FYS-STK4155

Machine Learning

PROJECT 2

Henrik Løland Gjestang
henriklg@student.matnat.uio.no
Betina Høyer Wester
betiahw@student.matnat.uio.no
Mona Heggen
monahe@student.matnat.uio.no

Dato: November 12, 2018

Abstract

The main aim of this project is to study both classification and regression problems, starting with the regression algorithms studied in project 1. We will implement our own multilayer perceptron code and use it to study both regression and classification problems. The data we will look into is the so-called Ising model, and we will try to classify the phase (magnetization) of this model given a temperature.
Our findings are linear regression works well with one-dimensional Ising data, and for logistic regression deep learning works best, especially with tools like Keras and Tensorflow.

Contents

1 Introduction

This paper is written for Project 2 in the course FYS-STK4155, at the University of Oslo. The main objective of this paper is to study different methods of machine learning and look at how they can be used and differ from each other. A central objective of this study is to learn, evaluate and try to quantify the performance of various models on the same data set.

We will focus on supervised learning and use the same dataset for all methods. The methods we will investigate are:

- Regression methods (OLS, Ridge and Lasso)
- Binary classification (Logistic regression)
- Neural Networks

The dataset used in this project is the Ising model. We will follow closely the recent article of Mehta et al, arXiv 1803.08823.

ML techniques play an increasingly large role in many aspects of modern technology. Some examples are self driving cars, smart devices, optimizing energy consumption, advertisement and more. For physicists and people working in field of data science it is of high importance to understand ML basics.

2 Theory

a) Linear Regression

For information about Ordinary least squares, Ridge regression, Lasso regression and bootstrap, see Project 1 section 2.

b) Logistic Regression

Logistic regression is often used to solve problems where each data point is assigned to a specific class with discrete response values y_i, and our goal is to classify new data points. The idea of logistic regression is to use the logit-function (*Sigmoid* function) given by

$$p(t) = \frac{1}{1 + exp(-t)} = \frac{exp(t)}{1 + exp(t)} \tag{1}$$

to calculate the probability that a data point x_i belongs to a certain class, and choose the class with the highest probability. In the case where we have two classes and a model with two β values and y_i can be assigned to either 0 or 1, we have that

$$p(y_i = 0|x_i, \hat{\beta}) = \frac{exp(\beta_0 + \beta_1 x_i)}{1 + exp(\beta_0 + \beta_1 x_i)} \tag{2}$$

$$p(y_i = 1|x_i, \hat{\beta}) = 1 - p(y_i = 0|x_i, \hat{\beta}) \tag{3}$$

and we assign y_i to 0 if $p(y_i = 0|x_i, \hat{\beta}) > p(y_i = 1|x_i, \hat{\beta})$ and 1 otherwise.

c) Neural Network

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the biological nervous systems. The nervous system, such as the brain, is a vast network with interconnected neurons which is 'active' when the right inputs have a current flowing to them. They all work in unison to solve specific problems. ANNs are configured for problem-specific applications, such as pattern recognition or data classification. The way the neural networks learn is by adjusting the connections that exist between the neurons. The neurons exists in layers and they interact by sending signals in the form of mathematical functions between them. The layers can contain an arbitrary number of neurons.

The output of each neuron is the value of its activation function y, which is the sum of all the signals received from the other neurons connected to it.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{4}$$

There have been developed a wide variety of different ANNs but the most common characteristics is an input layer, an output layer and eventual layers in-between which are called hidden layers. We will mainly be using a neural network with just one hidden layer.

For each node i in the first hidden layer, we calculate a weighted sum z_i^1 of the input coordinates x_j,

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{5}$$

Here, b_i is the bias which is normally needed in case of zero activation weights or inputs. The value of z_i^1 is the argument to the activation function f_i of each node i. The variable M stands for all possible inputs to a given node i in the first layer. We define the output y_i^1 of all neurons in layer 1 as

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right) \tag{6}$$

where we assume that all the nodes in the same layer have the same activation functions. We could assume in the more general case that different layers have different activation functions but this increases complexity so we have chosen not to do this.

The output of neuron i in the second layer is

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \tag{7}$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{8}$$

where we have substituted y_k^1 with the inputs x_k. Finally, the ANN output reads

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right) \tag{9}$$

$$= f^3\left[\sum_{j} w_{ij}^3 f^2\left(\sum_{k} w_{jk}^2 f^1\left(\sum_{m} w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_m^3\right] \tag{10}$$

And if we generalize this expression to an multilayer perceptron, MLP, with l hidden layers, the complete functional form is

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(...f^1\left(\sum_{n=1}^{N_0} w_{mm}^1 x_n + b_m^1\right)...\right) + b_k^2\right) + b_1^3\right] \tag{11}$$

which illustrates a basic property of MLPs, that the only independent variables are the input values x_n.

d) Stochastic gradient descent

Given a dataset X, a model $g(\beta)$ and a cost function $C(X, g(\beta))$, we wish to find the values of β that minimize the cost function. In the cases where we are not able to solve this problem for β analytically we must use a numerical method to compute the minimum. The idea of gradient descent is that given a function $F(x)$ for $x = (x_1, ..., n_n)$, we try to find the global minimum by searching in the direction of the negative gradient $\nabla F(x)$, until convergence

$$x_{k+1} = x_k - \gamma_k \nabla F(x), \quad k \geq 0 \tag{12}$$

where the step length $\gamma_k > 0$. In our problem, we want to minimize

$$C(\beta) = \sum_{i=1}^{n} c_i(x_i.\beta) \tag{13}$$

the gradient is then given by

$$\nabla_\beta C(\beta) = \nabla_\beta \sum_{i=1}^{n} c_i(x_i.\beta) \tag{14}$$

For n datapoints, for every randomly picked minibach B_k with size M for $k = 1, ..., n/M$ we have that

$$\nabla_\beta C(\beta) \to \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i.\beta) \tag{15}$$

The gradient descent step is then

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i.\beta) \tag{16}$$

Where we choose starting point $\beta_0 = 1$. We stop when the gradient is zero or the norm of the gradient is smaller than a given threshold

e) Ising model

The Ising model is a binary value system, where the variables can be in one of two states (in our case ± 1) and the variables of the model can only take two different values (± 1). Since we in this project will use the model to simulate phase transition in a magnetic field, we will refer to the the variables as spin. We can describe the energy of a system by the one-dimensional Ising model with nearest-neighbor interactions

$$E[\hat{s}] = -J \sum_{j=1}^{N} s_j s_{j+1} \tag{17}$$

where N is the number of data points and Ising spin variables $S_j = \pm 1$. We will later estimate the coupling constant of the one-dimensional Ising model using linear regression.

To include phase transition to our model, we use the all-to-all Ising model:

$$E_{model}[s^i] = -\sum_{j=1}^{N}\sum_{k=1}^{N} J_{j,k} s_j^i s_k^i, \tag{18}$$

where s^i is a particular spin configuration. We will use the two-dimensional data computed at different temperatures in order to classify the phase of the two-dimensional Ising model.

3 Method

a) Linear regression

We look at the one-dimensional Ising model with nearest-neighbor interactions

$$E[\hat{s}] = -J\sum_{j=1}^{N} s_j s_{j+1} \tag{19}$$

We create random Ising states and use the model to calculate the energies of the states. We will then use ordinary least squares (OLS), Ridge regression and LASSO regression to predict the coupling strengths $J_{j,k}$ in the all-to-all Ising model given by

$$E_{model}[s^i] = -\sum_{j=1}^{N}\sum_{k=1}^{N} J_{j,k} s_j^i s_k^i \tag{20}$$

Since the model is linear in J, we can apply linear regression like in project 1 by writing the model on the form

$$E_{\text{model}}^i \equiv \mathbf{X}^i \cdot \mathbf{J} \tag{21}$$

Where $X^i = \{s_j^i s_k^i\}_{j,k=1}^N$

b) Logistic regression

We can view our problem as a binary classification problem where the two possible classes are ordered states (states below the critical temperature) and disordered states (states above the critical temperature). We can therefore use logistic regression to determine the phase of a sample given the spin configuration. Recall that we in OLS predicted $\hat{y}$ by

$$\hat{y} = X\omega \tag{22}$$

Where X is input data and ω are weights of the regression. The *Sigmoid* function of OLS is then given by

$$f(X\omega) = \frac{1}{1 + exp(-X\omega)} \tag{23}$$

Our goal is to minimize the cost function

$$C(X, \omega) = \sum_{i=1}^{n}\{-y_i log(f(x_i^T)) - (1 - y_i) log[1 - f(x_i^T)]\} \tag{24}$$

We use the data sets generated by Mehta et al. We use a fixed lattice of L $\times$ L = 40 $\times$ 40 spins and set the theoretical critical temperature for a

phase transition to TC = 2.3. We will use the accuracy score to measure the performance of the model on new data. The accuracy is defined as the number of images correctly labeled divided by the total number of images

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{25}$$

where

$$I = \text{Indicator function} = \begin{cases} 1 & \text{if } t_i = y_i \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

and t_i represents target and y_i represents outputs. Before doing this we need to optimize the parameters using stochastic gradient descent, as described in the theory section, and compare our result with outputs from scikit-learn's toolbox.

c) Neural Network

To find the optimal weights and biases we will be implementing a multilayer perceptron. The MLP is a popular and easy to implement approach to deep learning. It can be summarized as

1. A neural network with one or more layers of nodes between the input and the output nodes.
2. The multilayer network structure consists of an input layer, one or more hidden layers, and one output layer.
3. The input nodes pass values to the first hidden layer, its nodes pass the information on to the second and so on till we reach the output layer.

In a MLP with only linear activation functions, each layer of the neural network will only perform a linear transformation of its input. For this reason we will introduce some kind of non-linearity to the NN to be able to fit non-linear functions. One of the most used such function is the logistic *Sigmoid*

$$f(x) = \frac{1}{1 + e^{-x}} \tag{27}$$

As we have seen in feed forward networks, we can express the output of a single neuron in terms of basic matrix-vector multiplications. The only unknown is the weights w_{ij} and we need an algorithm for adjusting them to minimize our error. This leads us to the back propagation algorithm.

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{28}$$

To set up the back propagation algorithm we follow these steps

1. Set up the input data $\hat{x}$ and the activations $\hat{z}_1$ of the input layer and compute the activation function and the outputs $\hat{a}^1$.

2. Perform the feed forward until we reach the output layer and compute all $\hat{z}_l$ of the input layer and compute the activation function and the outputs $\hat{a}^l$ for $l = 2, 3, ..., L$.

3. Compute the output error δ^L by computing all

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial(a_j^L)}$$

4. Compute the back propagation error for each $l = L-1, L-2, ..., 2$ as in equation 28.

5. Update the weights and biases using gradient descent for each l, and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta\delta_j^l a_k^{l-1}$$

$$b_j^l \leftarrow b_j^l - \eta\frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta\delta_j^l$$

For our Neural Network we chose to go with a 2-layer network, ie. one hidden layer and one output layer. Our hidden layer uses the *ELU* activation function. We first tried with *Sigmoid* but this gave us poor results. In the hidden layer we put $\lambda = 1e^{-3}$ and 1600 neurons. Our outer layer has a linear activation function because we want the output to be all numbers, not between 0 and 1. To get good results We tested with different learning rates but found that MSE and R^2-scores would not converge on some values for η, the learning rate parameter. So we ended up with using $\eta = 10^{-6}$. This meant however that training our network would take a long time. After about 600 epochs we ended up with some decent results.

When classifying the Ising energies our main modification was to change the activation function to *Sigmoid* for the hidden layer. We weren't able to get very good accuracy with the *Sigmoid* either unfortunately. For further studies we would have liked to tested other activation functions for our hidden layer.

To verify our results we used Keras. Keras is a high-level neural network API written in Python, and capable of running on top of Tensorflow.

η	0.1
epochs	5
λ	0.01
hidden neurons	40
batch size	32

Table 1: Parameters for neural network in Keras.

The code for how we configure Keras can be found in the code section of the Appendix.

4 Results and discussion

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

We start off with finding the best α value for Ridge regression and best λ value for Lasso Regression, by comparing the result of different α- and λ-values and choosing the which results in the best R^2-score. In figure 3 and 2. From the plots we see that the performance of the models are best for α equal to 0.01 and λ equal to 0.0001, which is the same as in Mehta et al, [3]. Comparing our result to the results in Mehta et al, we see the optimal α and lambda values does not perform as good as our model. Computing our results, we use 80% of the 10000 generated data points for training, and the remaining data for testing. In Meta et al, they only used 400 data points for training a model with 1600 β values and 200 data points for testing, which may give an explanation to our different result. testing our code with the same amount of training and testing data, we get the same results. MSE gives us the best model, but in Mehta et al, Lasso was the only linear regression method able to achieve a performance up to 100% with both train and test set for $\lambda = 0.0001$

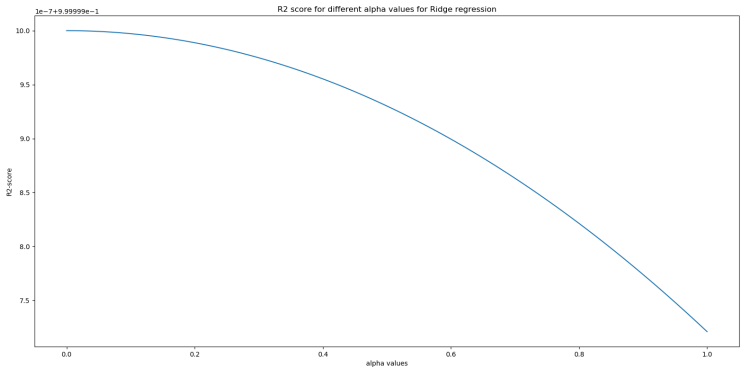

Figure 1: MSE and R^2-score for different α-values

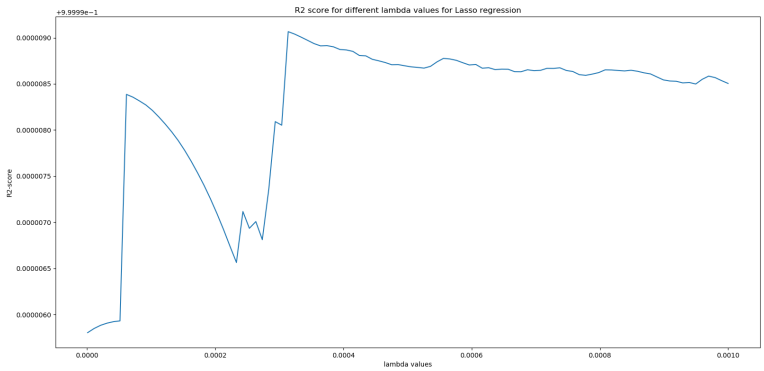

Figure 2: R^2-score for different lambda-values

comparing the models

	MSE	R^2	bias	variance
Ordinary Least squares	5.18e-28	1.0	4.48e-28	5.31e-29
Ridge regression	1.02e-06	0.99	1.02e-06	6.91e-29
Lasso regression	0.00017	0.99	0.00017	7.57e-29

Table 2: Bootstrap validation of different regression methods with Ising model. $\lambda = 0.0001$ and $\alpha = 0.01$

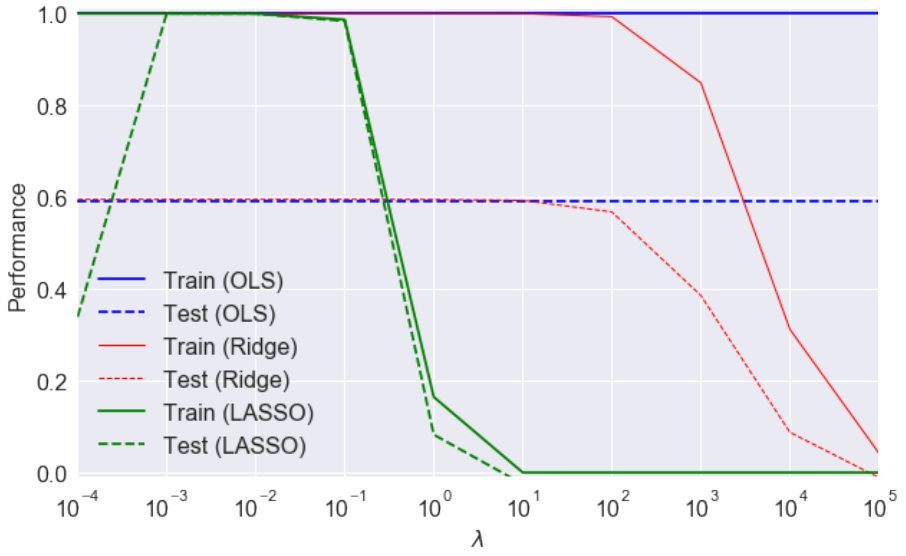

Figure 3: Train test performance for different alpha-values compared OLS, Ridge and LASSO [3]

A result which is presented in Mehta et al, and is represented in figure 3,

is that the models produces by linear regression models overfits the training data. This results was not possible to read from our results, due to the high R^2-score. Lasso provides the best results in Meta et al. It also gives a good model when we test with our parameters. This is because Lasso selects only the non correlated features while reducing the other coefficients (that correlates) to zero. This method is good for use cases with large number of features, because of this automatic feature selection.

b) Determine the phase of the two-dimensional Ising model using logistic regression with stochastic and standard gradient descent

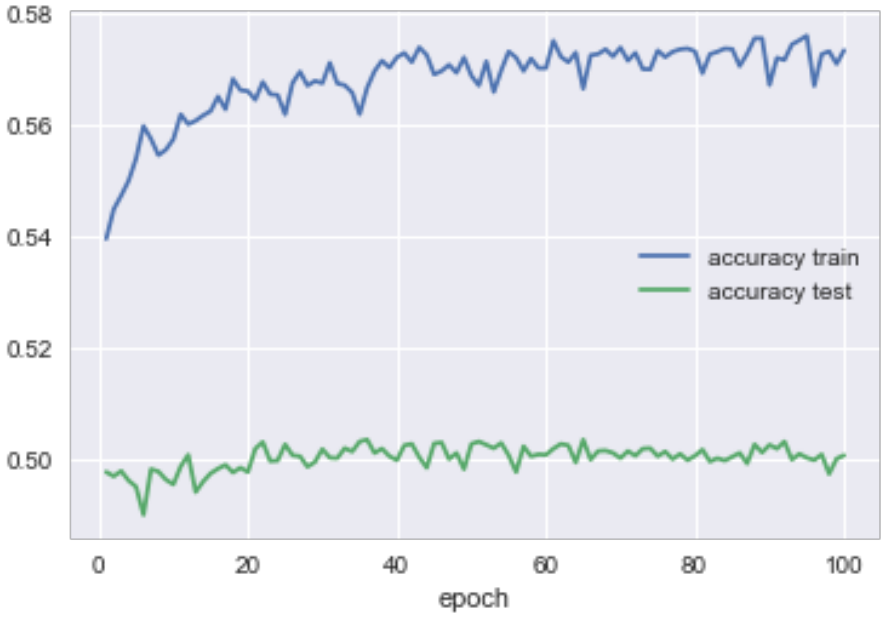

Figure 4: Stochastic accuracy pr epochs test and train. Epochs = 100, t0=50, t1=500

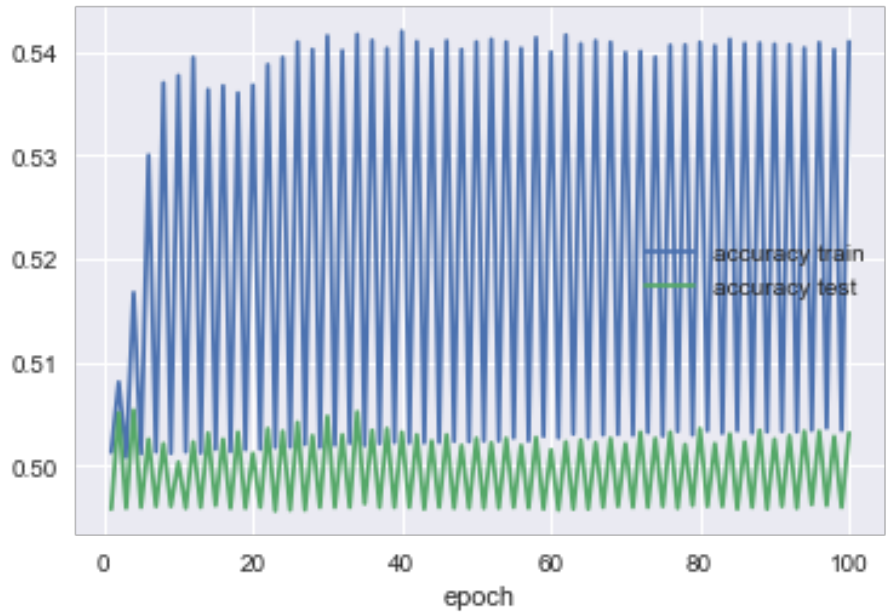

Figure 5: Standard gradient descent: Test and train accuracy pr iterations, max iterations = 100, learning rate = 0.1, 30% of data set

Logistic regression accuracy	train	test
30% of data Standard	54.1%	50.3%
30% of data Stochastic	57.3%	50.0%
80% of data Stochastic	54.0%	50.0%
80% of data Standard	50.6%	50.0%
All data stochastic	40%	

Table 3: Approximately accuracy using Logistic Regression with different gradient descent methods

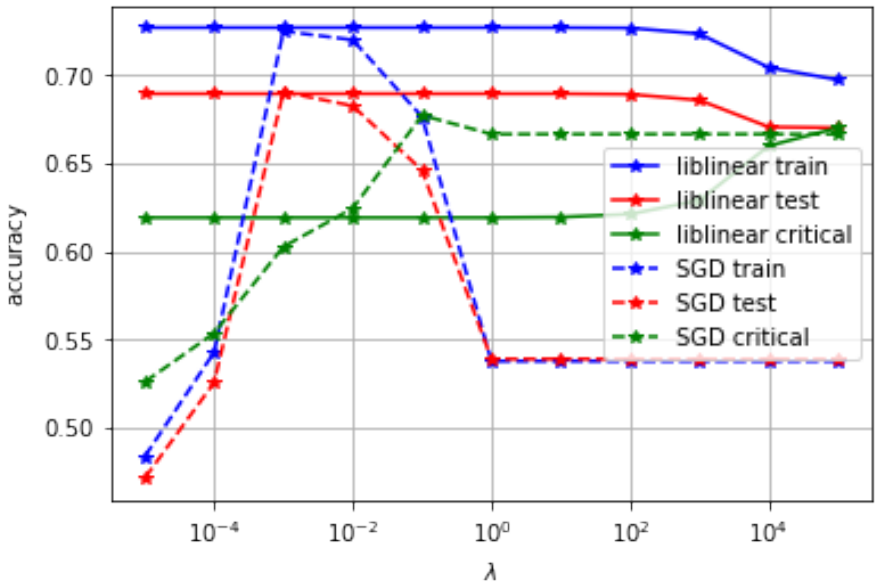

Figure 6: Mehta et al. results: Train test performance for different λ-values compared scikit's logistic regression (libliner), and stochastic gradient descent (SGD) [3]

The 2D Ising model data set is large and we choose to only use 30% of the original data for saving time and space. Further we consider only the two classes ordered and disordered states. We implement two methods to compute the accuracy: the standard gradient descent and a stochastic gradient descent. One of the main problem with these methods is to find the best parameters. We therefore test both methods with different parameters and ends up with these parameter:

- For standard gradient descent we use 100 iterations and 0.1 as the learning. This gives a accuracy converging to approximate 54.1%
- For stochastic gradient descent we use 100 epochs and set t0 to 50 and t1 to 500. The accuracy converges to approximate 57.3%

The results after testing with the two gradient descent methods:

- Use all data and all three classes: under 40%
- Use all data but only the binary classes ordered and disordered: 53-55%
- Use only 30% of data with binary classification approximate 57%

Neither methods is good. Both methods have approximate the same results like guessing. Where the stochastic method gives a slightly better results, but not good. We might get higher results if we use some parameters other than learning rate. This will still not give a good classification model for this problem. There might be several reasons for this.

- The variables might be correlated to each other.

- One of the main problem is that there are so many independent variables (40X40).
- It is necessary with more data.
- using regularization parameter

We can also see from the results that the test accuracy is much lower than the train accuracy. The train accuracy is very low, and gives approximately the same results that guessing would have given.

As we can see from [3], they get a better result than us. This is could be because that SciKit learn uses other and better regularization parameters and will therefore get better results than our results, but still logistic regression is not the best method to classify this problem. They only get up to 70%.

c) Neural Network

For our regression analysis of the one-dimensional Ising model using neural networks, er got the following result after 600 epochs: We could have tweaked

MSE	1.2137
R^2	0.9690

Table 4: Results for regression with neural network.

our model a bit more, by increasing the learning rate without getting unstable results we could have achieved a higher R^2-score.

After 20 epochs and batch size of 1000 elements, our best result was 65% accuracy. We believe this is due to the *Sigmoid* function. One function which has been shown to work better is the hyperbolic tangent. This activation function has become the most popular for deep neural networks.

When we ran the neural network with Keras we were able to obtain an accuracy of 99.99% in the classification problem.

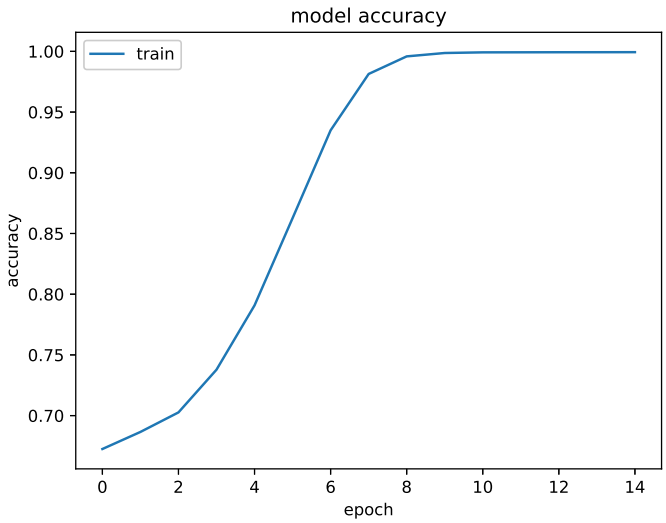

Figure 7: Accuracy of keras NN plotted for 15 epochs.

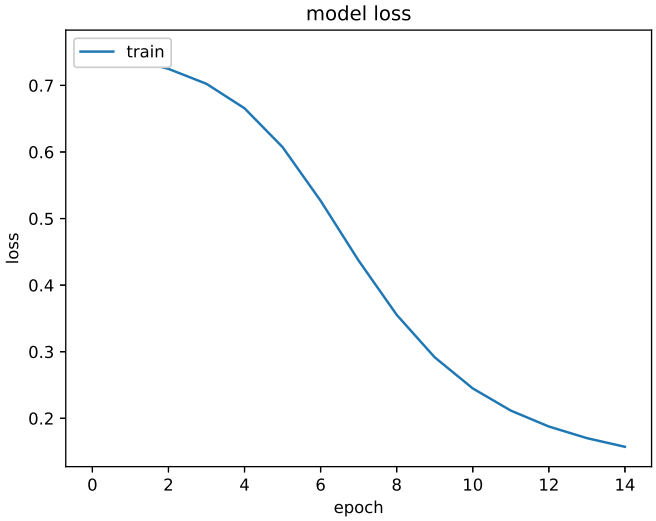

Figure 8: Loss of keras NN plotted for 15 epochs.

Here we see how Keras perform after each epoch. To get these nice plots we did however have to "dull down" the parameters, otherwise the NN learns so quickly that after the first epoch it has already reached 99% accuracy and the rest of the graph is just a flat line. The parameters we used to generate these figures were $\eta = 0.01$, $\lambda = 0.001$, epochs $= 15$, neurons $= 30$ and batch size $= 100$.

5 Conclusions and perspectives

In this project we have looked at several methods for linear and logistic regression, and found the performance for each model. We also compared regression methods with Neural Networks.

In general, the linear regression, Lasso and Ridge regression seems to give us good approximations to the Ising model, unlike the results in Mehta et al, but this is caused by the difference in the size of training an test set. From our results it seems that the Linear Regression methods is chosen over Neural Network and gives the best results for this kind of data set. Both the train and test results give approximately a perfect R^2-score to 1. This method is also the one that uses least computation time. But when we consider the amount of data variables and the unknown of correlation status of each variables, it might give cases of overfitting. The lasso is the method that also consider this problem. So with the right λ value, Lasso will be the best Linear Regression method in this case to insure to avoid over fitting.

When comparing the error (MSE) and the performance score (R^2) from the Neural Network and the Linear Regression methods, we can tell that the performance of Neural Network was worse than the Linear Regression methods. This indicates that Linear Regression methods is better than the Neural Network method for these kind of data set.

From Mehta et al and our own code we can observe that logistic regression is not a good approximation to classification problem with Ising model. This might be due to the amount of independent variables that might also correlate. As expected the stochastic gradient descent method is slightly better than the standard method.

Neural Network will give a better classification model. This method can handle a larger amount of variables. This is also confirmed by using Classification in the Neural Network that gives almost a perfect score. Metha et al gets approximately the same result.

6 Appendix

a) Programming code

Producing the data for the one-dimensional Ising model and estimating the coupling constant of the one-dimensional Ising model using linear regression

```
def DesignMatrix(states):
    N = np.size(states, 0)
    size2 = (L,L)
    xy = np.zeros(size2)

    size1 = (1,L**2)
    X_design = np.zeros(size1)

    size3 = (N,L**2)
    X = np.zeros(size3)

    for i in range(0,N):
        X[i] =  np.outer(states[i,:],states[i,:]).reshape(1,-1)#.ravel()
    return  X

def beta_model(model, xyb, z, param = 0.5):
    #OLS
    if (model == 'Linear'):
        betaLinear = np.linalg.pinv(xyb.T.dot(xyb)).dot(xyb.T).dot(z)
        return betaLinear
    #Ridge
    elif (model == 'Ridge'):
        I = np.identity(np.size(xyb, 1))
        Lambda = param
        betaRidge = np.linalg.inv(xyb.T.dot(xyb) + Lambda*(I)).dot(xyb.T).dot(z)
        return betaRidge
    #Lasso
    elif (model == 'Lasso'):
        (M,N) = np.shape(xyb)
        X = np.c_[xyb[:,1:]]
        poly = PolynomialFeatures(degree = 1)
        X_ = poly.fit_transform(X)
        clf = linear_model.Lasso(alpha=param, max_iter=100, tol=0.001,
        fit_intercept=False)
        clf.fit(X_, z)
        beta = clf.coef_.reshape(N,1)
        return np.asarray(beta)

def bootstrap(x, y, model, param = 0.5, n_bootstrap=100):
    # Randomly shuffle data
    data_set = np.c_[y, x]
    np.random.shuffle(data_set)
    set_size = round(len(x)/5)

    # Extract test-set, never used in training. About 1/5 of total data
    x_test = data_set[0:set_size, 1:]
    y_test = data_set[0:set_size, 0]
    test_indices = np.linspace(0, set_size-1, set_size)

    # And define the training set as the rest of the data
    y_train = np.delete(data_set[:, 0], test_indices, axis = 0)
    x_train = np.delete(data_set[:, 1:], test_indices, axis = 0)

```

```
    Y_predict = []
    MSE = []
    R2s = []
    #beta = 0
    for i in range(n_bootstrap):
        x_, y_ = resample(x_train, y_train)
        beta = beta_model(model, x, y, param).reshape(1600,)
        y_hat = x_test.dot(beta)
        Y_predict.append(y_hat)

        # Calculate MSE and R2-score
        MSE.append(np.mean((y_test - y_hat)**2))
        R2s.append(R2(y_test, y_hat))

    # Calculate MSE, Bias and Variance
    MSE_M = np.mean(MSE)
    R2_M = np.mean(R2s)
    bias = np.mean((y_test - np.mean(Y_predict, axis=0, keepdims=True))**2)
    variance = np.mean (np.var(Y_predict, axis=0, keepdims=True))
    return MSE_M, R2_M, bias, variance

def ising_energies(states,L):
    """
    This function calculates the energies of the states in the nn Ising Hamiltonian
    """
    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)
    #print(J.shape)
    return E

def predict(xyb,beta):
    """
    predicts values given a beta_model
    """
    zpredict = xyb.dot(beta)
    return zpredict

def mu(z):
    """
    Compute mean value
    """
    n = np.size(z, 0)
    z_mean = (1/n ) * np.sum(z)
    return z_mean

def calc_Variance(z, z_mu):
    """
    Compute variance
    """
    n = np.size(z, 0)
    #Sample variance:
    var_z = (1/n)* sum((z-z_mu)**2)
    return var_z

def MSE(z, z_tilde):
    """
    compute MSE of a model
    z = real z
    z_tilde = computed z
```

```
    """
    n = np.size(z, 0)
    #Mean Squared Error: z = true value, z_tilde = forventet z utifra modell
    MSE = (1/n)*(sum(z-z_tilde)**2)
    #error = np.mean( np.mean((z - z_tilde)**2, axis=1, keepdims=True) )
    return MSE

#def calc_R_2(y, y_tilde, y_mean):
#    n = np.size(y, 0)
#    R_2 = 1- ((sum(y.reshape(-1,1)-y_tilde)**2)/(sum((y-y_mean)**2)))
#    return R_2

def R2(yReal, yPredicted):
    """
    compute R2-score
    """

    meanValue = np.mean(yReal)
    numerator = np.sum((yReal - yPredicted)**2)
    denominator = np.sum((yReal - meanValue)**2)
    result = 1 - (numerator/denominator)
    return result
```

Logistic Regression with standard and stochastic gradient descent

```
import pandas as pd
import matplotlib.pyplot as plt
import tqdm
import copy
import time
from IPython.display import display
from math import log, exp
from numba import jit
from sklearn import datasets
from sklearn.model_selection import train_test_split

%matplotlib inline
#sns.set(color_codes=True)
class LogisticRegression:
    def __init__(self, X_train, Y_train, X_test, Y_test):
        np.random.seed(55)
        self.weights = None
        self.X_train = X_train
        self.Y_train = Y_train.reshape(len(Y_train), 1)
        self.X_test = X_test
        self.Y_test = Y_test.reshape(len(Y_test), 1)

        self.historytrain = []
        self.historytest = []

        self.target_epochs = 500
        self.target_max_iter = 10

    def f(x, w):
        return 1/(1 + exp(-x.dot(w)))

    def costFunction(X, labels, w):
        """
        :param X: (nx1600)
        :param labels: y (nx1)
        :param w: omega, weights (1600x1)
```

```
        :return:
        """
        n, p = np.shape(X)
        C = 0
        for i in range(n):
            C += -labels[i]*log(f(X[i, :].dot(w))) - (1-labels[i])*
            log(1 - f(X[i, :].dot(w)))

        return C

    def fit_standard(self, learning_rate=0.1): #=0.01):
        # Initialize weights
        max_iter = self.target_max_iter
        self.weights = np.random.randn(np.shape(self.X_train)[1], 1)

        for i in range(max_iter):
            # Compute probabilities
            z = np.dot(self.X_train, self.weights)
            pred = self.sigmoid(z)

            # Compute gradient, note division by data size
            gradient = self.X_train.T.dot((pred-self.Y_train))

            #Update weights
            self.weights -= learning_rate * gradient

            # Print progress
            if (i+1) % 100 == 0:
                print('{}% done'.format(100*(i+1)/max_iter))
            #print('Accuracy train: ', self.accuracy(self.X_train, self.Y_train))
            #print('Accuracy test: ', self.accuracy(self.X_test, self.Y_test))

            tmpEpoch = i + 1
            tmpAccTrain = self.accuracy(self.X_train, self.Y_train)
            tmpAccTest =  self.accuracy(self.X_test, self.Y_test)

            #self.history.append((tmpEpoch, tmpAcc))

            self.historytrain.append((tmpEpoch, tmpAccTrain))
            self.historytest.append((tmpEpoch, tmpAccTest))

            #print('Epoch nr {0} of {1}'.format(tmpEpoch, n_epochs))
            print('Accuracy train: ', tmpAccTrain)
            print('Accuracy test: ', tmpAccTest)


    def fit_stochastic(self, t0=50, t1=500):
        n_epochs = self.target_epochs
        nr_data_points = np.shape(self.Y_train)[0]  # Data points
        M = nr_data_points

        # initiate weights
        self.weights = np.ones([np.shape(self.X_train)[1], 1])

        for epoch in range(n_epochs):
            for i in range(M):
                random_index = np.random.randint(M)
                X_i = self.X_train[random_index:random_index+1, :]
                Y_i = self.Y_train[random_index:random_index+1, :]

                # Calculate gradient and update weights
                z = np.dot(X_i, self.weights)
```

```
                pred = self.sigmoid(z)
                gradient = X_i.T.dot((pred-Y_i))
                eta = t0/(epoch*M+i + t1)
                self.weights -= eta*gradient

            tmpEpoch = epoch + 1
            tmpAccTrain = self.accuracy(self.X_train, self.Y_train)
            tmpAccTest =  self.accuracy(self.X_test, self.Y_test)


            self.historytrain.append((tmpEpoch, tmpAccTrain))
            self.historytest.append((tmpEpoch, tmpAccTest))

            print('Epoch nr {0} of {1}'.format(tmpEpoch, n_epochs))
            print('Accuracy test: ', tmpAccTest)
            print('Accuracy train: ', tmpAccTrain)

    def sigmoid(self, z):
        # Sigmoid function
        return 1 / (1 + np.exp(-z))

    def getWeights(self):
        return self.weights

    def loss_function(self, pred, Y):
        # Compute loss function (normalized)
        return (-Y * np.log(pred) - (1 - Y) * np.log(1 - pred)).mean()

    def predict_threshold(self, X, threshold=0.5):
        # Predict
        return self.sigmoid(np.dot(X, self.weights)) >= threshold

    def accuracy(self, X=None, Y=None):
        if X is None:
            X = self.X_test
        if Y is None:
            Y = self.Y_test
        # Compute accuracy using test data

        I = self.predict_threshold(X) == Y
        return np.sum(I)/np.shape(X)[0]

if __name__ == '__main__':

    # data contains 30\% of data set with only ordered and disordered states
    data = np.load('test_set.npy')
    X = data[:, 0:1600]
    Y = data[:, -1]

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.30)

    logreg = LogisticRegression(X_train, Y_train, X_test, Y_test)

    logreg.fit_standard()
    #logreg.target_epochs = 100
    #logreg.fit_stochastic()

    df1 = pd.DataFrame(logreg.historytrain, columns=['epoch', 'accuracy train'])
    df2 = pd.DataFrame(logreg.historytest, columns=['epoch', 'accuracy test'])

    ax = df1.plot(x='epoch', y='accuracy train')
    df2.plot(ax=ax, x='epoch', y='accuracy test')
```

Reading in the data from file

```
def read_data_set(train_size=80000,validation_size=5000):
    import pickle
    from sklearn.model_selection import train_test_split
    from keras.utils import to_categorical
    import collections

    L=40 # linear system size

    # load data
    file_name = 'Ising2DFM_reSample_L40_T=All.pkl'
    data = pickle.load(open(file_name,'rb'))
    data = np.unpackbits(data).reshape(-1, 1600)
    data=data.astype('int')
    data[np.where(data==0)]=-1

    file_name = "Ising2DFM_reSample_L40_T=All_labels.pkl"
    labels = pickle.load(open(file_name,'rb'))

    # divide data into ordered, critical and disordered
    X_ordered=data[:70000,:]
    Y_ordered=labels[:70000]

    X_critical=data[70000:100000,:]
    Y_critical=labels[70000:100000]

    X_disordered=data[100000:,:]
    Y_disordered=labels[100000:]

    del data,labels

    # define training and test data sets
    X=np.concatenate((X_ordered,X_disordered))
    Y=np.concatenate((Y_ordered,Y_disordered))

    del X_ordered, X_disordered, Y_ordered, Y_disordered

    # pick random data points from ordered and
    # disordered states to create the training and test sets
    X_train,X_test,Y_train,Y_test=train_test_split(X,Y,train_size=train_size)

    del X,Y
    return X_train,Y_train
```

Classification with Keras

```
import tensorflow as tf
from keras import regularizers
import numpy as np

def to_binary(y, threshold):
    n = len(y)
    binary = np.zeros((n,))
    for i in range(n):
        if y[i] <= threshold:
            binary[i] = 0
        else:
            binary[i] = 1
    return binary

X,y = read_data_set()
print ("X:",np.shape(X), "y:", np.shape(y))

# Parameters
Xm, Xn = X.shape
epochs = 5
eta = 0.1
lmbd = 0.001
n_hidden_neurons = 40
batch_size = 32

# Setting up the network
clf = tf.keras.Sequential()
clf.add(tf.keras.layers.Dense(n_hidden_neurons, activation = 'sigmoid', ...
        input_dim = Xn, kernel_regularizer = regularizers.l2(lmbd)))
clf.add(tf.keras.layers.Dense(1, activation = 'sigmoid', ...
        kernel_regularizer = regularizers.l2(lmbd)))
sgd = tf.keras.optimizers.SGD(lr = eta)
clf.compile(optimizer=sgd, loss='binary_crossentropy', metrics=['accuracy'])

# Training
clf.fit(X, y, epochs = epochs, batch_size = batch_size, verbose = 1)

# Statistics
yhat = np.reshape(clf.predict(X), (Xm,))
yhat = to_binary(yhat, 0.5)
Accuracy_train = np.sum(y == yhat) / Xm
```

Bibliography

[1] Pankaj Mehta, Ching-Hao Wang, Alexandre G. R. Day, and Clint Richardson. *A high-bias, low-variance introduction to Machine Learning for physicists*. (March 26. 2018).

[2] The textbook of Trevor Hastie, Robert Tibshirani, Jerome H. Friedman, The Elements of Statistical Learning, Springer, chapters 3 and 7 are the most relevant ones for the analysis here.

[3] The notebooks of Pankaj Mehta, Associate Professor of Physics, Boston University, Department of Physics. NB_CIX-DNN_ising_TFlow, NB_CVII-logreg_ising and NB_CVII-linreg_ising