Project 2 on Machine Learning

Classification and Regression, from linear and logistic regression to neural networks

Jessica Alatorre Flores

Abstract

In this project we will study regression and classification problems, starting with the same algorithm we studied in project 1, then we will inlcude logistic regression for classification problem, and finally we will implement a multilayer perceptron model for both problems, regression and classification.

We will be working with the data from the Ising model and we will focus on supervised training.

Part a) Producing the data for the one-dimensional Ising model

"The Ising model, is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables that represent magnetic dipole moments of atomic spins that can be in one of two states (+1 or −1). The spins are arranged in a graph, usually a lattice, allowing each spin to interact with its neighbors. The model allows the identification of phase transitions, as a simplified model of reality" (https://en.wikipedia.org/wiki/Ising_model (https://en.wikipedia.org/wiki/Ising_model))

For the discussion here, we will use the one-dimensional Ising model that consists of a simple binary value system where the variables of the model (spins) can take two values only, for example (+1, -1) or (0, 1).

$$E = -J \sum_{\langle kj \rangle} S_i S_j$$

We start with the one-dimensional Ising model with nearest neighbor iteraction, this model has no phase transition at finite temperature.

The following Python code generates the training data, and defines the Ising model parameters. This codes is based in the one that was provided to us in the poject specifitations. (https://compphysics.github.io/MachineLearning/doc/Projects/2018/Project2/pdf/Project2.pdf (https://compphysics.github.io/MachineLearning/doc/Projects/2018/Project2/pdf/Project2.pdf))

In [1]:
```
import numpy as np
np.random.seed(12)

def ising_energies(states,L):
    """
    This function calculates the energies of the states in the nn Ising Hamilt
onian
    """
    J=np.zeros((L,L),)
    for i in range(L):
        J[i,(i+1)%L]-=1.0
    # compute energies
    E = np.einsum('...i,ij,...j->...',states,J,states)
    return E


""" Define Ising model """
#system size
L = 40
#create 1000 random Ising states
states = np.random.choice([-1, 1], size=(1000, L))

# calculate Ising energies
energies=ising_energies(states,L)  # Y or dependent var

# reshape Ising states into RL samples: S_iS_j --> X_p
states=np.einsum('...i,...j->...ij', states, states)
shape=states.shape
states=states.reshape((shape[0],shape[1]*shape[2])) # X or independent var
```

To apply Linear regression, we have to recast the moddel in the form:

$$H^i_{\mathrm{model}} \equiv \mathbf{X}^i \cdot \mathbf{J},$$

where the vectors X_i represent all two-body interactions $\{S^i_j S^i_k\}_{j,k=1}^L$, and the index i runs over the samples in the data set. To make the analogy complete, we can also represent the dot product by a single index $p = \{j,k\}$, i.e.$\mathbf{X}^i \cdot \mathbf{J} = X^i_p J_p$ Note that the regression model does not include the minus sign, so we expect tolearn negative $J's$.

Part b) Estimating the coupling constant of the one-dimensional Ising model using linear regression

In this part I used the same codes from project 1, but this time with a clearer structure and separating into functions.

In [2]:
```
#Least square regression
def ols_regression(data_train, data_test, depen):
    """ Function that performs the OLS regression with the inverse"""
    beta_ols = np.linalg.inv(data_train.T @ data_train) @ data_train.T @ depen
    pred= data_test@beta_ols
    return beta_ols, pred

import scipy.linalg as scl
def ols_SVD(x: np.ndarray, y: np.ndarray) -> np.ndarray:
    """ Function that performs the OLS regression with the Singular Value Desc
omposition"""
    u, s, v = scl.svd(x)
    return v.T @ scl.pinv(scl.diagsvd(s, u.shape[0], v.shape[0])) @ u.T @ y
```

In [3]:
```
def ridge_regression(data_train, data_test, depen, alpha):
    """Funtions that performs the Ridge Regression"""
    beta_olsRidge = np.linalg.inv(data_train.T @ data_train + alpha*np.identit
y(1600)) @ data_train.T @ depen
    pred= data_test@beta_olsRidge
    return beta_olsRidge, pred
```

In [4]:
```
"""The following functions calculate different measurements to assess the mode
l"""
def mse(y_pred, y_test):
    return np.mean( np.mean((y_test - y_pred)**2) )
def r2(y_pred, y_test):
    return np.sqrt(np.mean( np.mean((y_test - y_pred)**2) ))
def bias(y_pred, y_test):
    return np.mean( (y_test - np.mean(y_pred))**2 )
def variance(y_pred, y_test):
    return np.mean( np.var(y_pred))

from tabulate import tabulate

def all_values(y_pred, y_test):
    measuremnts = [['MSE', mse(y_pred,y_test)],
         ['R2', r2(y_pred,y_test)],
         ['Bias', bias(y_pred,y_test)],
         ['Variance', variance(y_pred,y_test)]]
    return measuremnts
```

After define the function we need, the data was splitted in train and test using sklear.

In [5]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(states, energies, test_siz
e=0.2)
```

Performing the linear regression with single value descomposition:

In [6]:
```
coefs_OLS = ols_SVD(x_train, y_train)
y_pred= x_test@coefs_OLS

print('\n--OLS with SVD--')
print(tabulate(all_values(y_pred, y_test)))
```

```
--OLS with SVD--
--------  ------------
MSE        0.000592563
R2         0.0243426
Bias      36.2716
Variance  36.2368
--------  ------------
```

Perform Ridge Regression with different values for lamda and choosing the one with the best result.

In [7]:
```
alphas= np.logspace(-4, 5, 10)
MSE_R = 1
for val in alphas:
    beta, prediRidge = ridge_regression(x_train, x_test, y_train, val)
    mseRidge = mse(prediRidge, y_test)
    if (mseRidge <= MSE_R):
        alphaR= val
        Beta_R= beta
        pred_Ridge = prediRidge
        MSE_R = mseRidge

#Print results
print('\n--Ridge regression--')
print('The best value for alpha = ', alphaR)
print(tabulate(all_values(pred_Ridge,y_test)))
```

```
--Ridge regression--
The best value for alpha =  0.0001
--------  ------------
MSE        7.68038e-09
R2         8.76378e-05
Bias      36.2716
Variance  36.2715
--------  ------------
```

Perform Lasso Regression using sklear because this is the way we did in project 1.

In [10]:

```
#--------------Lasso Regression-------------
from sklearn.linear_model import Lasso #Using Sklearn as I did in 1st project
alphasL= np.logspace(-4, 5, 10)
regr = Lasso()
scores = [regr.set_params(alpha = alpha).fit(x_train, y_train).score(x_test, y
_test) for alpha in alphasL]
best_alpha = alphasL[scores.index(max(scores))]
regr.alpha = best_alpha
regr.fit(x_train, y_train)
pred_Lasso = regr.predict(x_test)

#Print results
print('\n--Lasso regression--')
print('The best value for alpha = ', best_alpha)
print(tabulate(all_values(pred_Lasso,y_test)))
```

```
--Lasso regression--
The best value for alpha =  0.0001
--------  ------------
MSE        1.59884e-06
R2         0.00126445
Bias      36.2716
Variance  36.2656
--------  ------------
```

Part c) Determine the phase of the two-dimensional Ising model

Now, we will use the two-dimensional Ising Model, and we will use the data sets generated by d by Mehta et al (https://physics.bu.edu/~pankajm/ML-Review-Datasets/isingMC/).

We will use a fixed lattice of L × L = 40 × 40 spins in two dimensions.

The aim of this section is to use logistic regression to train our model and predict the phase of a sample given the spin configuration, wheter it represents a order or a disorder state. Is it an order state when it is below the critical temperature, and a disordered state when it is above this temperature. The theoretical critical temperature for a phase transition is $TC \approx 2.269$ in units of energy.

The algorithm to resolve this part was:

1.- Read the data (based in the code from Metha et all (https://physics.bu.edu/~pankajm/ML-Notebooks/HTML/NB_CVII-logreg_ising.html).

2.- Write the code to perform Logistic Regression:

```
 i) Given a set of inputs, assign them to a category

 ii) Genetare the probabilities with a function that gives outputs between 0 and
1. (Sigmoid function)

 iii) Define a function that give us the parameters/weights. -> Cost function

 iv) In order to minimize the cost function we increse/decrese the weigths with th
e derivate of the loss function with respect to each weight. (Gradient Descent)

  vi) Update the weights and repeat until reach the optimal.

  vii) Predict the output using the sigmoid function
```

3.- Evaluate the model using the accuracy score

$$Accuracy = \frac{\sum_{k=1}^{n} I(t_i = y_i)}{n}$$

The following code and the theory discussed above is based on [https://medium.com/@martinpella/logistic-regression-from-scratch-in-python-124c5636b8ac (https://medium.com/@martinpella/logistic-regression-from-scratch-in-python-124c5636b8ac)](https://medium.com/@martinpella/logistic-regression-from-scratch-in-python-124c5636b8ac) and also on the lectures notes on [logistic regression (https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg-bs.html)](https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg-bs.html)

In [11]:
```
import numpy as np
from sklearn.model_selection import train_test_split
import sklearn.model_selection as skms
import pickle,os, glob
```

In [12]:
```
class LogisticRegression:
    def __init__(self, lr=0.01, num_iter=100000, fit_intercept=True, verbose=F
alse):
        self.lr = lr
        self.num_iter = num_iter
        self.fit_intercept = fit_intercept
        self.verbose = verbose

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))
    def __loss(self, h, y):
        return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

    def fit(self, X, y):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        # weights initialization
        self.theta = np.zeros(X.shape[1])

        #Standard Gradient Descent
        for i in range(self.num_iter):
            z = np.dot(X, self.theta)
            h = self.__sigmoid(z)
            gradient = np.dot(X.T, (h - y)) / y.size
            self.theta -= self.lr * gradient

    def predict_prob(self, X):
        if self.fit_intercept:
            X = self.__add_intercept(X)
        return self.__sigmoid(np.dot(X, self.theta))

    def predict(self, X):
        return self.predict_prob(X).round()
```

Reading the data

In [13]:

```
#Ising model
L=40 # linear system size
J=-1.0 # Ising interaction
T=np.linspace(0.25,4.0,16) # set of temperatures
#Read the files for the 2D data
filenames = glob.glob(os.path.join("..", "dat", "*"))
label_filename = "Ising2DFM_reSample_L40_T=All_labels.pkl"
dat_filename = "Ising2DFM_reSample_L40_T=All.pkl"
file_name = "Ising2DFM_reSample_L40_T=All.pkl"
# Read in the labels
with open(label_filename, "rb") as f:
    labels = pickle.load(f)

# Read in the corresponding configurations
with open(dat_filename, "rb") as f:
    data = np.unpackbits(pickle.load(f)).reshape(-1, 1600).astype("int")

# Set spin-down to -1
data[data == 0] = -1
```

Define the train and test sets with the corresponding slices of the data set for the ordered and disordered phases

In [14]:

```
# Set up slices of the dataset
ordered = slice(0, 70000)
critical = slice(70000, 100000)
disordered = slice(100000, 160000)

X_train, X_test, Y_train, Y_test = skms.train_test_split(
    np.concatenate((data[ordered], data[disordered])),
    np.concatenate((labels[ordered], labels[disordered])),
    test_size=0.95
)
```

Set the object of the class LogisticRegression with a learning rate = 0.1 and 100000 iterations.

In [15]:

```
model = LogisticRegression(lr=0.1, num_iter=100000)
model.fit(X_train,Y_train)
preds = model.predict(X_test)

accuracy = (preds == Y_test).mean()
print(accuracy)
print(model.theta)
```

```
0.47640485829959517
[ 8.25215081  0.6803206  -1.94657074 ... -0.90526201  0.78357407
 -2.06865464]
```

About the previous results:

I got a very low accuracy (lower than 50%) which means that my model it is not very good, but it was interesting see that I got different results here from the iPhyton console, where I got and accuracy = 0.7011, which is closer and also higher that the accuracy I got using sklearn. So maybe it could be somthing with jupyter notebook and not with the model itself.

In [17]:
```
from sklearn.linear_model import LogisticRegression

modelS = LogisticRegression(C=1e20)
modelS.fit(X_train, Y_train)
preds = modelS.predict(X_test)
accuracyS = (preds == Y_test).mean()
print(accuracyS)
print(modelS.intercept_, modelS.coef_)
```

```
0.694834008097166
[1.3309357] [[ 0.10347112 -0.36762477  0.3178758  ... -0.21941697  0.1148904
  -0.43908601]]
```

Part d) Regression analysis of the one-dimensional Ising model using neural networks

The goal now is to write a code that perform a multilayer perceptron model, implementing backpropagation algorithm.

"A multilayer perceptron (MLP) is a class of feedforward artificial neural network. An MLP consists of, at least, three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable" [1 (https://en.wikipedia.org/wiki/Multilayer_perceptron)](https://en.wikipedia.org/wiki/Multilayer_perceptron)

The algorithm to perform Backpropagation to train the model is:

1.- Inizialize the network collecting and pre-processing the data

2.- Define the model and architecture

3.- Propagate the network using feed forward

4.- Choose a cost function and an optimizer

5.- Train the Network

6.- Compute the back-propagate errors

7.- Predict the values for the test data

In [18]:
```
class Neural_Network:
    def __init__(self,  X_dat, Y_dat,epochs=10,
        batch_size=100,
        eta=0.1,
        lmbd=0.0):
        #parameters
        self.X_data_full = X_dat
        self.Y_data_full = Y_dat

        self.inputSize, self.n_features = X_dat.shape
        self.outputSize = 10
        self.hiddenSize = 50

        self.epochs = epochs
        self.batch_size = batch_size
        self.iterations = self.inputSize // self.batch_size
        self.eta = eta
        self.lmbd = lmbd

        #weights
        self.W1 = np.random.randn(self.n_features, self.hiddenSize)
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)
        #bias
        self.B1 = np.zeros(self.hiddenSize) + .01
        self.B2 = np.zeros(self.outputSize) + .01

        self.weights = [self.W1, self.W2]
        self.biases = [self.B1, self.B2]

    def sigmoid(self, s):
        return 1/(1+np.exp(-s))

    def sigmoidPrime(self, s):
        return s * (1 - s)

    def feed_forward(self):
        # feed-forward for training
        self.z_h = np.matmul(self.X_dat, self.W1) + self.B1
        self.a_h = self.sigmoid(self.z_h)
        self.z_o = np.matmul(self.a_h, self.W2) + self.B2
        self.probabilities = self.sigmoid(self.z_o)
        exp_term = np.exp(self.z_o)
        np.seterr(divide='ignore', invalid='ignore')
        self.probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True
)

    def feed_forward_out(self, X):
      # feed-forward for output
      z_h = np.matmul(X, self.W1) + self.B1
      a_h = self.sigmoid(z_h)

      z_o = np.matmul(a_h, self.W2) + self.B2
      exp_term = np.exp(z_o)

      probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)
      return probabilities
```

```
    def backpropagation(self):
        # error in the output layer
        error_output = self.probabilities - self.Y_dat[0]
        # error in the hidden layer
        error_hidden = np.matmul(error_output, self.W2.T) * self.a_h * (1 -sel
f.a_h)

        self.output_weights_gradient = np.matmul(self.a_h.T, error_output)
        self.output_bias_gradient = np.sum(error_output, axis=0)

        self.hidden_weights_gradient = np.matmul(self.X_dat.T, error_hidden)
        self.hidden_bias_gradient = np.sum(error_hidden, axis=0)

        if self.lmbd > 0.0:
                self.output_weights_gradient += self.lmbd * self.W2
                self.hidden_weights_gradient += self.lmbd * self.W1

        self.W2 -= self.eta * self.output_weights_gradient
        self.B2 -= self.eta * self.output_bias_gradient
        self.W1 -= self.eta * self.hidden_weights_gradient
        self.B1 -= self.eta * self.hidden_bias_gradient

    def predict(self, X):
        probabilities = self.feed_forward_out(X)
        return np.argmax(probabilities, axis=1)

    def train(self):
        data_indices = np.arange(self.inputSize)
        for i in range(self.epochs):
            for j in range(self.iterations):
                # pick datapoints with replacement
                chosen_datapoints = np.random.choice(
                    data_indices, size=self.batch_size, replace=False)
                # minibatch training data
                self.X_dat = self.X_data_full[chosen_datapoints]
                self.Y_dat = self.Y_data_full[chosen_datapoints]

                self.feed_forward()
                self.backpropagation()
```

Defining the parameters

In [19]:
```
epochs = 100
batch_size = 100
eta = 0.01 #learning rate
lmbd = 0.01
```

Call the Neural_Network class and train the model to predict the values for the test data

In [20]:
```
dnn = Neural_Network(x_train, y_train, epochs=epochs, batch_size=batch_size, e
ta=eta,  lmbd=lmbd)

dnn.train()
test_predict = dnn.predict(x_test)
```

Define a function that compute the accuracy score

In [21]:
```
def accuracy_score(y_test, Y_pred):
    return np.sum(y_test == Y_pred) / len(y_test)
```

In [22]:
```
print("Accuracy score on test set: ", accuracy_score(y_test, test_predict))
```

```
Accuracy score on test set:  0.235
```

About the previous result

I got a very small Accuracy score and I think is due the calculation of the probabilies when doing the forward propagation.

Conclusions

Again I wasn´t able to finish all the project because I got a lot of trouble in the understanding of some calculations. And since this is my first course programing this way, I get very delayed because many simple programming stuff. I have to be checking a lot the documentation and tutorial to program, and also I did not find the courage to go to ask in the lab sessions because I started late because I was busy with my other courses and I was afraid that everyone had a lot already done, and I was just starting. I promise to put more attention on that and stop being too shy and go and ask my question for the next project.