FYS-STK3155 – Applied data analysis and machine learning

Autumn 2018

Project 2

Vincent David Friedrich &

Frederik Stihler

11.11.2018

UiO

University of Oslo

Deadline: 12.11.2018

Prof. Morten Hjorth-Jensen

Abstract

The topics of this project are linear regression, logistic regression and the development of a neural network. The data used for the project is derived from the one and two dimensional Ising model. The overall goal is to experience that specific data can be analyzed in different ways and the performance of our analysis depends highly on the chosen method and on various parameters such as for instance the learning rate or the regularization parameters.

Different linear regression methods were used for estimating the coupling constant in the one-dimensional Ising model and it was found that the Lasso regression performs the best for a specific range of penalization, while the Ordinary Least Squares method and the Ridge regression were overfitting the data.

The two-dimensional Ising model data was used for a classification case, where the applied techniques should determine the state of a given sample. In order to perform the logistic regression, the Newton-Raphson method was used as the gradient descent solver. As this method requires to approximate the inverse of the Hessian in case it is singular, it was made use of the SVD decomposition which led to convergence problems. Using Stochastic Gradient Descent turned out to be the most promising gradient descent solver as the highest accuracy on both training and test data was obtained for well-adjusted hyperparameters.

Another approach to tackle a logistic regression problem is to develop a neural network, in our case a fully-connected multilayer perceptron (MLP) with backpropagation consisting of three layers (one input layer, one hidden layer, one output layer). Provided that the network consists of enough neurons in the hidden layer and well-chosen learning rates and regularization parameters, this resulted in a significant higher accuracy on training and test data compared to the logistic regression.

1 Contents

2 Introduction

The focus of this project is to study classification and regression methods. More specific, linear and logistic regression are applied and subsequently, a neural network is set up. This is done by producing a multilayer perceptron (MLP), which is a fully connected feed-forward network using non-linear activation functions (Hjorth-Jensen, 2018).

The classification model that is implemented in this project should be able to classify new data into predefined categories on the basis of training data for which the correspondent category is known. In machine learning, this classification problem belongs to the supervised learning methods. As such, it represents a valuable tool for making yes-or-no predictions for example in deciding if a tumor is cancerous or not. Classification techniques can also be extended to include more than only two possible categories.

The neural network that is set up is an MLP and consists of at least three layers: an input layer, one or more hidden layers and an output layer (Hjorth-Jensen, 2018). Neural networks have revolutionized machine learning and are inspired by the biological structure of neurons in a brain. In a perceptron model, information and errors are fed from the front to the back, a process that is called backward propagation. In each step, the weights at the nodes in the network are updated, which can be interpreted as a learning procedure (Le, 2018). Furthermore, different activation functions are tested. MLPs and other types of Neural networks have a vast range of application in different industries and can be used for example in banking and finance for pricing securities, in medicine to identify diseases based on medical records or in data analysis for image and language recognition.

In order to be able to study the performance of the various regression methods, classifications and networks, training data from the Ising model is utilized for this experiment. First, regression methods are used to estimate the coupling constant for the energy in the one-dimensional Ising model. Here, Ordinary Least Squares, Ridge and Lasso regression are applied and compared. Afterwards, the data set of the two-dimensional Ising model at different temperatures is used for studying a binary classification method. In this case, logistic regression is implemented to determine the phase of the Ising model. The classification model should then predict, whether the Ising model is in an ordered or disordered state. Thereafter, neural networks are used to perform the same kind of analyses and predictions. The report is concluded by a comparative analysis of the algorithms and methods used.

The codes in form of Jupyter notebooks, as well as other supporting material can be found in the github repository at: *https://github.com/Fredesti/Proj2_ML_FS_VF*

3 Methodology

For the implementation of the project the programming language Python 3.0 was used. The relevant python packages for analysis, computations and visualization of the data were *numpy, scikit-learn, scipy, seaborn* and *matplotlib.* Regarding the information and theoretical background on regression, classification and neural networks, most of the techniques applied are based on the lecture notes of the course FYS-STK3155 (“Applied data analysis and machine learning”) of autumn semester 2018 at the department of physics at the university of Oslo in Norway, as well as on the books “Neural Networks and Deep Learning” by Michael Nielsen (2015) and “The Elements of Statistical Learning” by Hastie et al. (2009) referenced in the appropriate sections of this report.

The basis data set for the implementation of the different supervised learning techniques in this project is provided by the Ising model. The model is named after the German physicist Ernst Ising and is concerned with the physics of phase transitions. These transitions may occur when a relatively small change in temperature leads to a large-scale change in the state of a system (Cipra, 1987, p. 937). Furthermore, the model can be interpreted as mathematical model of ferromagnetism. The variables in the model, called the Ising variables, can take two values only, which are here interpreted as spins. In one dimension, nearest neighbor interactions are considered and phase transitions do not exist at finite temperatures (Hjorth-Jensen, 2018). For the binary classification, a two-dimensional Ising model in a fixed lattice of 40 x 40 spins is the basis. To each lattice site, one of the two possible values, 1 or -1 is assigned. Such an assignment is called a spin configuration. The phase transition that may occur in the two-dimensional Ising model is called spontaneous magnetization and the critical temperature for it is found to be close to 2.3°C. The energy of the system, in the most basic form, can be defined as (Hjorth-Jensen, 2018):

$$E = -J \sum_{<kl>}^{N} s_l s_k$$

where $s_k = \pm 1$, N is the total number of spins and J is the coupling constant that expresses the strength of the interactions between neighbor-spins. The one-dimensional Ising model is used for attempting to estimate the coupling constant with linear regression and a neural network and the two-dimensional Ising model offers a binary classification scenario, where the possible states are ordered and disordered. The aim of the application of the logistic regression is to predict the phase of a sample, when given the spin configuration for the lattice. The ordered state represents a net magnetic moment and appears below the critical temperature. Above the critical temperature, an unmagnetized state, the disordered state, is reached (Cipra, 1987, pp. 940-941).

The regression methods used in the first part of the project were Ordinary Least Squares (OLS), Ridge and Lasso. As the Ising model and the problem for estimating the coupling constant J can be stated as a linear regression exercise by choosing the all-to-all Ising model, the application of the above mentioned regression methods is appropriate. A quick repetition of the theory behind those methods can be found in the appendix (Appendix 1).

Logistic regression is used for the identification of the Ising state of a given spin configuration in the two-dimensional Ising model. As the task of classifying the state of the model is a typical machine learning classification problem, logistic regression is assumed to be suitable for it. Logistic regression in general is a method that is often used for binary classification problems. The function at its core is the so called sigmoid function (or logistic function). It represents the probability for a data point to belong to one of the two categories and consequently, the logistic regression is a soft classification model. Compared to the linear regression, one cannot use the same cost functions (see Appendix 1), due to the fact that in logistic regression, the class values are to be predicted. For this purpose, the loss function for misclassifying is a logarithmic function, the log-likelihood. The minimization of this function leads to a non-linear problem, which requires gradient descent methods for solving (Hjorth-Jensen, 2018). The accuracy of a classification model can easily be calculated as the percentage of correctly classified data points and therefore facilitates the performance measurement and helps to identify overfitting when comparing results on test and training data.

The same tasks that were executed by the linear regression methods and the logistic regression are finally also performed by a neural network, meaning the estimation of the coupling constant in the one-dimensional Ising model and the classification of the states in the two-dimensional Ising model. This incorporates the setup of a multilayer perceptron model and a back-propagation algorithm. Within a neural network, incoming signals at a node are forwarded to the next layer based on a chosen activation function. In this project an MLP with one hidden layer was implemented as a fully connected, feed-forward network, meaning that a node is connected to all nodes in the following layer. For each connection between the nodes there exists a weight variable, that is updated in the back-propagation process. The procedure of the back-propagation "repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector" (Rumelhart, Hinton, & Williams, 1986). Whether a network is able to learn quickly or not is for example dependent on the size of the derivative of the cost function with respect to the weights, which is a calculation required for updating the weights as part of the back-propagation. For this project, this is something that cannot be predetermined and has to be explored during the implementation of the network. Similar mechanisms apply to the activation function, leaving a wide range of possibilities to tune the effectiveness and accuracy of the model. Moreover, the learning rate and regularization parameters for the weight size also immensely influence the network's performance.

In addition, the minimization problems in these algorithms also ask for gradient descent methods. Using a stochastic gradient descent in this case can decrease the risk of convergence problems and can reduce calculation time, as not all available data is processed. Finally, as the MLP model can be very flexible, for instance by varying the number of nodes in the hidden layer or changing the activation function, it is expected to suit well the purpose of estimating the coupling constant and classifying the Ising states. For the specific tasks of this project that the network is supposed to perform, labelled data is easily available, as one can simply generate the training data by oneself. Hence, the common limitation for neural networks of lacking availability of homogenous and labelled data is avoided.

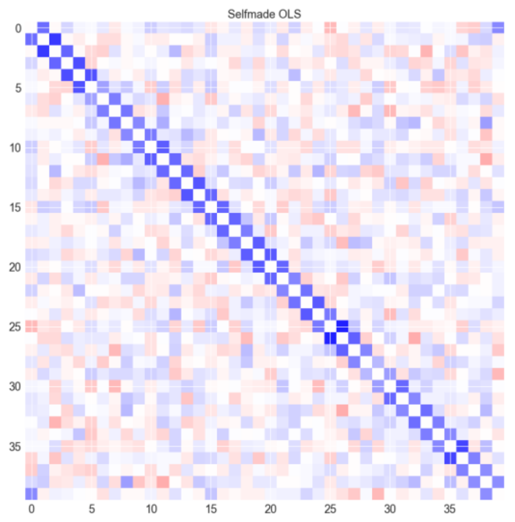
4 Implementation and Results

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

For the estimation of the coupling constant in the one-dimensional Ising model, several regression methods were applied. The methods used were Ordinary Least Squares regression (OLS), Ridge regression and Lasso Regression. A penalization parameter λ on the size of the regression coefficients is introduced in the Ridge and Lasso regressions.

During the implementation of the OLS regression, it was found that the minimization of the cost function led to a singular matrix problem, preventing the determination of the regression coefficients. As an alternative approach, a singular value decomposition (SVD) was used. For the implementation of the SVD, a pseudoinverse was calculated, while the use of the QR-factorization that reduces computational costs was waived. Due to the fact that the codes and functions for the OLS and other regressions in Project 1 were only designed for the specific polynomial approximation for defined degrees and used a matrix inversion, their application on the Ising model and new data was very limited. As a result, new codes based on the notebook 4 of Mehta et al. (2018) have been created for the calculations. In order to validate these results, a control computation in scikit-learn for the standard OLS was done. The Ridge regression was also implemented with and without the usage of scikit-learn features, while for the Lasso regression only the functions provided in the scikit-learn package were used.

In figure 1, one can see the visualization of the coupling coefficients found with the OLS method and plotted as a matrix. On the righthand side, the results from scikit-learn are shown. This figure suggests that the OLS that was implemented performs very similar to the linear regression of scikit learn. The $J_{j,k}$ entry of the matrix J, represents the coupling strength from the j-th to the k-th Ising spin variable in the system. Consequently, J appears to be symmetrical, which is supported by the visualizations below.



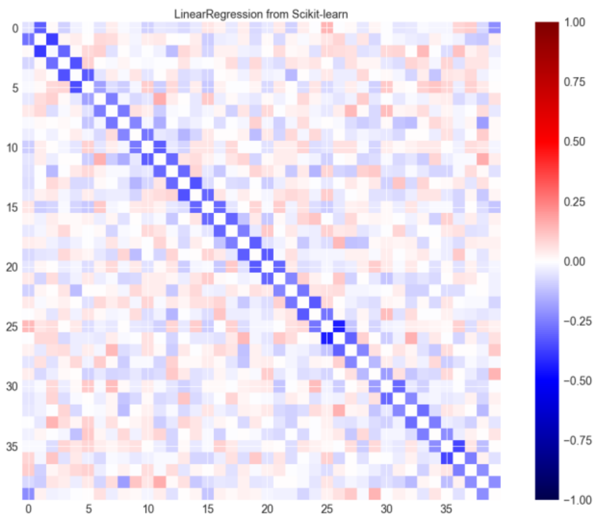
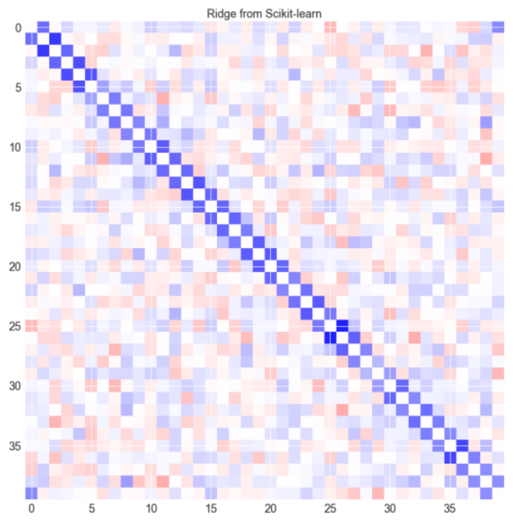


Fig.1: Visualization of coupling coefficients J, plotted as a matrix (left: OLS with own code; right: scikit-linear regression)

Another observation is that the coupling strength of nearest-neighbor interactions is at roughly -0.5. The nearest-neighbor interactions can be found on the entries next to the diagonal, i.e. the entries $J_{j,k}$ with $|j-k|=1$. Coupling strengths that apply to Ising spin variables that are not immediate neighbors are observed to be less than 0.5 in absolute value.

Compared with the coefficients of the OLS regression, the Ridge regression yields similar results. This can be seen in Fig. 1 & 2. Note that the coupling constant matrix J of the Ridge regression also appears to be symmetric. This is not the case for the results of the Lasso regression (see Fig. 2). The entries for coupling coefficients of Ising spin variables that are not direct neighbors are zero. Only coefficients $J_{j,j+1}$ show significant values close to -1. Exceptions are very weak values of $J_{j,j-1}$ entries.



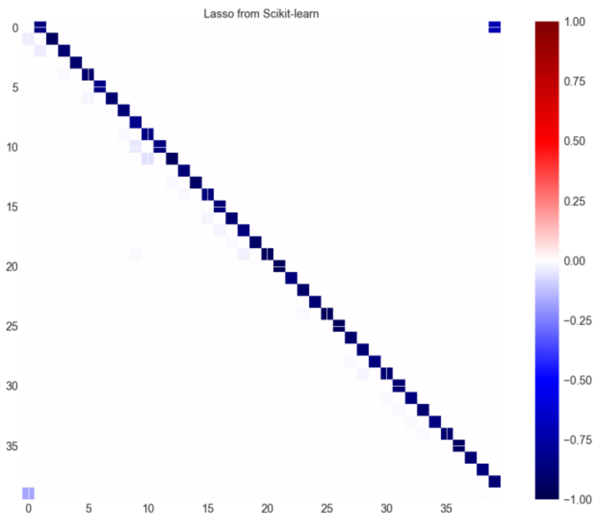


Fig.2: Visualization of coupling coefficients J, plotted as a matrix (left: Ridge scikit; right: Lasso scikit; penalization parameter λ=0.1)

In conclusion, one has to be careful with the information about symmetry that the regression methods deliver about the unknown model that created the data. This information can vary for different regression methods and penalization parameters. In the way the data was created for this experiment in the code, no symmetry was defined:

```
### define Ising model aprams
# system size
L=40

# create 10000 random Ising states
states=np.random.choice([-1, 1], size=(10000,L))

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)
    return E
# calculate Ising energies
energies=ising_energies(states,L)
```

For a further analysis of the performance of the models, the MSE of the models of OLS, Ridge and Lasso regression was plotted for varying values of the penalization parameter λ. This was done similarly to the code of the accompanying notebooks of Mehta et al. (2018). The graph can be found in figure 3:

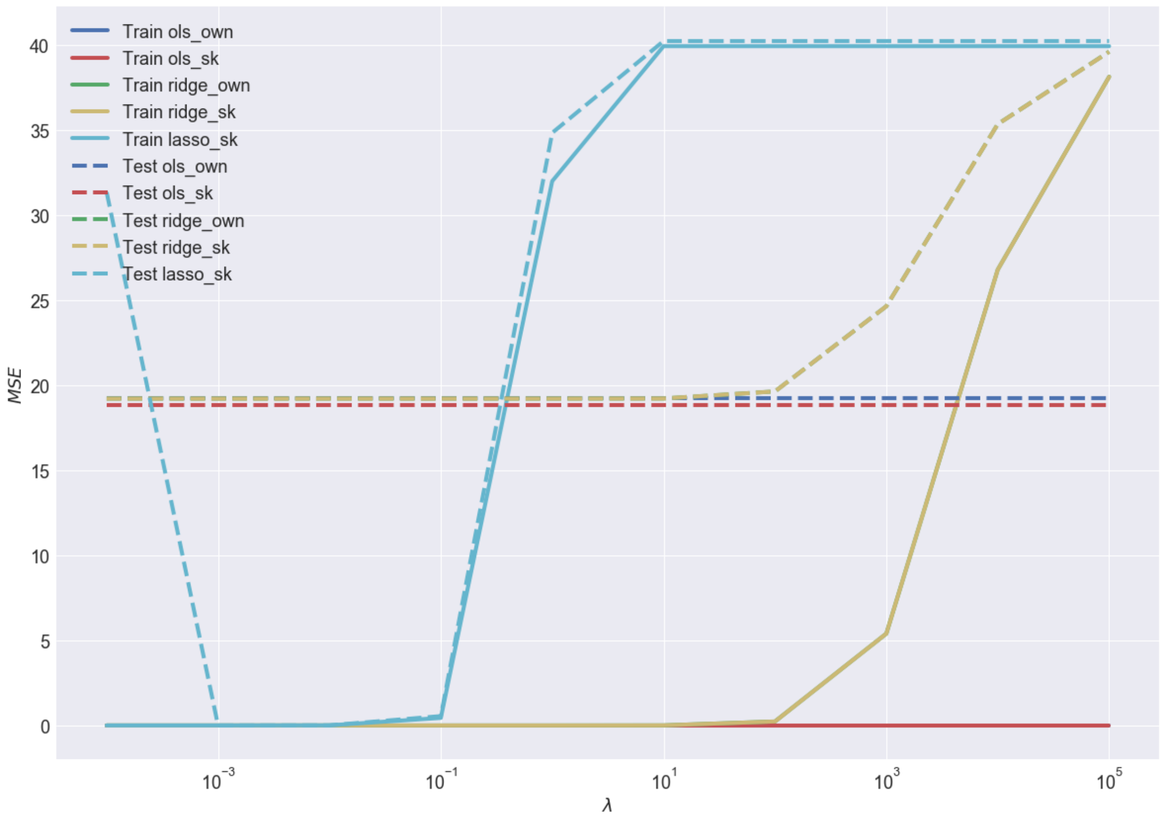


Fig.3: Performance of different regression methods for varying penalty λ, measured as MSE

As the OLS does not include a penalty on the coefficient size, its performance is constant under different values for λ. However, the OLS regression performs poorly on test data, with a high MSE of around 19. This indicates that the model with OLS is overfitting the data and the generalization to new data is not accurate. For the Ridge regression, similar observations apply. For large values of λ, the training and test curve for Ridge explode. Although Ridge regression is stable for a smaller λ, it also overfits the data as the test error differs significantly from the performance on training data.

Lasso regression on the contrary, seems to reach an optimum for small values of λ at around 10^{-2}. At this optimum, the model performs very well on test data with an MSE close to 0. A very similar behavior can be observed in Tab.1, which shows the R^2-score of the models created without scikit-learn (except for the Lasso) on test data. OLS and Ridge are performing poorly with an R^2-score of around 0.5, while the Ridge regression for large values of λ eventually diminishes. The Lasso however has an optimum for values in the range of 1.e-03 to close to 0.1. In this range it nearly reaches 1.0 on the test data, making it a very well model. The coupling constants in J for this model are estimated to include nearest-neighbor interactions only, which is in accordance with how the data for the one-dimensional Ising model was originally generated.

Tab. 1: R^2-score of different regression methods on test data for varying penalty λ

R^2	lamda									
	1.e-04	1.e-03	0.001	0.1	1	10	100	1000	1.e-04	1.e-05
OLS	0.52247	0.52247	0.52247	0.52247	0.52247	0.52247	0.52247	0.52247	0.52247	0.52247
Ridge	0.52247	0.52247	0.52247	0.52247	0.52247	0.52231	0.51197	0.38743	0.12137	0.01574
Lasso	0.20906	0.99998	0.99986	0.98638	0.13400	0.00000	0.00000	0.00000	0.00000	0.00000

Next, a bias-variance decomposition was performed for the OLS model. The result can be seen in Tab.2. To generate the decomposition, a 10-fold cross-validation was used by adapting the code of the previous project, following the recommendation of Hastie et al. (2009, p. 242) to find a compromise between bias and variance of the model. The bias-variance decomposition was calculated as in Hastie et. al (2009, p.223) and revealed that the MSE is made up of the squared bias and variance in similar proportion. The bias is the deviation between the true mean and the mean of the model. As the variance is also at an intermediate level, it suggests that there is a good balance in the complexity of the model for fitting the data, which contradicts with the findings from the analysis of the MSE. However, it is noticeable that the MSE calculated here (around 7) is significantly less than the one in the preceding analysis, where it was around 19 for OLS. This might be due to the fact that in Fig.3, the training data was only made up of 4% of the total available data set. Consequently, the model might not be able to perform ideally on test data. For the bias-variance decomposition on the contrary, the training data was 90% and the model was tested on the remaining 10% of the data. Hence, the MSE might be lower in this analysis.

Tab. 2: Bias-variance analysis using a 10 fold cross-validation

bias-variance decomposition	OLS
Prediction Error (MSE)	6.98917515
Bias ^2	3.25473848
Variance	3.34692604
MSE - Bias^2 - Variance	0.38751063

The performance measurement of the various methods in this section was targeted on how well the prediction of the energies per model was done. That means that MSE and R^2-score were applied on the data of the energies. However, this does not entirely reflect on the performance of the models in estimating the coupling constants in the one-dimensional Ising model. As a result, it is important to also analyze the estimated constants themselves. For instance, a model that uses completely different coupling constants compared to the true values, could still be able to closely predict the energies, while it fails to accurately describe the true constants J.

4.2 Phase determination of the two-dimensional Ising model using logistic regression

In this part of the project, we were supposed to use logistic regression in order to define the phases of the two-dimensional Ising model. In concrete terms, this means training our model to predict the phase of a sample given its spin configuration, whether it represents a state above or below the critical temperature (which is approximately 2.3 units of energy). States below the critical temperature are called ordered, while states above are called disordered.

As we face a binary classification problem, applying Newton-Raphson method to determine the optimal parameters of the logistic regressor is a natural choice due to the fact that we are able to set up the equations required for implementing the Newton-Raphson algorithm. After having read in the data provided by the notebook of Mehta et al (2018), one can see that we have 160000 spin configurations consisting of 1600 spins each.

As a result, we have in theory 1601 predictors and the sigmoid function looks the following way:

$$\textbf{(I)} \quad p(\hat{\beta}\hat{x}) = \frac{\exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}{1 + \exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}$$

where x_k denotes the kth spin variable and β_k the corresponding parameter (Hjorth-Jensen, 2018). Furthermore **(I)** can be interpreted as the probability of one spin configuration sample being ordered given parameter $\beta = \beta_0 \ldots \beta_p$ and spin variables $x = x_1 \ldots x_p$. The overall goal in this part of the project is to determine the optimal parameter β to classify spin configurations that were not part of the training data.

Implementing Newton-Raphson

In order to simplify the final Newton-Raphson function, we generated several python routines in advance. Regarding **(I)**, one can see that parameter β_0 is not dependant on any spin variable x_k, so we have to modify the input data by appending an additional 1 to each spin configuration.

```
def Xmodified(X):
    #input1  : designmatrix X
    #output1 : modified designmatrix --> column containing ones is added
    #function modifies the input designmatrix
    # column with ones is added
    rows = X.shape[0]
    onearray = np.ones([1,rows])
    Xt = np.transpose(X)
    Y=np.concatenate((onearray,Xt))
    Z = np.transpose(Y)
    return Z
```

The implementation of **(I)** looks the following way:

```python
def probability(beta,x):
    # input1 :  parameter beta
    # input2 :  predictors (in our case spin variables)
    # output1 : probability for state y = 1 given beta and x

    expo = beta.dot(np.transpose(x))
    y = np.exp(expo)/(1+np.exp(expo))
    return y
```

Based on the lecture notes the following properties have to be calculated:

Diagonal matrix **W**

W with $\mathbf{W_{k,k} = p(y_k | x_k, \beta)(1- p(y_k | x_k, \beta))}$ for k=1..n

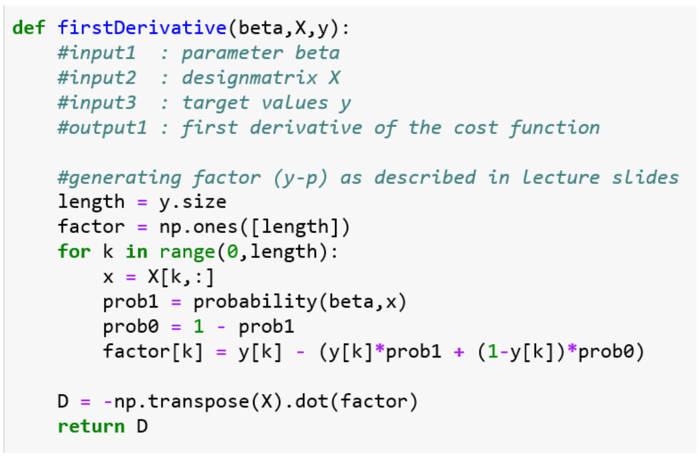
$\mathbf{W_{i,j} = 0}$ for $i \neq j$, i,j = 1…n

Python implementation:

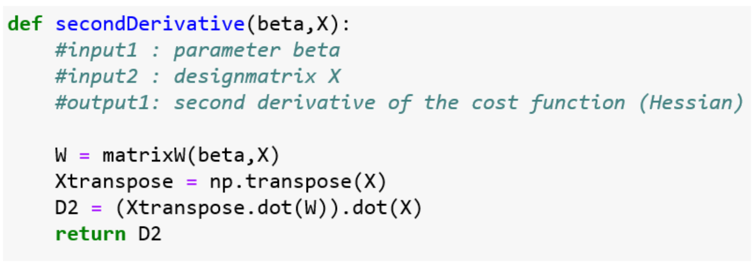
```python
def matrixW(beta,X):
    #input1 : parameter beta
    #input2 : Designmatrix X
    #output1: diagonal matrix p(y=1|beta,X[k,:])*p(y=0|beta,X[k,:]) as kth diagonal element

    rows = X.shape[0]
    columns = X.shape[1]

    #initializing the diagonal
    diagonale = np.zeros([rows])

    #inserting values
    for j in range(0,rows):
        x = X[j,:]
        p = probability(beta,x)
        diagonale[j]= p*(1-p)

    #creating diagonal matrix based on diagonal values
    W=np.diag(diagonale)
    return W
```

The first derivative and second derivative (Hessian) of the cost function look the following way in our setting (lecture slides):

(II) $$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T(\hat{y} - \hat{p})$$ **(III)** $$\frac{\partial^2 \mathcal{C}(\hat{\beta})}{\partial \hat{\beta} \partial \hat{\beta}^T} = \hat{X}^T \hat{W} \hat{X}.$$



```python
def firstDerivative(beta,X,y):
    #input1  : parameter beta
    #input2  : designmatrix X
    #input3  : target values y
    #output1 : first derivative of the cost function

    #generating factor (y-p) as described in lecture slides
    length = y.size
    factor = np.ones([length])
    for k in range(0,length):
        x = X[k,:]
        prob1 = probability(beta,x)
        prob0 = 1 - prob1
        factor[k] = y[k] - (y[k]*prob1 + (1-y[k])*prob0)

    D = -np.transpose(X).dot(factor)
    return D
```

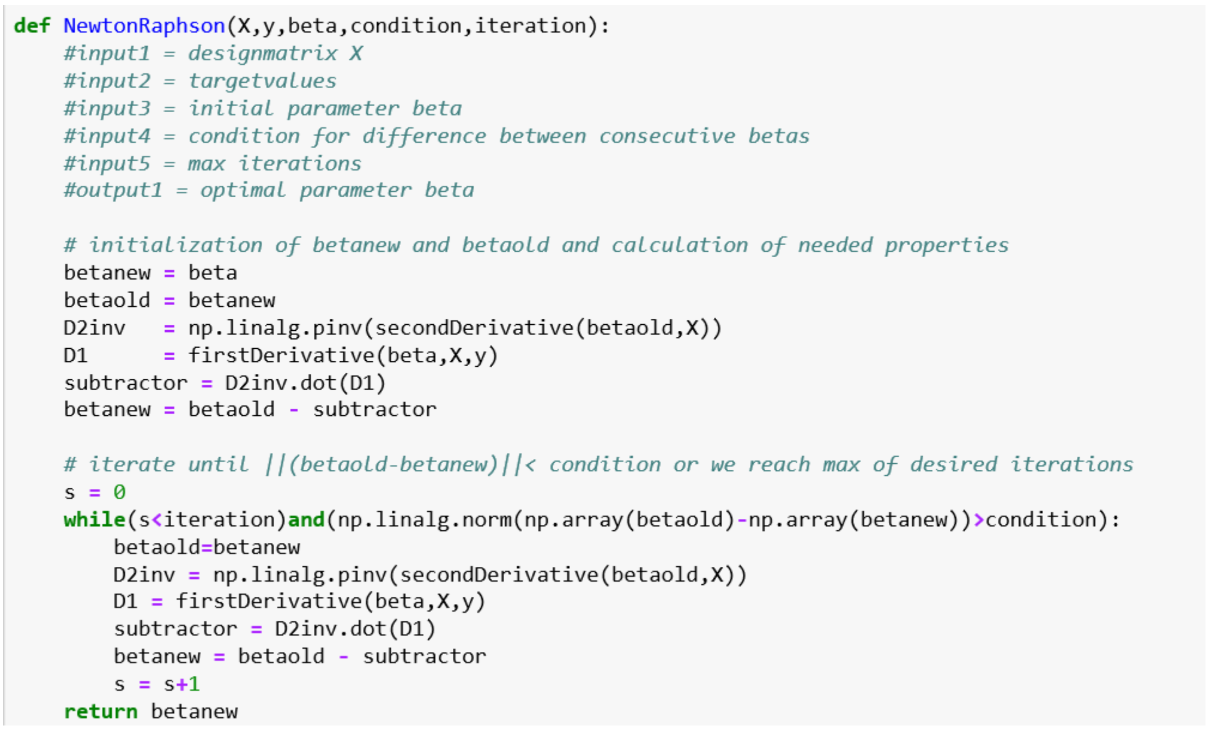


```python
def secondDerivative(beta,X):
    #input1 : parameter beta
    #input2 : designmatrix X
    #output1: second derivative of the cost function (Hessian)

    W = matrixW(beta,X)
    Xtranspose = np.transpose(X)
    D2 = (Xtranspose.dot(W)).dot(X)
    return D2
```

Thanks to the python routines above we are now able to define the Newton-Raphson routine whose iterative scheme looks the following:

(IV) $$\hat{\beta}^{\text{new}} = \hat{\beta}^{\text{old}} - \left(\hat{X}^T \hat{W} \hat{X}\right)^{-1} \times \left(-\hat{X}^T(\hat{y} - \hat{p})\right)_{\hat{\beta}^{\text{old}}}$$

Code for Newton-Raphson method:



```python
def NewtonRaphson(X,y,beta,condition,iteration):
    #input1 = designmatrix X
    #input2 = targetvalues
    #input3 = initial parameter beta
    #input4 = condition for difference between consecutive betas
    #input5 = max iterations
    #output1 = optimal parameter beta

    # initialization of betanew and betaold and calculation of needed properties
    betanew = beta
    betaold = betanew
    D2inv   = np.linalg.pinv(secondDerivative(betaold,X))
    D1      = firstDerivative(beta,X,y)
    subtractor = D2inv.dot(D1)
    betanew = betaold - subtractor

    # iterate until ||(betaold-betanew)||< condition or we reach max of desired iterations
    s = 0
    while(s<iteration)and(np.linalg.norm(np.array(betaold)-np.array(betanew))>condition):
        betaold=betanew
        D2inv = np.linalg.pinv(secondDerivative(betaold,X))
        D1 = firstDerivative(beta,X,y)
        subtractor = D2inv.dot(D1)
        betanew = betaold - subtractor
        s = s+1
    return betanew
```

Equipped with our optimal parameter β we have to have a routine to classify any spin configuration from the test set. To do so, we remind ourselves that if equation **(I)**> 0.5, the spin configuration is more likely to be ordered than disordered and accordingly we generated the following code:

```python
def Classifier(beta,x):
    #input1 : parameter beta
    #input2 : x values (for example from training set)
    #output : predicted class (based on soft classifier)

    p = beta.dot(np.transpose(x))

    #because exp(z)/1+exp(z) is monotonously increasing we get
    # z>0 => exp(z)/1+exp(z) > 0.5
    if (p>0):
        selected_class = 1
        return selected_class
    if(p<=0):
        selected_class = 0
        return selected_class
```

(Remark : one can show that $\frac{e^k}{1+e^k} > \frac{1}{2} \Leftrightarrow k > 0$ *as* $k \rightarrow \frac{e^k}{1+e^k}$ *is monotonous increasing in k)*

Finally, we need a measure to evaluate the performance of our logistic regression. For this purpose, the method of choice is called the accuracy score (Lecture Slides) which counts the number of correctly classified spin configurations and divides it by the total number of spin configurations.

Accuracy $= \frac{\sum_{i=1}^{n} I[ith\ sample\ correctly\ labeled]}{n}$

where *I[]* denotes the indicator function.

Python implementation:

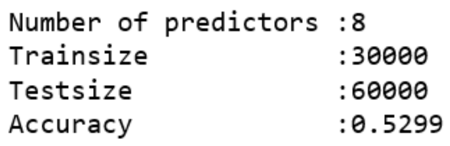
```python
def Accuracy(y,beta,Xtest):
    #input1 : target value
    #input2 : parameter beta
    #input3 : designmatrix of data
    #output1: Accuracy score

    #adding column containing ones to the designmatrix
    X = Xmodified(Xtest)

    rows = X.shape[0]
    columns= X.shape[1]
    length = rows

    #initializing ypredict
    ypredict = np.zeros(length)

    #filling ypredict with the predicted classes
    for j in range(0,length):
        ypredict[j] = Classifier(beta,X[j,:])

    #generating vector correct with : correct[k] = 1 if kth predtiction is correct
    #                                 correct[k] = 0 if kth prediction is incorrect
    correct = 1-(abs(y-ypredict))

    #calculating percentage of correct predictions
    percentage = sum(correct)/rows
    return percentage
```

Training and Analysis of the logistic regression with Newton-Raphson

The basic idea behind the Newton-Raphson method is to extend the tangent line at a current point β_k until it crosses zero and setting the the next guess β_{k+1} to the abscissa of that specific zero-crossing. The procedure of deriving the equations above from this general idea can be found in the lecture slides in more detail (Hjorth-Jensen, 2018). One advantage of this method is the high rate of convergence whereat one has to keep in mind that the initial guess β_{start} can influence dramatically the performance. This can be explained by the fact that the Newton-Raphson method is based on a taylor approximation utilizing only low-order terms. Initializing the algorithm with β_{start} far away from the actual root β, higher-order terms in the Taylor series are significant and we face the risk of being stuck in a local minimum. As a result, we wish for an initial guess β_{start} close to the root β and a convex function at its best as this ensures that a local minimum is also a global minimum. Another aspect concerning the suitability of the Newton-Raphson method is the necessity of computing the inverse of the Hessian (second derivative) of the cost function in every iteration. This is inasmuch of importance as we were facing numerical problems in the form of singular matrices while running the above-shown *NewtonRaphson* routine. To avoid the failure of our implemented algorithm, we had to fall back on calculating the pseudoinverse (using numpy) as no inverse exists for singular matrices. However, due to the fact that numpy makes recourse to SVD decomposition in order to compute the pseudoinverse we had to reduce the number of predictors as the SVD algorithm failed to converge in case of more than 8 predictors.

As a result, only a fraction of the 1600 possible predictors could be used in order to determine the parameter β_{opt} which allows us to classify spin configurations that were not included in the training set.

The result of the analysis exercising our code (using the developed python routines above) looks the following:



```
Number of predictors :8
Trainsize            :30000
Testsize             :60000
Accuracy             :0.5299
```

One can see that the accuracy (52.99%) of correct classified spin configurations is not satisfying as we perform only slightly better than a system guessing randomly. One possible explanation for the poor result is that only 8 out of 1600 spin variables may fail to represent the whole spin configuration properly, so even in case the logistic regression routine operates smoothly we may not be able to predict plausibly based on such a small number of predictors. As we were not in a position to increase the number of predictors in consequence of the above-mentioned convergence problems computing the pseudoinverse, we failed to improve the accuracy score of our logistic regression.

Furthermore, the lack of a regulation mechanism in the iterative Newton-Raphson method for the parameter β may also justify the poor performance as our algorithm is accessible to the risk of overfitting without the implementation of regulators. In order to have the possibility to further analysis of the the binary classification of the 2D Ising model, we refer to the notebook of Mehta et al. (2018).

Instead of the usage of the Newton-Raphson method, both Stochastic Gradient Descent (SGD) and the so-called liblinear routine (default optimization routine based on linear regression in skikit learn) were implemented. As these optimization methods were tested for different regularization parameters λ, one can see that having a well-adjusted λ is a necessary precondition for a high accuracy of correctly predicted spin configurations.

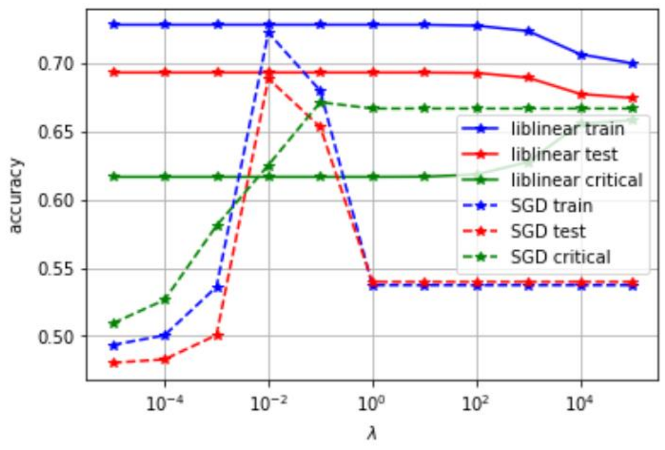


Fig.4: Accuracy of different descent solvers in logistic regression

Especially the course of the SGD graph manifests the importance of the choice of λ as the performance on training data is approximately 50% for extremely small λ (even worse than our implemented Newton-Raphson method without any regulation) while we obtain the best accuracy of approximately 68% for $\lambda = 10^{-2}$.

To summarise this part of the project, we have to highlight the importance of the fine-tuning in the optimization methods used in binary classification problems, as both the Newton-Raphson method and the SGD method perform poorly without a well-adjusted regulation parameter λ, which is inasmuch of relevance as the usability of an algorithm depends significantly on its ability to avoid overfitting.

4.3 Classifying the Ising model phase using neural networks

After having implemented a routine for tackling the binary classification problem with logistic regression, a natural question arising is whether we can improve the accuracy of the classification using neural networks. To get to the bottom of this question, we develop a fully-connected multilayer perceptron (MLP) with backpropagation consisting of three layers (one input layer, one hidden layer, one output layer). In this context, fully-connected means that each node is connected to all nodes in the subsequent layer (Hjorth-Jensen, 2018). In order to develop such a routine, one has to go through six phases, namely collecting and pre-processing data, defining the model and architecture, choosing cost function and optimizer, training the model, evaluating its performance on test data and adjusting hyperparameters.

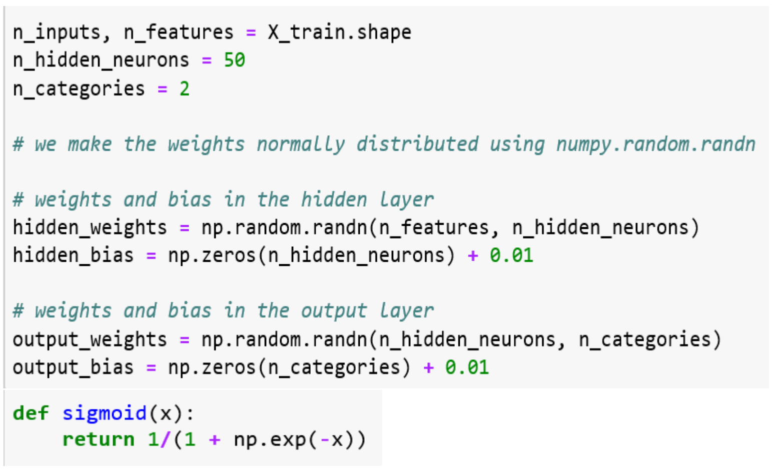
As the collecting and pre-processing of the data has already been exercised during part c) of this project, we start by defining the model and its architecture. As mentioned above, our model will consist of three layers (input layer, hidden layer, output layer) whereat the activation of each neuron is a weighted sum of inputs passed through an activation function. According to the universal approximation theorem an activation function has to be non-constant, bounded, monotonically-increasing and continuous. In practice, the sigmoid and the hyperbolic tangent are popular functions fulfilling these properties (Walia, 2017).

(I) Sigmoid : $f(x) = \frac{1}{1+e^{-x}}$

(II) Hyperbolic Tangent : $f(x) = \tanh(x)$

In general, one should have a bias towards the Hyperbolic Tangent as it is zero-centered while the Sigmoid function may face the problem of updating the gradients too far in different directions as the output is not zero-centered (notice $0 < \frac{1}{1+e^{-x}} < 1$ for all x ε R). Nevertheless, we utilize the Sigmoid function thanks to its nice properties regarding the derivatives.

To get started with the code, we have to evaluate the number of input spin configurations, the number of features which equals the number of spin variables in one configuration and the number of categories. As we tackle a binary classification problem, we only have two categories.



```
n_inputs, n_features = X_train.shape
n_hidden_neurons = 50
n_categories = 2

# we make the weights normally distributed using numpy.random.randn

# weights and bias in the hidden layer
hidden_weights = np.random.randn(n_features, n_hidden_neurons)
hidden_bias = np.zeros(n_hidden_neurons) + 0.01

# weights and bias in the output layer
output_weights = np.random.randn(n_hidden_neurons, n_categories)
output_bias = np.zeros(n_categories) + 0.01
```

```
def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

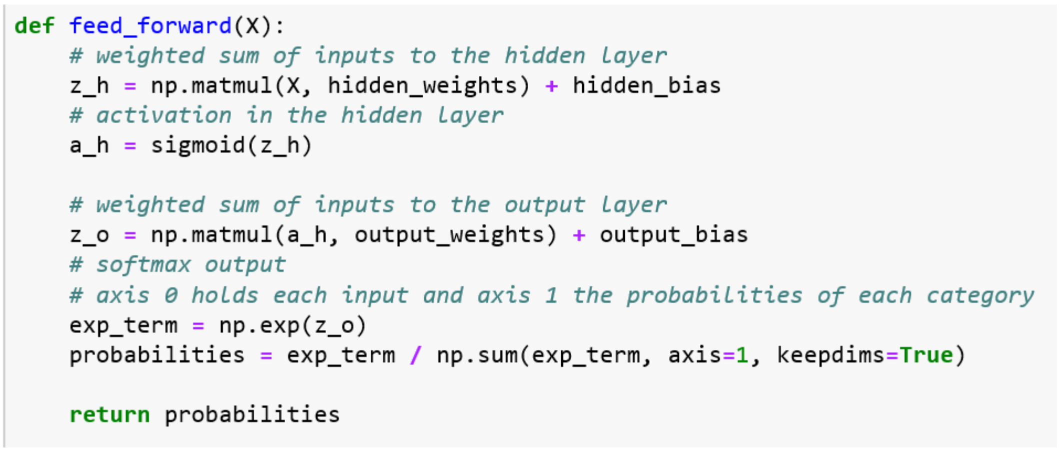
Furthermore, it is necessary to initialize the weights and biases both in the hidden layer and the output layer in order to be able to exercise the feed-forward.

In the feed-forward process, we are supposed to compute a weighted sum of input features (in our case spin variables) to each neuron in the hidden layer, passing it through the sigmoid function (our activation function) and repeating this procedure by computing a weighted sum to each neuron in the output layer with respect to the activated values in the hidden layers.

Finally, the output of neuron k (k ε {0,1}) is calculated by applying the **softmax** function:

$$output(k) = \frac{e^{z(k)}}{\sum_{j=0}^{1} e^{z(j)}}, where\ z(k)\ denotes\ the\ weighted\ sum\ assigned\ to\ output\ neuron\ k\ \varepsilon\ \{0,1\}$$

The realisation of the feed-forward routine looks the following:



```python
def feed_forward(X):
    # weighted sum of inputs to the hidden layer
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    # activation in the hidden layer
    a_h = sigmoid(z_h)

    # weighted sum of inputs to the output layer
    z_o = np.matmul(a_h, output_weights) + output_bias
    # softmax output
    # axis 0 holds each input and axis 1 the probabilities of each category
    exp_term = np.exp(z_o)
    probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)

    return probabilities
```

As the output of neuron k in the output layer can be interpreted as its likelihood of belonging to category 0 or 1, our predicted category is the one with the higher likelihood. The log cross-entropy cost function is used, which we aim to minimize using the backpropagation algorithm (Nielsen, 2015).

The backpropagation algorithm offers for instance the possibility to solve classification problems in a more efficient way and was introduced in the 1970s. It is also known as the “workhorse” (Nielsen, 2015) of learning in neural networks. Comparing the feed-forward output of the network with the desired result, adjusting the weights and biases in the hidden/output layer based on the computation of their gradients and repeating this routine until we obtain a satisfying precision on training data is the basic idea behind the backpropagation. A more detailed derivation of the backpropagation algorithm can be found in the lecture slides (Hjorth-Jensen, 2018). The implementation in python looks the following:

```python
def backpropagation(X, Y):
    a_h, probabilities = feed_forward_train(X)

    # error in the output layer
    error_output = probabilities - Y
    # error in the hidden layer
    error_hidden = np.matmul(error_output, output_weights.T) * a_h * (1 - a_h) #sigma L in lecture

    # gradients for the output layer
    output_weights_gradient = np.matmul(a_h.T, error_output)
    output_bias_gradient = np.sum(error_output, axis=0)

    # gradient for the hidden layer
    hidden_weights_gradient = np.matmul(X.T, error_hidden)
    hidden_bias_gradient = np.sum(error_hidden, axis=0)

    return output_weights_gradient, output_bias_gradient, hidden_weights_gradient, hidden_bias_gradient,error_output
```

Optimizing the learning rate

Training the neural network means optimizing the weights and biases in the hidden layer and output layer such that the cost function is minimized. Widely used is the so-called Gradient Descent method. As we aim to find the minimum of the cost function we adjust the weights in the direction where the gradient of the cost function is large and negative. This ensures that we advance towards a local minimum.

In mathematical terms, the Gradient Descent iterative routine for each parameter Θ can be expressed as:

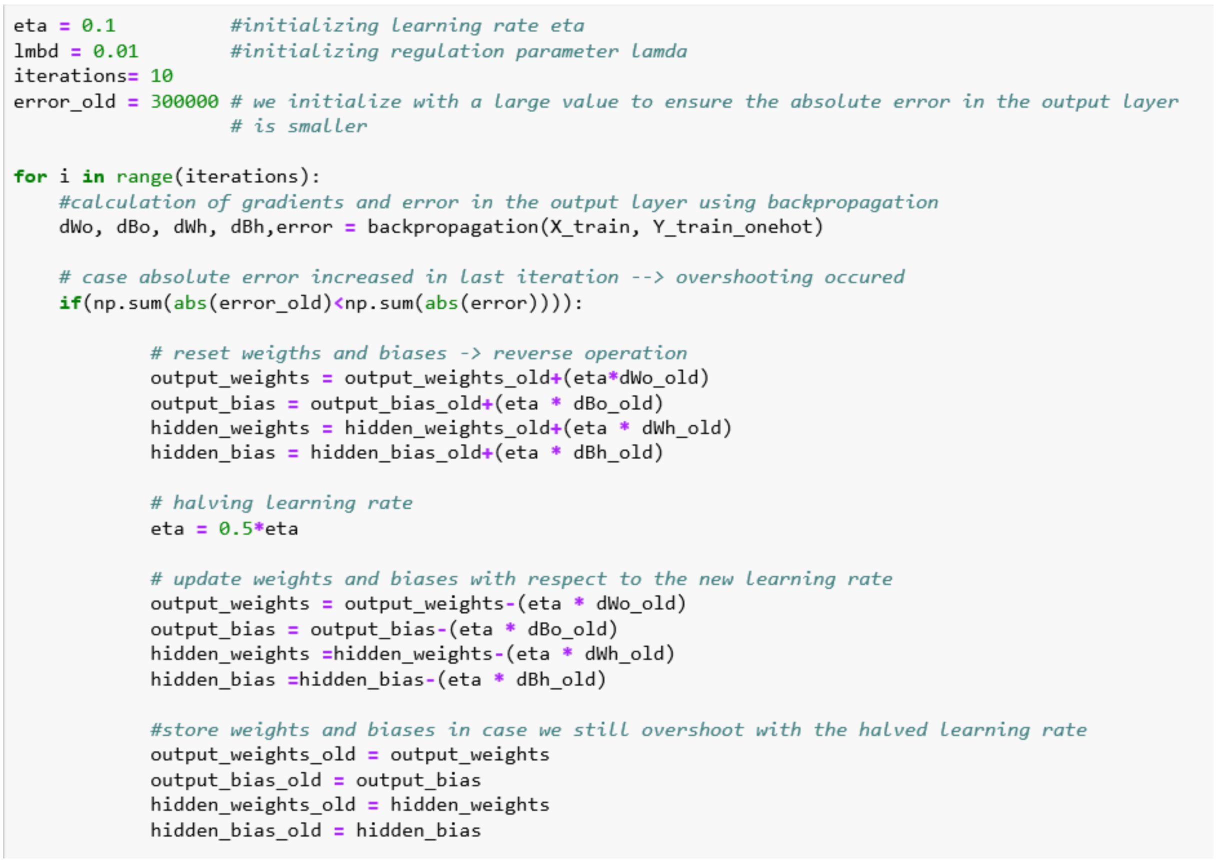
$$\Theta_{k+1} = \Theta_k - \eta \nabla C(\Theta_k) \,,$$

where η denotes the learning rate and $\nabla C(\Theta_k)$ the cost function's gradient with respect to Θ_k.

In practice, the choice of the learning rate η is crucial regarding the performance of the neural network as it tells the optimizer how far to move the weights in the direction opposite to the gradient. A low learning rate results in a more reliable training but can lead to long runtime as steps towards the minimum of the cost function are small. On the other, a high learning rate poses the risk that the training may not converge as changes in the weights might get so big that the optimizer overshoots the minimum which conceivably leads to larger losses in the cost function (Surmenok, 2017).

In order to evaluate a satisfying learning rate, we decide to make use of the so-called “Bold-Driver” approach, a technique changing η in each iteration. The underlying idea is that the farther one is away from the optimum the faster one should move towards the solution which necessitates a larger value η. Contrary, the closer we get to the optimum the smaller our value η should be in order to avoid overshooting the minimum. As the optimum is unknown it is impossible to determine how far away we are in each iteration.

To resolve this, we sum up the absolute errors of each parameter in every iteration based on the predicted parameters. In case the absolute error was reduced since the last iteration we raise the learning rate η by 5% as we wish to speed up the convergence process. If the absolute error increases, we can deduce that we overshot the optimum. As a consequence, we reset the updates from the last iteration, decrease the learning rate η by 50% and check if the halved learning rate prevents us from overshooting the optimum. The implementation in python looks the following:



```python
eta = 0.1           #initializing learning rate eta
lmbd = 0.01         #initializing regulation parameter lamda
iterations= 10
error_old = 300000 # we initialize with a large value to ensure the absolute error in the output layer
                   # is smaller

for i in range(iterations):
    #calculation of gradients and error in the output layer using backpropagation
    dWo, dBo, dWh, dBh,error = backpropagation(X_train, Y_train_onehot)

    # case absolute error increased in last iteration --> overshooting occured
    if(np.sum(abs(error_old)<np.sum(abs(error)))):

            # reset weigths and biases -> reverse operation
            output_weights = output_weights_old+(eta*dWo_old)
            output_bias = output_bias_old+(eta * dBo_old)
            hidden_weights = hidden_weights_old+(eta * dWh_old)
            hidden_bias = hidden_bias_old+(eta * dBh_old)

            # halving learning rate
            eta = 0.5*eta

            # update weights and biases with respect to the new learning rate
            output_weights = output_weights-(eta * dWo_old)
            output_bias = output_bias-(eta * dBo_old)
            hidden_weights =hidden_weights-(eta * dWh_old)
            hidden_bias =hidden_bias-(eta * dBh_old)

            #store weights and biases in case we still overshoot with the halved learning rate
            output_weights_old = output_weights
            output_bias_old = output_bias
            hidden_weights_old = hidden_weights
            hidden_bias_old = hidden_bias
```

```
# case absolute error decreased in last iteration
# initialization must ensure that else part of if-statement is exercised in first loop
else:
        #raising eta
        eta = 1.05*eta

         # regularization term gradients
        dWo  = dWo + lmbd * output_weights
        dWh  = dWh + lmbd * hidden_weights

        # storing weights and biases and gradients in case we overshoot
        output_weights_old = output_weights
        output_bias_old = output_bias
        hidden_weights_old = hidden_weights
        hidden_bias_old = hidden_bias
        dWo_old =dWo
        dBo_old =dBo
        dWh_old =dWh
        dBh_old =dBh

        # update weights and biases
        output_weights = output_weights-(eta * dWo)
        output_bias = output_bias-(eta * dBo)
        hidden_weights =hidden_weights-(eta * dWh)
        hidden_bias =hidden_bias-(eta * dBh)

        #update error
        error_old = error
```

Unfortunately, our developed neural network performs poorly even on training data as one can see in the following result we obtained running the above-mentioned python routines:

```
Old accuracy on training data: 0.3618769230769231
New accuracy on training data: 0.4624923076923077
```

This means that our MLP is basically guessing and does not seem to learn at all which leads to the conclusion that the hyperparameters were not adjusted properly. One possible approach is to make use of minibatches, that is we do not take the whole available data set but only a subset on which we calculate an approximation of the gradients. As this method introduces stochasticity by picking minibatches randomly, we reduce the risk of being stuck in a local minimum in case our cost function is not convex. Furthermore, minibatching speeds up the computation as calculating gradients based on a subset is cheaper than taking the whole dataset into account. Another way of improving the performance of the neural network is to try different regularization parameters λ. The basic idea behind implementing regularisation mechanisms is that we do not wish for out-of-control-growing weights as this bears the risk of overfitting.

Due to the fact that our own neural network does not work properly on the training data it is unsuitable to serve as a basis for further analysis as we suspect that its malfunctioning cannot only be explained by the lack of minibatches and a possible non-optimal regularisation parameter λ.

As a consequence, we refer to the notebook of Mehta et al to get to the bottom of the question whether neural networks should be the considered analysis method in logistic regression problems.

In contrast to our neural network, the above-mentioned notebook makes use of minibatches and epochs, Stochastic Gradient Descent (SGD) instead of our more practical "Bold Driver" approach and varies different learning rates and number of neurons.

Performance on training data:

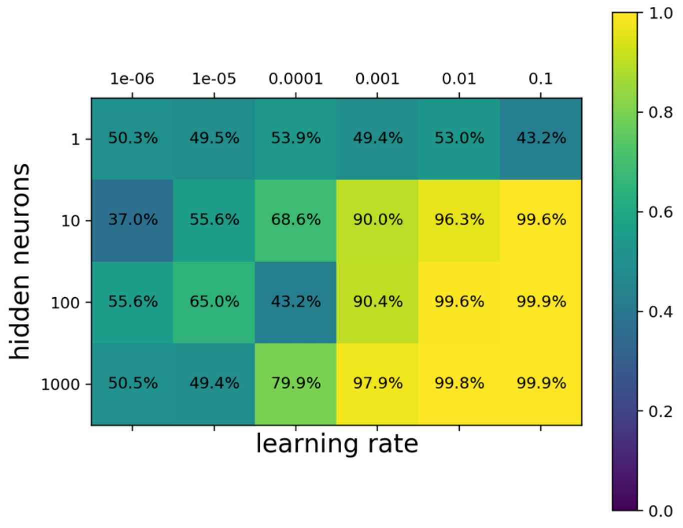


Fig.5: Performance map of neural network depending on number of neurons and learning rate (training data)

Performance on test data:

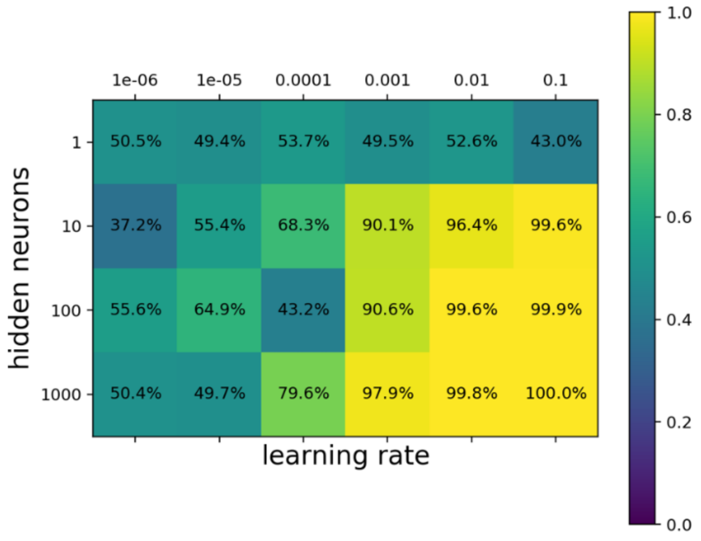


Fig.5: Performance map of neural network depending on number of neurons and learning rate (test data)

These figures illustrate brilliantly the chances and risks using neural networks for binary classification problems. One can see that an excellent accuracy is obtained in case the neural network consists of 100-1000 neurons and the learning rate varies between 0.01 and 0.1. However, a large number of hidden neurons fails to compensate a tiny learning rate, which stresses the complexity of creating a well-functioning neural network as one poorly chosen hyperparameter (for instance the learning rate or the regularisation parameter) can entail a network with bad performance.

Impressive is the fact that a neural network with well-adjusted hyperparameters has the potential to perform the classification of the Ising model almost perfectly, and this not only on the training data but also on test data.

5 Conclusion and Critical Evaluation

In the conclusion of the project, we are going to summarize the implemented algorithms and discuss its pros and cons in regards to their suitability in regression and classification cases.

It was found that the Ridge regression and OLS overfit the data when estimating the coupling constant of the one-dimensional Ising model. Only the Lasso method led to an accurate estimation when the penalization parameter was chosen in a specific range close to 10^{-2}. Furthermore, the models using OLS and Ridge led to a symmetric distribution of coupling strength between nearest neighbors, while only the Lasso regression model was able to break this symmetry and reflect the mode of how the data was generated. This, similarly as in project 1, reveals that the penalization of the regression coefficients' size requires fine tuning and is essential to prevent the model from growing to complex and overfitting.

For the classification problem of the project, we used logistic regression to train our model predicting the phase of the Ising model given its spin configuration with the help of the Newton-Raphson method. Its advantage is the high rate of convergence and its easy implementation. On the other hand, one has to keep in mind that the method requires the computation of the inverse of the Hessian matrix in every iteration which results in expensive calculations and one has to make use of computing the pseudoinverse in case the Hessian matrix is singular. As a result, the Newton-Raphson algorithm is suitable for small datasets with a somehow small number of predictors. Using Stochastic Gradient Descent in the logistic regression is another way of tackling the binary classification and resulted in a more satisfying result than the analysis based on the Newton-Raphson algorithm as we did not face the problem of handling with singular matrices in this case. One has to keep in mind that for both the Stochastic Gradient Descent and the Newton-Raphson approach the regularisation of the parameters is crucial for the performance of the model as unrestricted large parameters bear the risk of overfitting the training data which leads to bad performance on test data. To sum up, the suitability of Newton-Raphson depends highly on the number of datapoints and one should consider this method only in case it is likely not getting into trouble with too large Hessian matrices or even singular matrices. As we saw in part c) of the project, Stochastic Gradient Descent with a well-adjusted regularisation parameter λ may be the method of choice in case we face the above-mentioned problems with the Newton-Raphson algorithm.

As our best accuracy score on test data in the logistic regression case with Stochastic Gradient Descent was approximately 70%, one can deduce that neural networks, performing the binary classification with the backpropagation algorithm, do a better job as we obtained an impressive close-to-100% accuracy score on test data for a neural network consisting of 1000 neurons and a learning rate of 0.1. Acknowledging its immense potential, one has to keep in mind that creating a well-performing neural network requires not only a sufficiently large number of neurons, but also a wisely chosen learning rate, regularisation parameters to avoid overfitting, optimal activation and cost functions and at its best a mechanism to reduce the whole dataset to a representative subset to introduce stochasticity and to cheapen the calculation cost. We have to acknowledge painfully that it is a complex task to compose all these fragments to a well-functioning neural network as our developed MLP, based on the “Bold Driver” approach described in part d) of the project, basically guesses whether a spin configuration is ordered or disordered after training. The fact that the neural network algorithm provided by the notebook of Mehta et al also performs poorly for a sufficiently small learning rate highlights the necessity of adjusting all hyperparameters smoothly in order to generate a well-performing network. In practice, this means one should consider tackling a binary classification problem using neural networks to ensure a high accuracy score in case it is possible to carry out tests on different activation functions, different regularisation parameters λ and on different learning rates η as this is the precondition for building up a promising neural network.

6 Appendices

Appendix 1. Repetition of theory behind regression methods OLS, Ridge and Lasso

The least squares approach assumes a model that is linear in the parameters. From the training data, the regression parameters β are derived. The calculation of the parameters is based on minimizing a chosen cost function. In the case of OLS, β should minimize the following residual sum of squares (Hastie, Tibshirani, & Friedman, 2009, p. 44) :

$$\begin{aligned}\mathrm{RSS}(\beta) &= \sum_{i=1}^{N}(y_i - f(x_i))^2 \\ &= \sum_{i=1}^{N}\Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\Big)^2\end{aligned}$$

Here (x_I , y_i) for i = 1,…,N represents the training data and f the linear model.

The Ridge Regression uses a different cost function. It is a shrinkage method, preventing the model from growing to complex. According to Hastie et al. (2009, p. 63), “the ridge coefficients minimize a penalized residual sum of squares” :

$$\hat{\beta}^{\text{ridge}} = \underset{\beta}{\operatorname{argmin}}\left\{\sum_{i=1}^{N}\Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\Big)^2 + \lambda\sum_{j=1}^{p}\beta_j^2\right\}$$

In the Ridge Regression, λ is a non-negative parameter, controlling the complexity of the model by imposing a penalty on the size of the coefficients.

The Lasso method is similar to the Ridge Regression. The problem for finding the coefficients in this method is as follows (Hastie, Tibshirani, & Friedman, 2009, p. 68) :

$$\hat{\beta}^{\text{lasso}} = \underset{\beta}{\operatorname{argmin}}\left\{\frac{1}{2}\sum_{i=1}^{N}\Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\Big)^2 + \lambda\sum_{j=1}^{p}|\beta_j|\right\}$$

As this is not easy to solve, the computation was performed with the help of the *scikit-learn* python package, which uses a coordinated descent algorithm.

7 References

Cipra, B. (1987). *An Introduction to the Ising Model.*

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning - Second Edition.* Springer.

Hjorth-Jensen, M. (2018, 10). *Overview of course material: Data Analysis and Machine Learning*. Retrieved from Department of Physics (office FV308), University of Oslo, Norway: https://compphysics.github.io/MachineLearning/doc/web/course.html

Le, J. (2018, March 19). *A Gentle Introduction to Neural Networks for Machine Learning.* Retrieved from Codementor: https://www.codementor.io/james_aka_yale/a-gentle-introduction-to-neural-networks-for-machine-learning-hkijvz7lp

Mehta, P., Wang, C.-H., Day, A. G., & Richardson, C. (2018). A high-bias, low-variance introduction to Machine Learning for physicists. *Department of Physics, Boston University*.

Nielsen, M. (2015). *Neural Networks and Deep Learning.* Determination Press.

Rumelhart, D., Hinton, G., & Williams, R. (09. October 1986). Learning representations by back-propagating errors. *nature - international journal of science 323*, S. 533–536.

Surmenok, P. (2017, November 13). *Estimating an Optimal Learning Rate For a Deep Neural Network*. Retrieved from https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0

Walia, A. (2017, May 29). *Activation functions and it's types-Which is better?* Retrieved from https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f