1. Diagnóstico de cáncer de mamas

1.1. Código fuente

Carga de datos

```
import numpy as np

from preprocessor import OutlierFilter, FeatureNormalizer


class Ej1DataLoader:
    def LoadData(self):
        raw_data = np.genfromtxt('./ds/tp1_ej1_training.csv', delimiter=",")
        np.random.shuffle(raw_data)
        #features = raw_data[:, 1:]
        #labels = raw_data[:, 0]
        #transformed_features=OutlierFilter().process(features)
        #transformed_features=FeatureNormalizer().process(transformed_features)
        #return [transformed_features,labels]

        #Primero filtramos labels con features, luego separamos
        transformed_data=OutlierFilter().process(raw_data)

        #transformed_data=FeatureNormalizer().process(transformed_data)
        features = transformed_data[:, 1:]
        transformed_features = FeatureNormalizer().process(features)

        labels = transformed_data[:, 0]
        labels = labels.reshape((labels.shape[0],1))
        #print("features ", features.shape)
        #print("labels shape ",labels.shape)
        return [transformed_features,labels]
```

Preprocesamiento de datos

```
import numpy as np
import matplotlib.pyplot as plt


class FeatureNormalizer:
    def process(self, features):
        """
        :param features: datos a normalizar
        :return: devuelve datos normalizados por columna
        """
        feature_means = np.mean(features, axis=0)
        features_std = np.std(features, axis=0)
        features_normalized = (features - feature_means) / features_std
        return features_normalized


class OutlierFilter:
    def process(self, features):
        """
        :param features: datos
        :return: datos sin outliers
        """
        features_std = np.std(features, axis=0)
        feature_means = np.mean(features, axis=0)

        features_wo_outliers = np.array(filter(self.isOutlier(feature_means,
            features_std), features))

        # print(len(features), len(features_wo_outliers))

        # for i in range(len(features[0])):
        ##boxplot
        # fig = plt.figure(1, figsize=(9, 6))
        # ax = fig.add_subplot(111)
```

```
        # bp = ax.boxplot([features[:,i],features_wo_outliers[:,i]])
        # plt.show()

        ##histograma de la 3era feature con cortes de outliers
        # plt.hist(features[:,i],bins=np.max(features[:,i])-np.min(features[:,i]))
        # plt.axvline(feature_means[i]-2*features_std[i], color='b',
        #       linestyle='dashed', linewidth=2)
        # plt.axvline(feature_means[i]+2*features_std[i], color='b',
        #       linestyle='dashed', linewidth=2)
        # plt.show()

        # print("features_wo_outliers:", features_wo_outliers.shape)

        return features_wo_outliers

    def isOutlier(self, means, std):
        return lambda item: (np.abs(means - item) < 2 * std).all()
```

Modelo

```
import numpy as np
class LayerModel:

    # numInputUnits: Cantidad de unidades de entrada
    # hiddenLayers: Array con la cantidad de hidden layers de la red
    # numOutputLayers: Cantidad de unidades de salida
    def __init__(self, layerSizes, activationFn, activationDerivativeFunction):
        self._activationFn = activationFn
        self._activationDerivativeFunction = activationDerivativeFunction

        self._biases = [np.random.randn(y, 1) / 1000 for y in layerSizes[1:]]
        self._weights = [np.random.randn(y, x) / 1000
                         for x, y in zip(layerSizes[:-1], layerSizes[1:])]

        self._layer_sizes = layerSizes
        self._num_layers = len(layerSizes)

    def getInitializedWeightMats(self):
        return self._weights

    def getInitializedBiasVectors(self):
        return self._biases

    def getZeroDeltaW(self):
        return [np.zeros(w.shape) for w in self._weights]

    def getZeroDeltaB(self):
        return [np.zeros(b.shape) for b in self._biases]

    def getActivationFn(self):
        return self._activationFn

    def getActivationDerivativeFn(self):
        return self._activationDerivativeFunction

    def getNumLayers(self):
        return self._num_layers
```

Funciones sigmoideas

```
import numpy as np


def sigmoid_array(b, x):
    return 1.0 / (1.0 + np.exp(-b * x))


def sigmoid_gradient_array(b, x):
    return b * sigmoid_array(b, x) * (1.0 - sigmoid_array(b, x))


def sigmoid_tanh_array(b, x):
```

```
    return np.tanh(b * x)


def sigmoid_tanh_gradient_array(b, x):
    return b * (1 - (np.power(sigmoid_tanh_array(b, x), 2)))
```

Algoritmos

```
import numpy as np
from layer_model import LayerModel
import sigmoid


class NetworkSolver:
    def __init__(self, layer_model):
        self._weights = layer_model.getInitializedWeightMats()
        self._biases = layer_model.getInitializedBiasVectors()
        self._layer_model = layer_model

    def do_activation(self, sample):
        aa = [np.reshape(sample, (len(sample), 1))]
        zz = []
        # Bias
        for b, w in zip(self._biases, self._weights):
            z = np.dot(w, aa[-1]) + b
            a = self._layer_model.getActivationFn()(z)
            zz.append(z)
            aa.append(a)
        return (aa, zz)

    def do_backprop_and_return_grad(self, x, y):
        grad_w = self._layer_model.getZeroDeltaW()
        grad_b = self._layer_model.getZeroDeltaB()

        # feedforward
        activations, zs = self.do_activation(x)
        delta = (activations[-1] - y) * self._layer_model.getActivationDerivativeFn()(
            zs[-1])
        grad_b[-1] = delta
        grad_w[-1] = np.dot(delta, activations[-2].transpose())

        for l in xrange(2, self._layer_model.getNumLayers()):
            z = zs[-l]
            sp = self._layer_model.getActivationDerivativeFn()(z)
            delta = np.dot(self._weights[-l + 1].transpose(), delta) * sp
            grad_b[-l] = delta
            grad_w[-l] = np.dot(delta, activations[-l - 1].transpose())
        return (grad_b, grad_w)

    def correction_mini_batch(self, mini_batch, lr, n, lmbda=0.0):
        """
        :param mini_batch: set de entrenamiento
        :param lr: learning rate
        :param n: cantidad de samples
        :param lmbda: regularization parameter
        """
        grad_b = [np.zeros(b.shape) for b in self._biases]
        grad_w = [np.zeros(w.shape) for w in self._weights]
        for x, y in mini_batch:
            delta_grad_b, delta_grad_w = self.do_backprop_and_return_grad(x, y)
            grad_b = [gb + deltagb for gb, deltagb in zip(grad_b, delta_grad_b)]
            grad_w = [gw + deltagw for gw, deltagw in zip(grad_w, delta_grad_w)]

        self._weights = [(1.0 - lr * (lmbda / n)) * w - (lr / len(mini_batch)) * gw
                         for w, gw in zip(self._weights, grad_w)]
        # clasico sin regularizacion
        #     self._weights = [w - (lr / len(mini_batch)) * gw
        #                      for w, gw in zip(self._weights, grad_w)]
        self._biases = [b - (lr / len(mini_batch)) * gb
                        for b, gb in zip(self._biases, grad_b)]

    def learn_minibatch(self, mini_batches, mini_batches_testing, lr, epochs, epsilon,
```

```
                 lmbda=0.0):
        """
        :param mini_batches: set de entrenamiento
        :param mini_batches_testing: set de testing
        :param lr: learning rate
        :param epochs: cantidad de epocas
        :param epsilon: cota de error
        :param lmbda: parametro de regularizacion , si no se especifica , no se
    regulariza
        imprime errores de entrenamiento y testing por cada epoca
        """
        T = epochs
        t = 0
        e = 999
        n = sum([len(mbatch) for mbatch in mini_batches])
        while e > epsilon and t < T:
            for b in mini_batches:
                self.correction_mini_batch(b, lr, n, lmbda)
            t = t + 1
            e = self.get_prediction_error(mini_batches, False)
            et = self.get_prediction_error(mini_batches_testing, False)
            print ("Training Error: ", e, "Val_error:", et)

            # e = self.get_prediction_error(mini_batches, True)

    def get_prediction_error(self, mini_batches, bprint):
        e = 0
        cant = 0
        for b in mini_batches:

            for x, y in b:
                cant = cant + 1
                aa, zz = self.do_activation(x)

                e = e + np.linalg.norm(aa[-1] - y)
                if bprint:
                    print e / cant, np.linalg.norm(aa[-1] - y), aa[-1][0][0], y[0]

        return e / cant

    def get_hits(self, test_data):
        """
        :param test_data: set de datos de testing
        :return: Devuelve el numero de aciertos de inputs de test para los que
        los outputs que devuelve la red son correctos.
        """
        test_results = [(self.get_result(self.do_activation(x)[0][-1]), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def get_result(self, act):
        """
        :param act: resultado de nuestra red
        :return: el resultado es el mas cercano al resultado
        que devolvio la red entre 0 y 1
        """
        return np.argmin([abs(act - 0), abs(act - 1)])
```

Test

```
import sigmoid
import ej1_data_loader
from layer_model import LayerModel
from feed_forward_solver import NetworkSolver
import functools

loader = ej1_data_loader.Ej1DataLoader()
data = loader.LoadData()
features = data[0] #shape=(333,10)
labels = data[1]

all_data = zip(features, labels)
```

```
num_training_samples = int(len(all_data) * 0.75)
num_test_samples = len(all_data) - num_training_samples

training_data = all_data[0:num_training_samples - 1]
test_data = all_data[num_training_samples:len(all_data) -1]

mini_batch_size = 1
n = len(training_data)
beta = 5

mini_batches_training = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, num_training_samples - 1, mini_batch_size)]

mini_batches_testing = [
                test_data[k:k+mini_batch_size]
                for k in xrange(0, num_test_samples - 1, mini_batch_size)]

model =  LayerModel([10,12,1], functools.partial( sigmoid.sigmoid_array,beta),
    functools.partial( sigmoid.sigmoid_gradient_array,beta))
solver = NetworkSolver(layer_model=model)

lr = 0.005
epochs = 1000
epsilon = 0.05
reg_param = 0.0
solver.learn_minibatch(mini_batches_training,mini_batches_testing,lr,epochs,epsilon,
    reg_param)
```

Eficiencia energética