,  1, -1, -1, -1,  1, -1,  1]))

In [7]: lectura_Valid_Set()

Out[7]: (array([[  0.,   0., 156., ...,   0.,   0., 465.],
                [  0.,   7.,   0., ...,   0.,  34., 199.],
                [  0.,  32.,   0., ...,   0.,  47., 219.],
                ...,
                [  0.,   0.,   0., ...,   0.,   0., 313.],
                [ 45.,  28., 240., ...,   0.,  31., 414.],
                [ 13.,  58.,   0., ...,   0.,  75., 319.]]),
         array([-1, -1, -1,  1,  1,  1, -1,  1, -1, -1,  1, -1, -1,  1, -1, -1,  1,
                 1,  1,  1,  1, -1,  1, -1,  1, -1, -1, -1, -1, -1, -1, -1,  1,  1,
                 1, -1, -1, -1,  1, -1, -1,  1, -1, -1,  1, -1, -1,  1,  1, -1]))

In [8]: lectura_Test_Set()

Out[8]: (array([[ 14.,   0.,  97., ...,   0., 202., 383.],
                [  0.,   0.,   0., ...,  14., 149., 362.],
                [ 65.,   4., 168., ...,   0., 204., 328.],
                ...,
                [ 93.,  32., 137., ...,   0., 276., 312.],
                [119.,  12., 198., ...,   0.,   0., 350.],
```

```
        [112.,  19., 171., ...,   0.,   0., 367.]]),
array([-1,  1,  1, -1, -1, -1, -1,  1,  1,  1, -1,  1,  1, -1, -1, -1, -1,
        1, -1, -1,  1, -1,  1, -1, -1,  1, -1, -1,  1,  1,  1, -1, -1,  1,
        1, -1,  1,  1, -1, -1,  1, -1,  1,  1, -1, -1, -1,  1,  1, -1]))
```

2 IMPLEMENTACIÓN DE TÉCNICAS DE APRENDIZAJE AUTOMÁTICO

En la asignatura de Machine Learning hemos visto las siguientes técnicas aplicables al entrenamiento para la clasificación de unos datos de entrada en dos o varios grupos o clusters:

1 Regresión lineal con varias variables

2 Regresión logística

```
1 Redes Neuronales

2 Support Vector Machines(SVM)

3 Clustering(k-means algorithms)
```

De estas técnicas emplearemos regresión logística para entrenar una red neuronal y para aplicar Support Vector Machines en nuestro problema (gaussian_kernel), ya que tenemos una salida de valores discretos, representando paciente con cáncer y paciene sin cáncer. Esto nos permitirá comparar ambas técnicas y analizar el porcentaje de acierto respecto a los ejemplos de entrenamiento de un mecanismo y de otro.

Por otro lado clustering se aplica cuando disponemos de los datos de entrada $\boldsymbol{X}$ pero no de los datos de salida $\boldsymbol{y}$ para el entrenamiento.Este no es el caso de nuestro proyecto, de modo que tampoco necesitamos aplicar una técnica de entrenamiento no supervisado como la anterior.

2.0.1 Redes Neuronales

La red neuronal que aplicamos es la misma que hemos aplicado en prácticas (Código reutilizado de la práctica 4) de la Asignatura de Machine Learning and Big Data. Esta Red neuronal está formada por una capa de entrada, una capa oculta y una capa de salida

Para aprender los valores de óptimos , primero les damos unos valores aleatorios de partida en la función ***"pesos aleatorios"*** y aplicamos la función *optimize* de sklearn dentro de la función ***"parametros"***. En la función ***"train"*** iteramos sobre la invocación de la función ***"parametros"***, minimizando el error J().

Los que den lugar a ese valor óptimo del coste, serán los usados para predecir los ejemplos seleccionados para el test.

```
In [9]: def pesosAleatorios(L_in, L_out):
            epsilon = 6**(0.5) / (L_in + L_out)
            return np.random.rand(L_in, L_out + 1) * 2 * epsilon - epsilon
```

```
In [10]: def parametros(params, input_size, hidden_size, num_labels, X, y, reg):
             # scipy.optimize.minimize: para la minimización de una función escalar de una o m
             result = opt.minimize(fun=backprop, x0=params,
                                   args=(input_size, hidden_size,
```

```
                                num_labels, X, y, reg),
                                method='TNC', jac=True,
                                options={'maxiter':70})
            return result.x
```

Funciones auxiliares para el backprop de la red neuronal

```
In [11]: def saveMatrix(file, X):
             return np.savetxt(file, X)

In [12]: def sigmoide(z):
             return 1 / (1 + np.exp(-1*z))

In [13]: def derivadaSigmoide(z):
             return sigmoide(z)*(1-sigmoide(z))
```

Retropropagación para calcular el coste y gradiente en una red neuronal En la función ***backprop*** dados unos parámetros de entrada , y unos datos de entrada de la red neuronal X, los procesamos obteniendo la salida de la red neuronal y almacenándola en z (predicción de salidas). A continuación con esos datos de salida estimados, se hace unna retropropagación sobre la red neuronal para obetener el error y el vector gradiente.

```
In [14]: 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, (nu
             theta2 = np.reshape(params_rn[num_ocultas*(num_entradas + 1):], (num_etiquetas, (
             m = len(X)
            #Input
             ones_columns_input = np.array(np.ones(m))
             a1 = np.insert(X, 0,ones_columns_input, axis = 1)

             #hidden_layer
             z2 = np.dot(theta1, a1.transpose())
             a2 = sigmoide(z2)
             one_columns_hidden = np.array(np.ones(m))
             a2 = np.insert(a2, 0, one_columns_hidden, axis = 0)

             #Output_layer
             z3 = np.dot(theta2, a2)
             h = sigmoide(z3)
             y_converted = (np.ravel(y) + 1)/2

              #Cost
             regulation = (reg/(2*m)) * (np.sum(theta1**2) + np.sum(theta2**2))
             J = np.sum(-y_converted * np.log(h) - (1 - y_converted)*np.log(1 - h)) * (1/m)
             J_regulated = J + regulation

             # Retro-Propagation
```

```
            d3 = h - y_converted
            z2 = np.insert(z2, 0, np.ones(m), axis = 0)
            z2prima = derivadaSigmoide(z2)
            d2 = (np.dot(theta2.transpose(), d3))*z2prima

            #Gradient
            delta2 = np.dot(d3,a2.transpose())
            delta1 = np.dot(d2, a1)

            #Regularization

            D1 = (delta1[1:,:]/m + theta1*reg/m).ravel()
            D2 = (delta2/m + theta2*reg/m).ravel()
            gradient = np.r_[D1, D2]

            return J_regulated, gradient
```

2.0.2 Support Vector Machines

Aplicaremos esta otra técnica para definir el modelo de entrenamiento y comparar los resultados obtenidos de esta forma, con los resultados de la red neuronal.

```
In [15]: def gaussian_kernel(X, y, C, sigma):
             #fit the SVM model
             svm = SVC(kernel="rbf",C=C, gamma =1 /float(2 * sigma**2))
             svm.fit(X,np.ravel(y))

             y = svm.predict(X=X)
             return y
```

3 FUNCIONES PARA LA SELECCIÓN DE PARÁMETROS

En el procesamiento de datos con una red neuronal hay diversos **parámetros** cuya selección puede afectar a la predicción: - El factor de regresión ***reg*** - El número de **iteraciones** en el **cálculo de parámetros** óptimos. - El número de **atributos de los ejemplos** de entrenamiento(inicialmente 10000) - El número de **nodos de la capa oculta** de la red neuronal - El número de **ejemplos de entrenamiento** seleccionados en cada iteración de la regresión en el cálculo de los .

En la función ***coste*** se incluye el cálculo exclusivo del coste en una red neuronal, aplicado también en la función ***"evaluacion_red"***.

```
In [16]: def coste(params, X, num_entradas, num_ocultas, num_etiquetas):
             theta1 = np.reshape(params[:num_ocultas*(num_entradas + 1)], (num_ocultas, (num_e
             theta2 = np.reshape(params[num_ocultas*(num_entradas + 1):], (num_etiquetas, (num_
             m = len(X)
             #Input
             ones_columns_input = np.array(np.ones(m))
             a1 = np.insert(X, 0,ones_columns_input, axis = 1)
             #hidden_layer
```

```
    z2 = np.dot(theta1, a1.transpose())
    a2 = sigmoide(z2)
    one_columns_hidden = np.array(np.ones(m))
    a2 = np.insert(a2, 0, one_columns_hidden, axis = 0)
    #Output_layer
    z3 = np.dot(theta2, a2)
    return sigmoide(z3)
```

La función ***"evaluación_red"*** calcula el porcentaje de acierto de la clasificación de los datos comparando las salidas de la red neuronal ***h***(transformándola en números enteros ***0*** o ***1***, almacenados en la variable ***z***) con los "labels" (salidas ***y***) . En el caso de usar **svm**, aplicamos la función **evaluacion_svm**. Las ***y*** o labels leídos de los data set toman valores 1 y -1, por ello le sumamos 1 y dividimos entre 2 para que tomen valores 0 y 1 , y podamos compararlos con las z calculadas.

Así ejecutaremos nuestra función ***main*** con diferentes valores de los parámetros que se pueden seleccionar y compararemos los porcentajes de acierto, para determinar cómo de bueno es nuestro modelo y si estamos en algún caso de riesgo de *overfiting* o *underfiting*

```
In [17]: def evaluacion_red(params, X, Y, num_entradas, num_ocultas, num_etiquetas):
             h = coste(params,X,num_entradas,num_ocultas,num_etiquetas)
             z = (np.ravel(h) >= 0.5)
             Y = (np.ravel(Y) + 1)/2
             z = list(map((lambda x,y: x == y), z, Y))
             return np.ravel(sum(z)/float(len(z))*100)[0]

In [18]: def evaluacion_svm(outputs, Y):
             Y = (np.ravel(Y) + 1)/2
             outputs = (np.ravel(outputs) + 1)/2
             z =list(map((lambda outputs,y: outputs == y), outputs, Y))
             return np.ravel(sum(z)/float(len(outputs))*100)[0]
```

Para el diagnóstico de nuestro sistema de aprendizaje automático, definimos la función ***curvasAprendizaje*** que nos permite ver como se distribuyen el coste de predicción para los ejemplos de entrenamiento y para los ejemplos e validación

```
In [19]: def curvasAprendizaje(errorT, errorVal, reg, num_ocultas, iterations):
             plt.figure()
             plt.xlabel('Numero de ejemplos de entrenamiento')
             plt.ylabel('Error')
             plt.title("$lambda$ = " + reg + ", nodos ocultos = " + num_ocultas)
             plt.plot(np.array(range(len(errorT)))*iterations, errorT, '-', color='blue', label
             plt.plot(np.array(range(len(errorVal)))*iterations, errorVal, '-', color='orange'
             plt.legend()
             plt.savefig('curva_aprendizaje_' + reg + '_' + num_ocultas + '_' + str(iterations)
             plt.show()
```

3.0.1 Funciones Main(test)

Función de entrenamiento para la red Neuronal

```
In [20]: def train(X, y, cvX, cvY, pesos, num_entradas, num_ocultas, num_etiquetas, iterations
             Jts = []
             Jcvs = []
             for i in range(1, int(len(X)/iterations)):
                 pesos = parametros(pesos, num_entradas, num_ocultas, num_etiquetas, X[0:i*ite
                 Jts += [backprop(pesos, num_entradas, num_ocultas, num_etiquetas, X, y, reg)[
                 Jcvs += [backprop(pesos, num_entradas, num_ocultas, num_etiquetas, cvX, cvY,
             curvasAprendizaje(Jts, Jcvs, str(reg), str(num_ocultas), iterations)
             return pesos
```

Funciones de prueba de los modelos de Red neuronal y svm

```
In [26]: def test_svm(C_in, sigma_in):
              #Training data
             X, y = lectura_Train_Set()

             #Validation data
             cvX, cvY = lectura_Valid_Set()
             #Testing Data
             x_test, y_test  = lectura_Test_Set()

             y = y.astype(int)

             #parámetros
             C = C_in
             sigma = sigma_in
             outputs = gaussian_kernel(X,y,C,sigma)

             #print('Prediccion_svm con Training Data: ', outputs)
             print('Evaluación para Training Data: ')
             print(evaluacion_svm(outputs, y))

             outputs_valid = gaussian_kernel(cvX, cvY, C, sigma)

             #print('Prediccion_svm con Validation Data: ', outputs_valid)
             print('Evaluación para Validation Data: ')
             print(evaluacion_svm(outputs_valid, y))

             outputs_test = gaussian_kernel(x_test, y_test, C, sigma)

             #print("Prediccion_svm con Testing Data:", outputs_test)
             print('Evaluacion para Testing Data:')
             print(evaluacion_svm(outputs_test, y))
```

```
In [22]: def test_red(num_ocultas, reg, iterations):
             #Training data
             X, y = lectura_Train_Set()
```

```
#Validation data
cvX, cvY = lectura_Valid_Set()
#Testing Data
x_test, y_test = lectura_Test_Set()

#PCA - descomentar para usar
#pca = decomposition.PCA(n_components=50)
#pca.fit(X)
#X = pca.transform(X)

#pca.fit(cvX)
#cvX = pca.transform(cvX)

#pca.fit(x_test)
#x_test = pca.transform(x_test)

y = y.astype(int)
num_entradas = X.shape[1]
num_etiquetas = 1

theta1 = pesosAleatorios(num_ocultas, X.shape[1])
theta2 = pesosAleatorios(num_etiquetas, num_ocultas)

pesos = np.concatenate((np.ravel(theta1), np.ravel(theta2)))
#Entrenamiento de la red neuronal
pesos = train(X, y, cvX, cvY, pesos, num_entradas, num_ocultas, num_etiquetas, ite


print('Evaluación para Training.data')
print (evaluacion_red(pesos, X, y, num_entradas, num_ocultas, num_etiquetas))
print('Evaluación para Validation.data')
print (evaluacion_red(pesos, cvX, cvY, num_entradas, num_ocultas, num_etiquetas))
print('Evaluacion para los ejemplos de test(50 ultimos de Validation.data):')
print((evaluacion_red(pesos, x_test, y_test,num_entradas, num_ocultas, num_etiquet
```

4 PRUEBAS CON DIFERENTES CONFIGURACIONES DE LOS MODELOS

Parámetros para la red Neuronal: - reg = factor de regresión

- iterations = número de ejemplos de entrenamiento que añadimos a cada iteración del back-prop(lotes) (p.e si iterations = 2, en la primera vuelta del bucle del entrenamiento(función ***train***) usaremos 2 ejemplos de entrenamiento para el cálculo de , en la segunda usaremos 4... hasta completar la última con todos los ejemplos de entrenamiento).
- num_ocultos = numero de nodos de la capa oculta

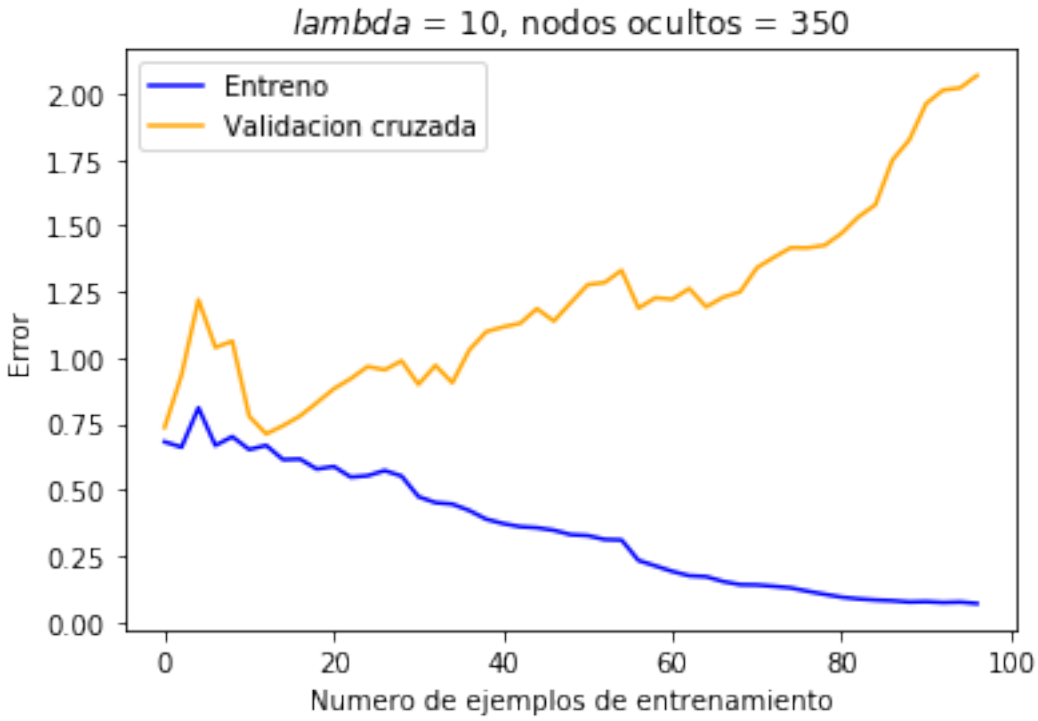
A continuación se muestran las diferentes pruebas realizadas para evaluar nuestos modelos de aprendizaje automático:

Ejecución aplicando PCA a los atributos de los ejemplos:

Con PCA

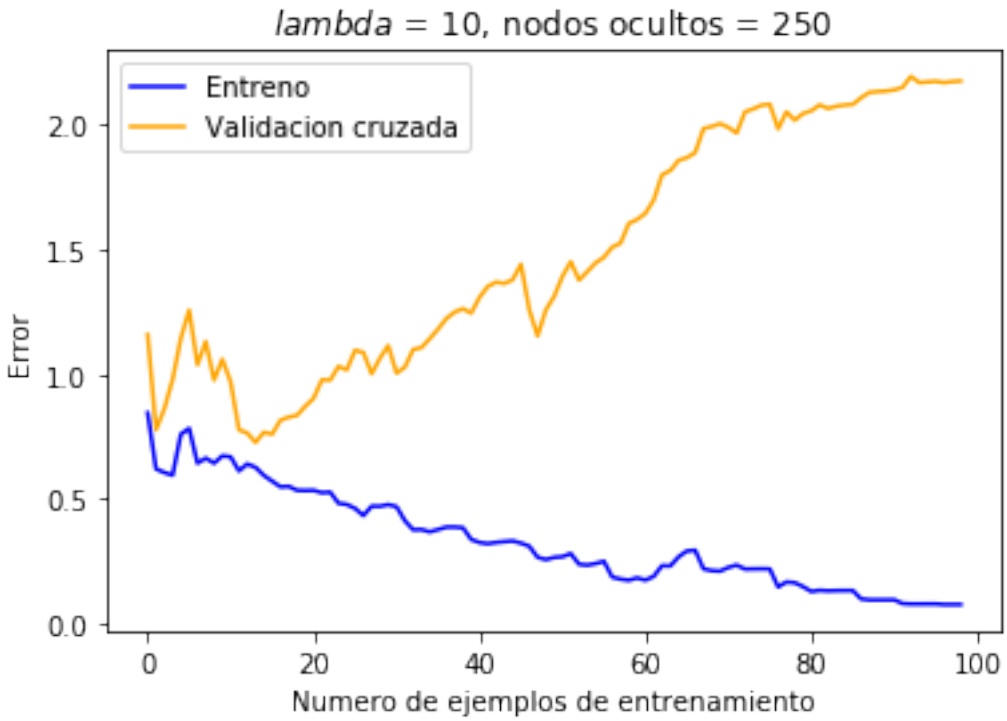
Probamos a utilizar PCA para aumentar la rapidez de ejecución, PCA = 50 ya que solo tenemos 50 ejemplos de validacion y es el número mínimo que seleccionará PCA para trabajar. Pues dada una Matriz (50,10000), PCA reduce los atributos(columnas) a la mínima dimensión; es decir 50.

```
In [32]: num_ocultas = 350
         reg = 10
         iterations = 2
         test_red(num_ocultas, reg, iterations)
```




```
Evaluación para Training.data
99.0
Evaluación para Validation.data
54.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
62.0
```

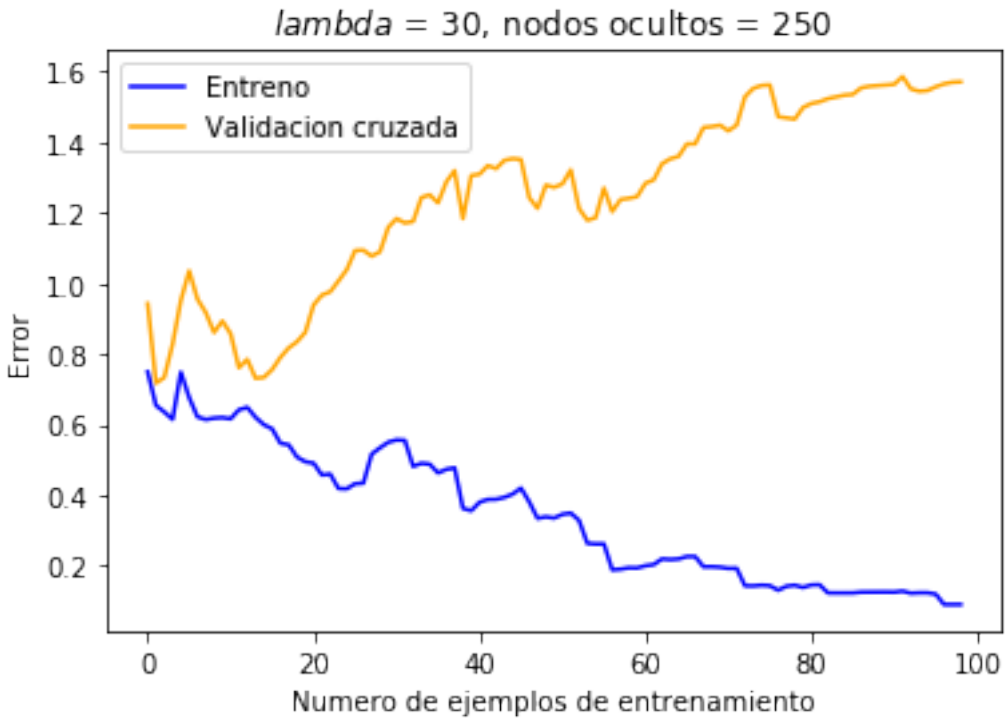
```
In [34]: num_ocultas = 250
         reg = 10
         iterations = 1
         test_red(num_ocultas, reg, iterations)
```




```
Evaluación para Training.data
99.0
Evaluación para Validation.data
50.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
60.0
```

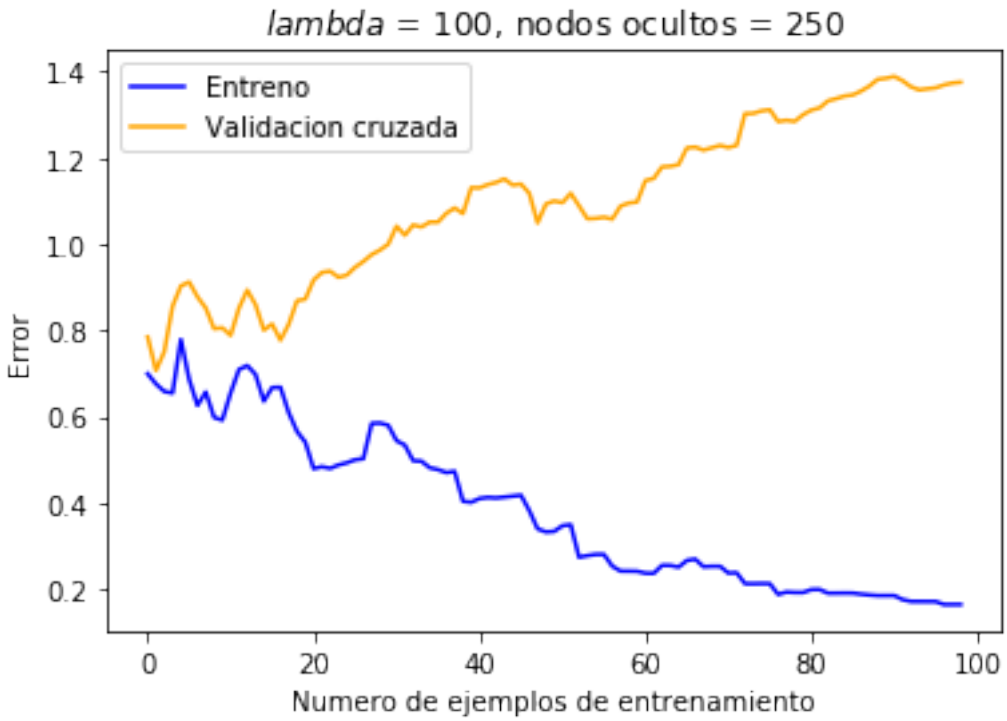
Observamos que hay un gran problema de varianza, así decidimos aumentar el factor de regresión para comprobar si se soluciona

```
In [37]: num_ocultas = 250
         reg = 30
         iterations = 1
         test_red(num_ocultas, reg, iterations)
```



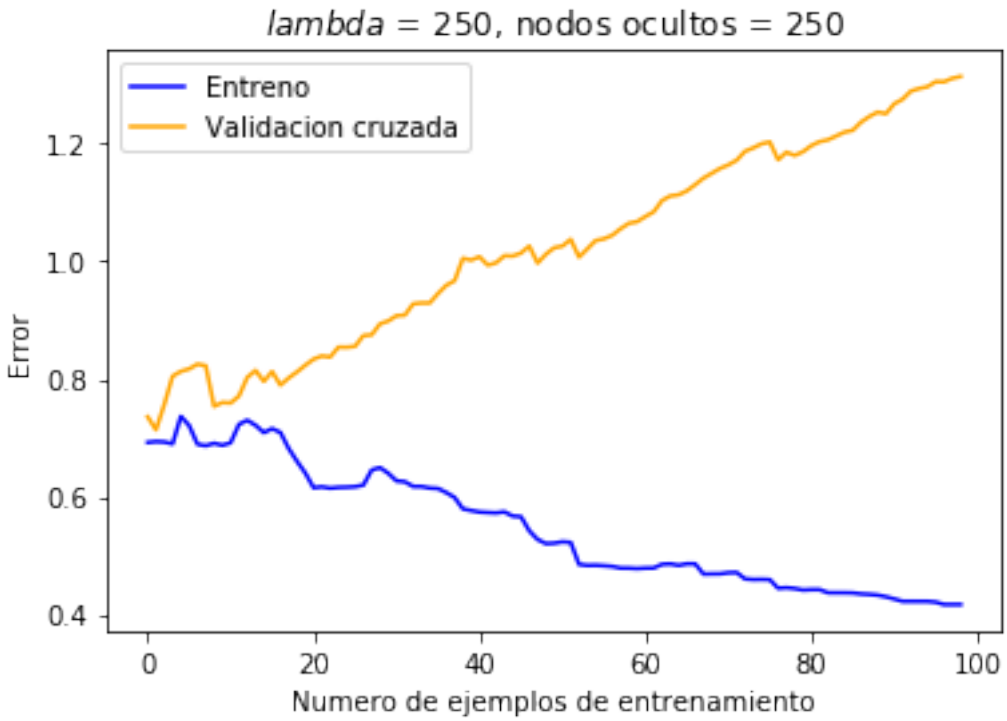

```
Evaluación para Training.data
99.0
Evaluación para Validation.data
56.00000000000001
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
64.0
```

```
In [38]: num_ocultas = 250
         reg = 100
         iterations = 1
         test_red(num_ocultas, reg, iterations)
```



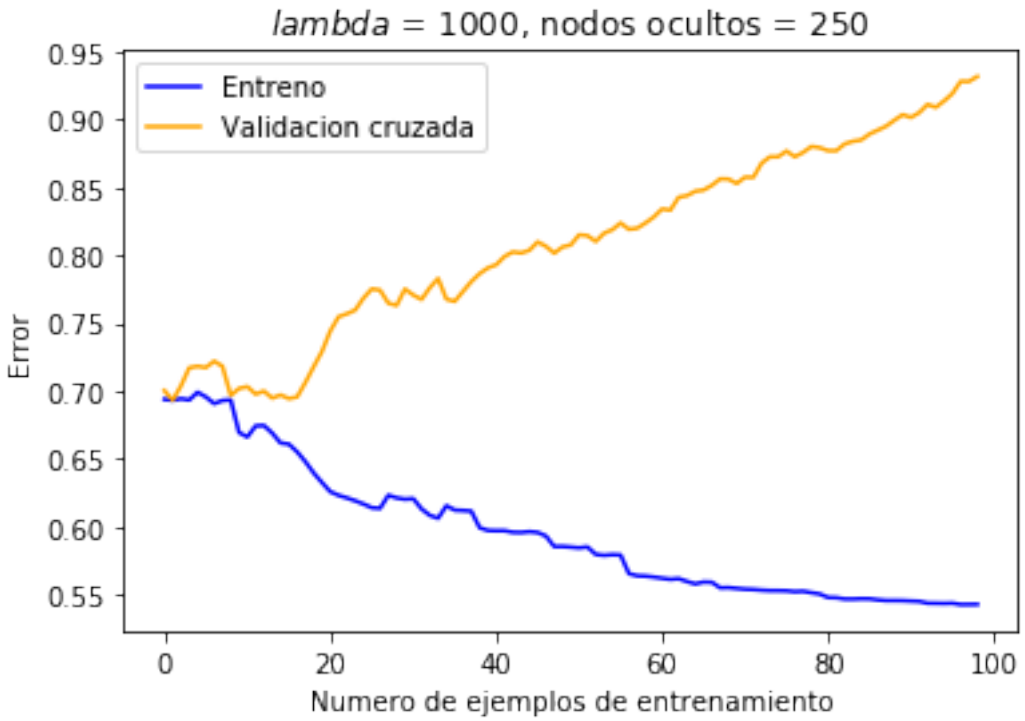

```
Evaluación para Training.data
99.0
Evaluación para Validation.data
60.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
54.0
```

```
In [42]: num_ocultas = 250
         reg = 250
         iterations = 1
         test_red(num_ocultas, reg, iterations)
```




```
Evaluación para Training.data
100.0
Evaluación para Validation.data
52.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
54.0
```

```
In [40]: num_ocultas = 250
         reg = 1000
         iterations = 1
         test_red(num_ocultas, reg, iterations)
```




```
Evaluación para Training.data
59.0
Evaluación para Validation.data
60.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
54.0
```

Si vamos aumentando el factor de regresión,vemos que la curva de error de validación va disminuyendo y por tanto el porcentaje de validación aumenta proporcionalmente. Sin embargo, el porcentaje de acierto del test, disminuye.

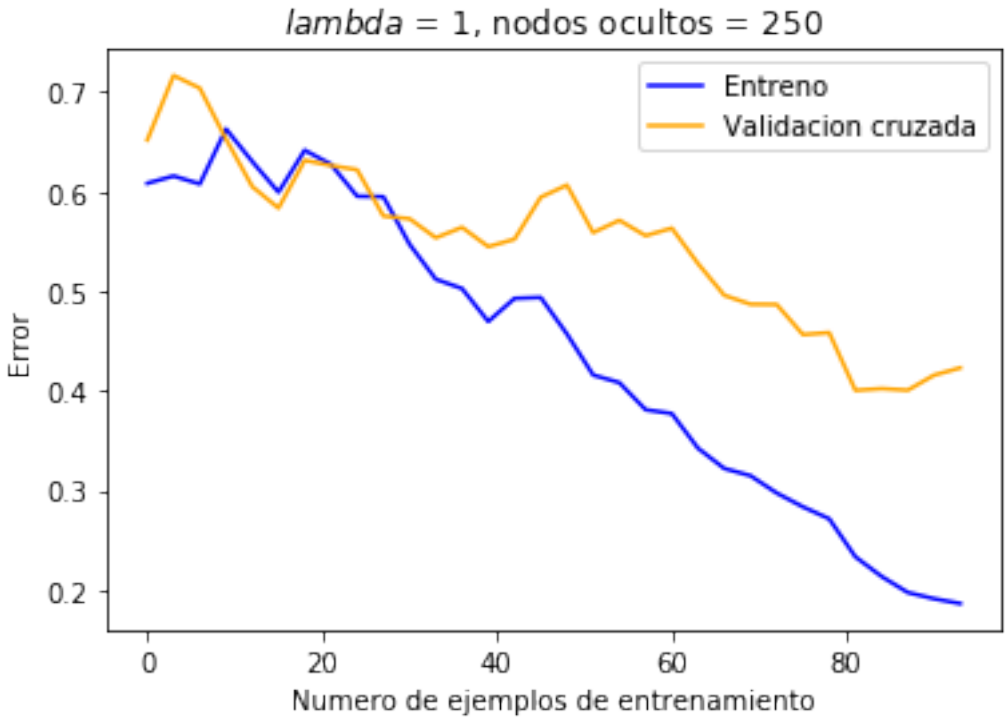
En el caso extremo de asignar un factor de regresión 1000, el porcentaje de acierto de la fase de validación deja de aumentar y el entrenamiento resulta pésimo pues hay un underfit dando lugar a un porcentaje de acierto de la fase de entrenamiento de 59%.

Analizando la funcionalidad del PCA, la única ventaja es aumentar la rapidez del procesamiento de entrenamiento pues al tener 10000 atributos y transformarlo a 50, la CPU tiene que procesar cálculos menores con matrices más pequeñas. Creemos que no conseguimos porcentajes altos debido a estar limitados a 50 dimensiones por tener solo 50 ejemplos de validación y 50 ejemplos de test, por ello esta técnica no nos vale. Procedemos a hacer pruebas sin PCA.

Sin PCA:

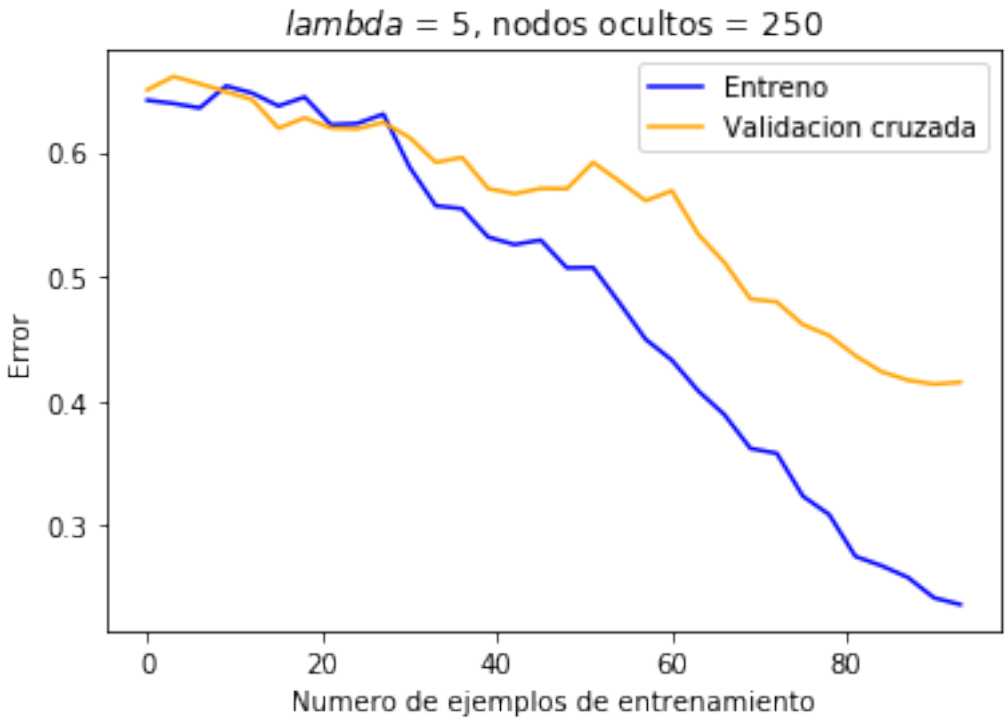
Procedemos a ejecutar el ***test_red*** con las mismas configuraciones anteriores de parámetros, pero ahora añadiendo PCA en la función para reducir el número de atributos de los ejemplos de datos

```
In [46]: num_ocultas = 250
         reg = 1
         iterations = 3
         test_red(num_ocultas, reg, iterations)
```



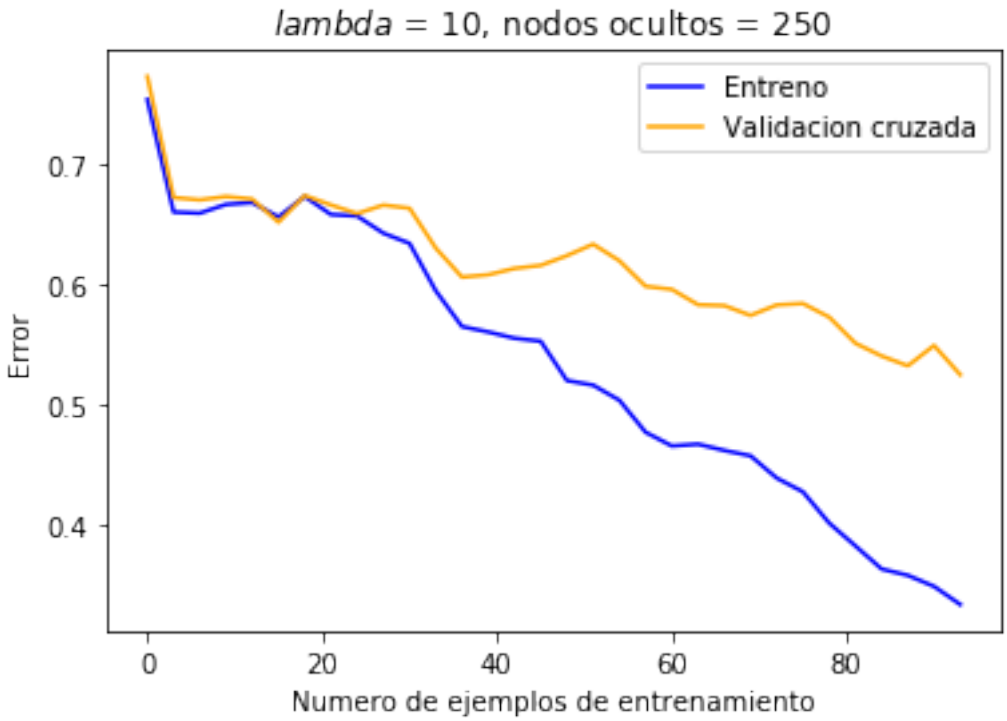

```
Evaluación para Training.data
98.0
Evaluación para Validation.data
82.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
86.0
```

```
In [47]: num_ocultas = 250
         reg = 5
         iterations = 3
         test_red(num_ocultas, reg, iterations)
```



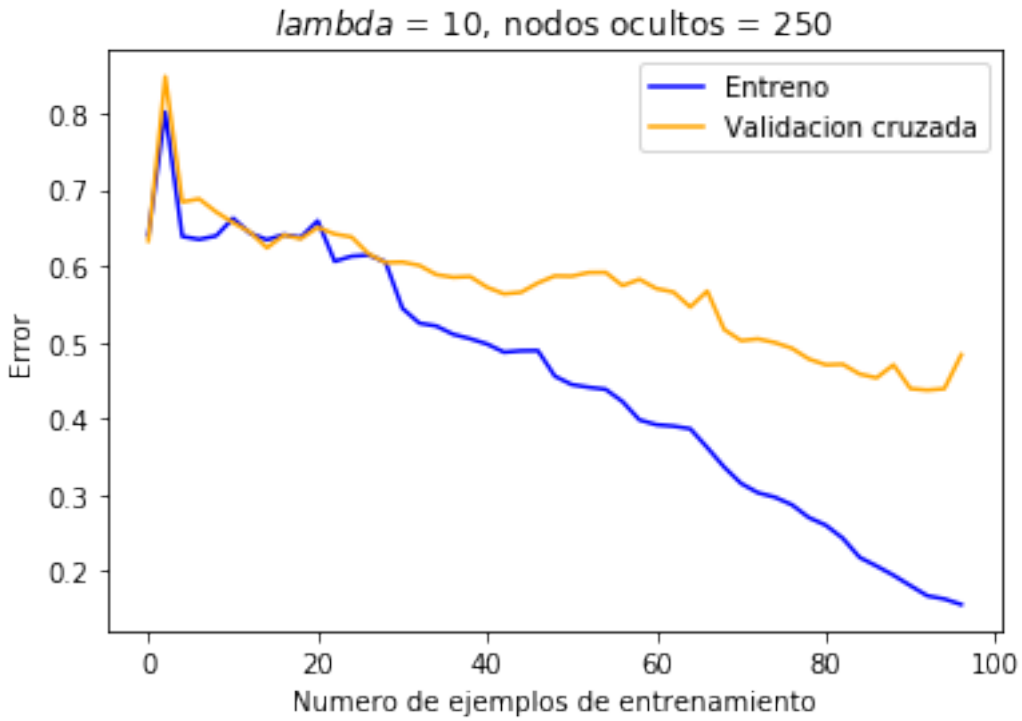

```
Evaluación para Training.data
92.0
Evaluación para Validation.data
86.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
88.0
```

```
In [44]: num_ocultas = 250
         reg = 10
         iterations = 3
         test_red(num_ocultas, reg, iterations)
```



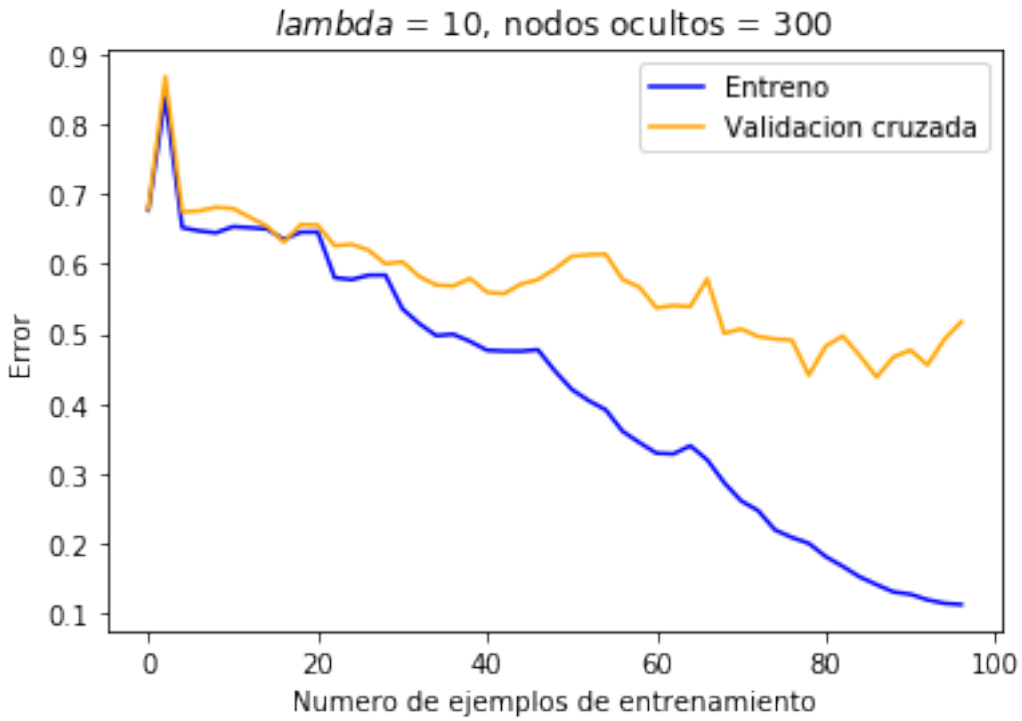

```
Evaluación para Training.data
90.0
Evaluación para Validation.data
76.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
84.0
```

```
In [45]: num_ocultas = 250
         reg = 10
         iterations = 2
         test_red(num_ocultas, reg, iterations)
```



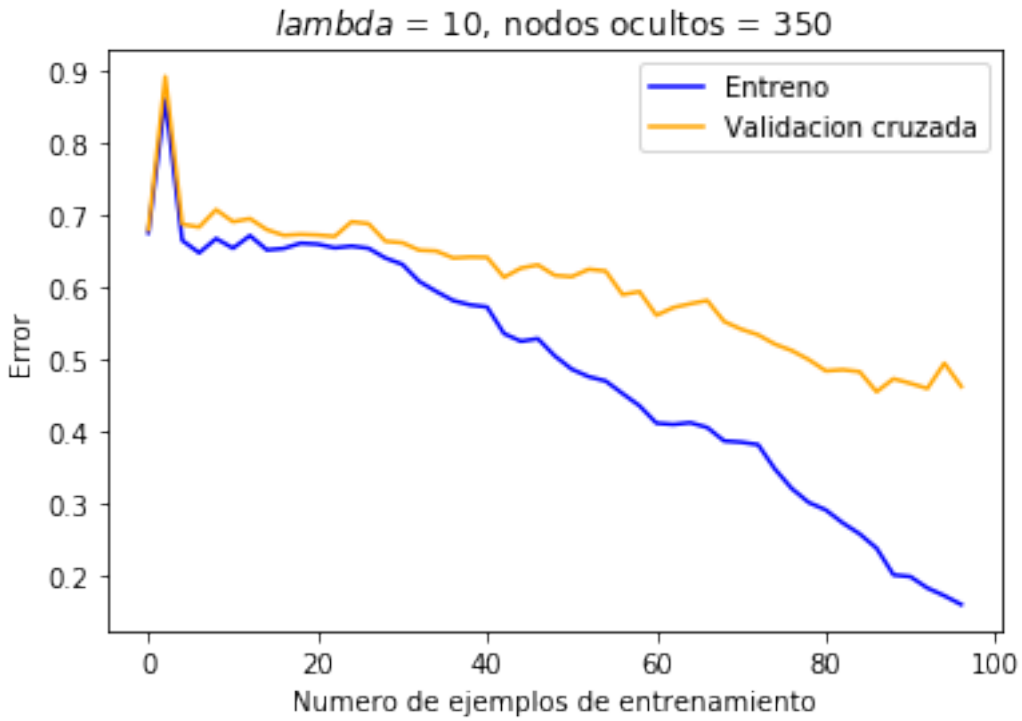

```
Evaluación para Training.data
99.0
Evaluación para Validation.data
84.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
84.0
```

```
In [23]: num_ocultas = 300
         reg = 10
         iterations = 2
         test_red(num_ocultas, reg, iterations)
```



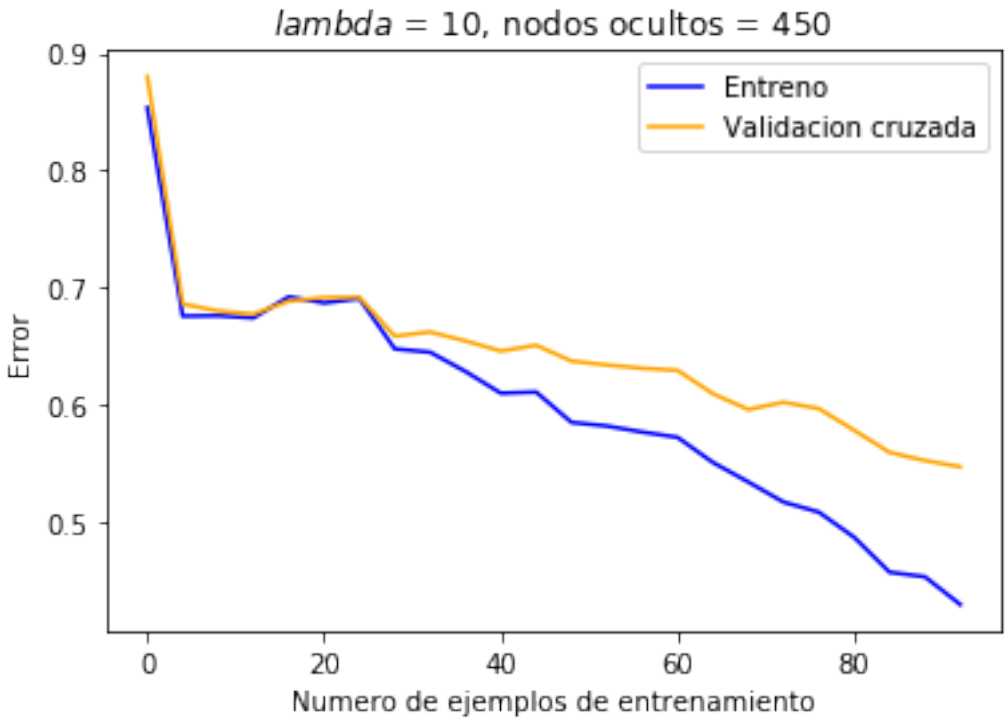

```
Evaluación para Training.data
99.0
Evaluación para Validation.data
84.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
82.0
```

```
In [23]: num_ocultas = 350
         reg = 10
         iterations = 2
         test_red(num_ocultas, reg, iterations)
```



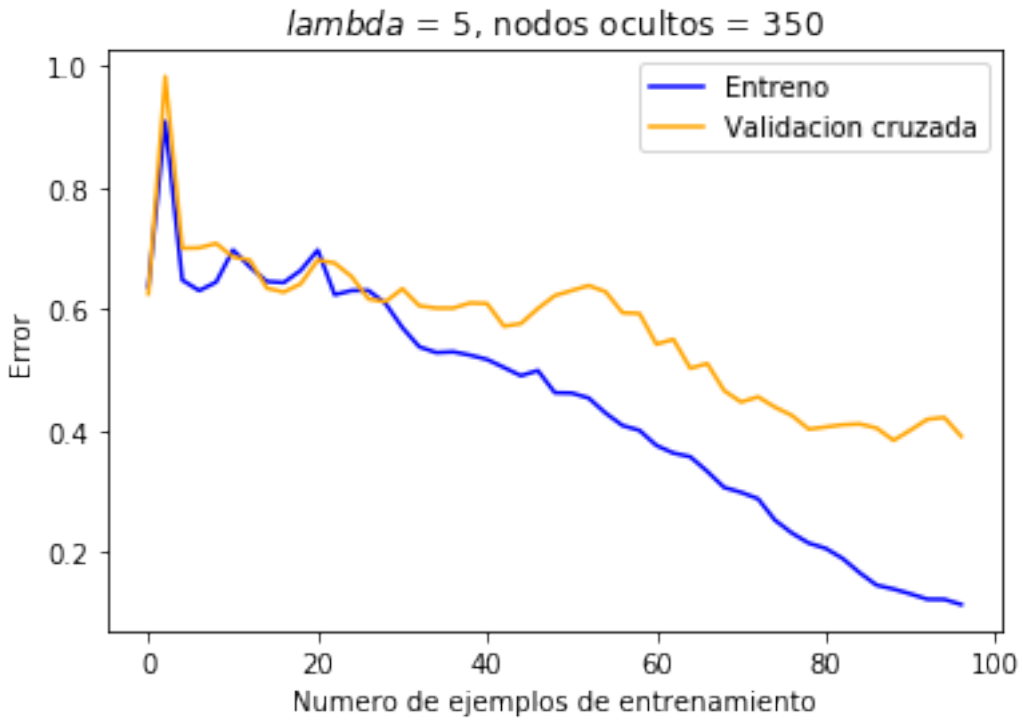

```
Evaluación para Training.data
99.0
Evaluación para Validation.data
86.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
88.0
```

```
In [24]: num_ocultas = 450
         reg = 10
         iterations = 4
         test_red(num_ocultas, reg, iterations)
```




```
Evaluación para Training.data
87.0
Evaluación para Validation.data
76.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
84.0
```

```
In [31]: num_ocultas = 350
         reg = 5
         iterations = 2
         test_red(num_ocultas, reg, iterations)
```




```
Evaluación para Training.data
99.0
Evaluación para Validation.data
86.0
Evaluacion para los ejemplos de test(50 ultimos de Validation.data):
88.0
```

Pruebas con Support Vector Machines:

```
In [27]: C = 1
         sigma = 0.1
         test_svm(C, sigma)
```

```
Evaluación para Training Data:
100.0
Evaluación para Validation Data:
40.0
Evaluacion para Testing Data:
44.0
```

```
In [28]: C = 2
         sigma = 0.1
         test_svm(C, sigma)
```

```
Evaluación para Training Data:
100.0
Evaluación para Validation Data:
40.0
Evaluacion para Testing Data:
44.0
```

```
In [29]: C = 10
         sigma = 1
         test_svm(C, sigma)
```

```
Evaluación para Training Data:
100.0
Evaluación para Validation Data:
40.0
Evaluacion para Testing Data:
44.0
```

5 CONCLUSIÓN

Por un lado, la técnica válida para entrenar nuestro sistema de clasificación es redes neuronales. SVM no da un resultado coherente ni representativo porque no es válido para un dataset con pocos ejemplos de entrenamiento frente al desmesurado número de atributos, pues a pesar de cambiar los parámetros C y sigma, el porcentaje de aciertos sobre los ejemplos de entrenamiento es 100% (overfit) frente a los bajos porcentajes de validación y test de 40% y 44 % respectivamente.Así descartamos los resultados obtenidos con ese modelo.

En las primeras pruebas ejecutadas con la red Neuronal, hemos aplicado la técnica PCA pero hemos obtenido un porcentaje de acierto de training muy alto frente a unos porcentajes de validación y test muy bajos, es decir hay gran varianza. Esta técnica no beneficia en la mejora de nuestro modelo clasificador.

Conclusión 1: No usar PCA

En las pruebas siguientes ejecutadas con la red neuronal sin PCA, para un mismo número de nodos ocultos y de lotes de ejemplos de entrenamiento(iterations), si aumentamos el factor de regresión entre 1 y 10, los porcentajes mejoran pero a partir de valor cercano a 5. Con reg = 10, los porcentajes de evaluación empeoran si el numero de nodos ocultos se aleja del óptimo, porque se produce underfitting o un overfitting. Sin embargo, el desajuste producido por reg = 10 se puede compensar como veremos a continuación con un número de nodos mejor.Concluimos en que para reg = 5 el valor es óptimo en todos los casos probados.

Conclusión 2: factor de regresión óptimo seleccionado = 5

Por otro lado, el numero de ejemplos de entrenamiento por lote(iterations) es proporcional al número de iteraciones del cálculo de los pesos en el aprendizaje. Si dejamos un mismo factor de regresión y de nodos ocultos, a menor número de ejemplos de entrenamiento por lote(3 y 2) el entrenamiento se ajusta más a los datos, luego mejora los porcentajes de validación y test de entrada; aunque con valor 1 ya tendríamos problema de overfitting.

Conclusión 3: valor para iterations óptimo seleccionado = 2

reg	Iterations	Num_ocultas	Training % - Red Neuronal	CrossValidation % - Red Neuronal	Test % - Red Neuronal
1	3	250	98	82	86
5	3	250	92	86	88
10	3	250	90	76	84
10	2	250	99	84	84
10	2	300	99	84	82
10	2	350	99	86	88
10	4	450	87	76	84
5	2	350	99	86	88

Configuraciones y evaluación

Los mejores resultados los obtenemos para un numero de nodos ocultos entre 250 - 350. Si la capa oculta de la red neuronal tiene mas de 400 nodos, el entrenamiento de nuevo se ajusta demasiado a los ejemplos de train, resultando en un menor porcentaje de acierto de validación y test. Si por el contrario utilizamos un numero de nodos ocultos menor de 250, se produce underfitting.

Conclusión 4: valor para número de nodos ocultos = 350

En la representación de curvas de aprendizaje, vemos que sigue existiendo algo de varianza(separación entre curvas de entrenamiento y validación) aunque los resultados sean finalmente bastante precisos. Una posible mejora adicional para evitar un problema de varianza es:

- Reducir numero de features o características -> Sería eficiente poder eliminar los 3000 atributos añadidos aleatoriamente por los creadores del DataSet, pero sería un trabajo muy costoso a nivel CPU y tiempo que escapa de nuestro ámbito.

Webgrafía Webgrafía del proyecto útil para profundizar en el entendimiento del DataSet y consultar métodos de aprendizaje automáticos ya definidos en otros lenguajes de programacion para procesar los datos del mismo:

Repositorio con el enunciado y los datos: https://archive.ics.uci.edu/ml/datasets/Arcene

Fuente de datos originales: http://clinicalproteomics.steem.com/downloadovar.php.

Autores:

- MONTSERRAT SACIE ALCÁZAR
- TOMÁS GOLOMB DURÁN