UiO : Universitetet i Oslo

Logistic Regression and Neural Network

Logistic Regression and Neural Network

Author:
Ana Costa
authors@email.com

Supervisors:

UiO

November 13, 2018

Contents

List of Figures

List of Tables

List of Codes

https://github.com/anacost/project2-FYS-STK4155

1 General structure

- Project title
- Name, email, course title, date, group assistant
- Abstract (1/2 page max)
- Introduction (1 page)
- Method
  - Packages used
  - Datasets (models and observations)
  - Analysis method
  - ...
- Results
- Discussion and outlook (1 page)
- Conclusions (1/2 page)
- References
- Acknowledgments

```python
import tensorflow
import sklearn
import pandas
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import seaborn
%matplotlib inline
import numpy as np
```

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

```python
import scipy.sparse as sp
np.random.seed(12)
import warnings
#Comment this to turn on warnings
warnings.filterwarnings('ignore')

### 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)
```

```
# reshape Ising states into RL samples: S_iS_j --> X_p
states=np.einsum('...i,...j->...ij', states, states)
shape=states.shape
states=states.reshape((shape[0],shape[1]*shape[2]))
# build final data set
Data=[states,energies]
```

```
# define number of samples
n_samples=400
# define train and test data sets
X_train=Data[0][:n_samples]
Y_train=Data[1][:n_samples] #+ np.random.normal(0,4.0,size=X_train.shape[0])
X_test=Data[0][n_samples:3*n_samples//2]
Y_test=Data[1][n_samples:3*n_samples//2] #+ np.random.normal(0,4.0,size=X_test.shape[0])
```

1.2 Part b) Estimating the coupling constant of the one-dimensional Ising model

Code 1.1: Code example

```
from sklearn import linear_model
# define error lists
train_errors_leastsq = []
test_errors_leastsq = []
train_MSE_leastsq = []
test_MSE_leastsq = []
train_bias_leastsq = []
test_bias_leastsq = []
train_var_leastsq = []
test_var_leastsq = []

train_errors_ridge = []
test_errors_ridge = []
train_MSE_ridge = []
test_MSE_ridge = []
train_bias_ridge = []
test_bias_ridge = []
train_var_ridge = []
test_var_ridge = []

train_errors_lasso = []
test_errors_lasso = []
train_MSE_lasso = []
test_MSE_lasso = []
train_bias_lasso = []
test_bias_lasso = []
train_var_lasso = []
test_var_lasso = []

# set regularisation strength values
lmbdas = np.logspace(-4, 5, 10)

#Initialize coeffficients for OLS, ridge regression and Lasso
coefs_leastsq = []
coefs_ridge = []
coefs_lasso=[]
# set up Lasso Regression model
lasso = linear_model.Lasso()

for _,lmbda in enumerate(lmbdas):
    ### ordinary least squares
    xb = np.c_[np.ones((X_train.shape[0],1)),X_train]
    #fit model/singularity :
    beta_ols = np.linalg.pinv(xb.T @ xb) @ xb.T @ Y_train
    coefs_leastsq.append(beta_ols) # store weights

    # use the coefficient of determination R^2 as the performance of prediction.
    fitted_train = xb  @ beta_ols
    xb_test = np.c_[np.ones((X_test.shape[0],1)),X_test]

    fitted_test = xb_test @ beta_ols
    R2_train = 1 - np.sum( (fitted_train - Y_train)**2 )/np.sum( (Y_train - np.mean(Y_train))**2 )
    R2_test = 1 - np.sum( (fitted_test - Y_test)**2 )/np.sum((Y_test - np.mean(Y_test))**2)
    MSE_train = np.sum((fitted_train - Y_train)**2)/len(Y_train)
    MSE_test = np.sum((fitted_test - Y_test)**2)/len(Y_test)
    var_train = np.sum((fitted_train - np.mean(fitted_train))**2)/len(Y_train)
```

Code 1.2: Performance

```
# Plot our performance on both the training and test data
plt.semilogx(lmbdas, train_errors_leastsq, 'b',label='Train (OLS)')
plt.semilogx(lmbdas, test_errors_leastsq,'--b',label='Test (OLS)')
plt.semilogx(lmbdas, train_errors_ridge,'r',label='Train (Ridge)', linewidth=1)
plt.semilogx(lmbdas, test_errors_ridge,'--r',label='Test (Ridge)', linewidth=1)
plt.semilogx(lmbdas, train_errors_lasso, 'g',label='Train (LASSO)')
plt.semilogx(lmbdas, test_errors_lasso, '--g',label='Test (LASSO)')

fig = plt.gcf()
fig.set_size_inches(10.0, 6.0)

#plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
#           linewidth=3, label='Optimum on test')
plt.legend(loc='lower left',fontsize=16)
plt.ylim([-0.01, 1.01])
plt.xlim([min(lmbdas), max(lmbdas)])
plt.xlabel(r'$\lambda$',fontsize=16)
plt.ylabel('Performance',fontsize=16)
plt.tick_params(labelsize=16)
plt.show()
```

1.3 Understanding the results

Let us make a few remarks: (i) the (inverse, see Scikit documentation) regularization parameter λ affects the Ridge and LASSO regressions at scales, separated by a few orders of magnitude. Notice that this is different for the data considered in Notebook 3 **Section VI: Linear Regression (Diabetes)**. Therefore, it is considered good practice to always check the performance for the given model and data with λ. (ii) at $\lambda \to 0$ and $\lambda \to \infty$, all three models overfit the data, as can be seen from the deviation of the test errors from unity (dashed lines), while the training curves stay at unity. (iii) While the OLS and Ridge regression test curves are monotonic, the LASSO test curve is not -- suggesting the optimal LASSO regularization parameter is $\lambda \approx 10^{-2}$. At this sweet spot, the Ising interaction weights $\mathbf{J}$ contain only nearest-neighbor terms (as did the model the data was generated from).

Gauge degrees of freedom: recall that the uniform nearest-neighbor interactions strength $J_{j,k} = J$ which we used to generate the data was set to unity, $J = 1$. Moreover, $J_{j,k}$ was NOT defined to be symmetric (we only used the $J_{j,j+1}$ but never the $J_{j,j-1}$ elements). The colorbar on the matrix elements plot above suggest that the OLS and Ridge regression learn uniform symmetric weights $J = -0.5$. There is no mystery since this amounts to taking into account both the $J_{j,j+1}$ and the $J_{j,j-1}$ terms, and the weights are distributed symmetrically between them. LASSO, on the other hand, can break this symmetry (see matrix elements plots for $\lambda = 0.001$ and $\lambda = 0.01$). Thus, we see how different regularization schemes can lead to learning equivalent models but in different gauges. Any information we have about the symmetry of the unknown model that generated the data has to be reflected in the definition of the model and the regularization chosen.

Code 1.3: Performance MSE

```
# Plot our performance on both the training and test data
plt.semilogx(lmbdas, train_MSE_leastsq, 'b',label='Train (OLS)')
plt.semilogx(lmbdas, test_MSE_leastsq,'--b',label='Test (OLS)')
plt.semilogx(lmbdas, train_MSE_ridge,'r',label='Train (Ridge)', linewidth=1)
plt.semilogx(lmbdas, test_MSE_ridge,'--r',label='Test (Ridge)', linewidth=1)
plt.semilogx(lmbdas, train_MSE_lasso, 'g',label='Train (LASSO)')
plt.semilogx(lmbdas, test_MSE_lasso, '--g',label='Test (LASSO)')

fig = plt.gcf()
fig.set_size_inches(10.0, 6.0)

#plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
#            linewidth=3, label='Optimum on test')
plt.legend(loc='lower left',fontsize=16)
plt.ylim([-0.01, 1.01])
plt.xlim([min(lmbdas), max(lmbdas)])
plt.xlabel(r'$\lambda$',fontsize=16)
plt.ylabel('Performance-MSE',fontsize=16)
plt.tick_params(labelsize=16)
plt.show()
```

Code 1.4: Performance bias/variance

```python
# Plot our bias-variance on both the training and test data
plt.semilogx(lmbdas, train_bias_leastsq, 'b',label='Bias-Train (OLS)')
plt.semilogx(lmbdas, test_bias_leastsq,'--b',label='Bias-Test (OLS)')
plt.semilogx(lmbdas, train_bias_ridge,'r',label='Bias-Train (Ridge)',linewidth=1)
plt.semilogx(lmbdas, test_bias_ridge,'--r',label='Bias-Test (Ridge)',linewidth=1)
plt.semilogx(lmbdas, train_bias_lasso, 'g',label='Bias-Train (LASSO)')
plt.semilogx(lmbdas, test_bias_lasso, '--g',label='Bias-Test (LASSO)')

plt.semilogx(lmbdas, train_var_leastsq, ':b',label='Variance-Train (OLS)')
plt.semilogx(lmbdas, test_var_leastsq,'.b',label='Variance-Test (OLS)')
plt.semilogx(lmbdas, train_var_ridge,':r',label='Variance-Train (Ridge)',linewidth=1)
plt.semilogx(lmbdas, test_var_ridge,'.r',label='Variance-Test (Ridge)',linewidth=1)
plt.semilogx(lmbdas, train_var_lasso, ':g',label='Variance-Train (LASSO)')
plt.semilogx(lmbdas, test_var_lasso, '.g',label='Variance-Test (LASSO)')

fig = plt.gcf()
fig.set_size_inches(10.0, 6.0)

#plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
#           linewidth=3, label='Optimum on test')
plt.legend(loc='lower left',fontsize=16)
#plt.ylim([-0.01, 1.01])
plt.xlim([min(lmbdas), max(lmbdas)])
plt.xlabel(r'$\lambda$',fontsize=16)
plt.ylabel('Bias-Variance',fontsize=16)
plt.tick_params(labelsize=16)
plt.show()
```

Code 1.5: Performance OlS bias/variance

```
# Plot our bias-variance on both the training and test data
plt.semilogx(lmbdas, train_bias_leastsq, 'b',label='Bias-Train (OLS)')
plt.semilogx(lmbdas, test_bias_leastsq,'--b',label='Bias-Test (OLS)')
#plt.semilogx(lmbdas, train_bias_ridge,'r',label='Bias-Train (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, test_bias_ridge,'--r',label='Bias-Test (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, train_bias_lasso, 'g',label='Bias-Train (LASSO)')
#plt.semilogx(lmbdas, test_bias_lasso, '--g',label='Bias-Test (LASSO)')

plt.semilogx(lmbdas, train_var_leastsq, ':b',label='Variance-Train (OLS)')
plt.semilogx(lmbdas, test_var_leastsq,'.b',label='Variance-Test (OLS)')
#plt.semilogx(lmbdas, train_var_ridge,':r',label='Variance-Train (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, test_var_ridge,'.r',label='Variance-Test (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, train_var_lasso, ':g',label='Variance-Train (LASSO)')
#plt.semilogx(lmbdas, test_var_lasso, '.g',label='Variance-Test (LASSO)')

fig = plt.gcf()
fig.set_size_inches(10.0, 6.0)

#plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
#           linewidth=3, label='Optimum on test')
plt.legend(loc='lower left',fontsize=16)
#plt.ylim([-0.01, 1.01])
plt.xlim([min(lmbdas), max(lmbdas)])
plt.xlabel(r'$\lambda$',fontsize=16)
plt.ylabel('Bias-Variance',fontsize=16)
plt.tick_params(labelsize=16)
plt.show()
```

Code 1.6: Performance Ridge bias/variance

```python
# Plot our bias-variance on both the training and test data
#plt.semilogx(lmbdas, train_bias_leastsq, 'b',label='Bias-Train (OLS)')
#plt.semilogx(lmbdas, test_bias_leastsq,'--b',label='Bias-Test (OLS)')
plt.semilogx(lmbdas, train_bias_ridge,'r',label='Bias-Train (Ridge)',linewidth=1)
plt.semilogx(lmbdas, test_bias_ridge,'--r',label='Bias-Test (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, train_bias_lasso, 'g',label='Bias-Train (LASSO)')
#plt.semilogx(lmbdas, test_bias_lasso, '--g',label='Bias-Test (LASSO)')

#plt.semilogx(lmbdas, train_var_leastsq, ':b',label='Variance-Train (OLS)')
#plt.semilogx(lmbdas, test_var_leastsq,'.b',label='Variance-Test (OLS)')
plt.semilogx(lmbdas, train_var_ridge,':r',label='Variance-Train (Ridge)',linewidth=1)
plt.semilogx(lmbdas, test_var_ridge,'.r',label='Variance-Test (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, train_var_lasso, ':g',label='Variance-Train (LASSO)')
#plt.semilogx(lmbdas, test_var_lasso, '.g',label='Variance-Test (LASSO)')

fig = plt.gcf()
fig.set_size_inches(10.0, 6.0)

#plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
#           linewidth=3, label='Optimum on test')
plt.legend(loc='lower left',fontsize=16)
#plt.ylim([-0.01, 1.01])
plt.xlim([min(lmbdas), max(lmbdas)])
plt.xlabel(r'$\lambda$',fontsize=16)
plt.ylabel('Bias-Variance',fontsize=16)
plt.tick_params(labelsize=16)
plt.show()
```

Code 1.7: Performance Lasso bias/variance

```python
# Plot our bias-variance on both the training and test data
#plt.semilogx(lmbdas, train_bias_leastsq, 'b',label='Bias-Train (OLS)')
#plt.semilogx(lmbdas, test_bias_leastsq,'--b',label='Bias-Test (OLS)')
#plt.semilogx(lmbdas, train_bias_ridge,'r',label='Bias-Train (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, test_bias_ridge,'--r',label='Bias-Test (Ridge)',linewidth=1)
plt.semilogx(lmbdas, train_bias_lasso, 'g',label='Bias-Train (LASSO)')
plt.semilogx(lmbdas, test_bias_lasso, '--g',label='Bias-Test (LASSO)')

#plt.semilogx(lmbdas, train_var_leastsq, ':b',label='Variance-Train (OLS)')
#plt.semilogx(lmbdas, test_var_leastsq,'.b',label='Variance-Test (OLS)')
#plt.semilogx(lmbdas, train_var_ridge,':r',label='Variance-Train (Ridge)',linewidth=1)
#plt.semilogx(lmbdas, test_var_ridge,'.r',label='Variance-Test (Ridge)',linewidth=1)
plt.semilogx(lmbdas, train_var_lasso, ':g',label='Variance-Train (LASSO)')
plt.semilogx(lmbdas, test_var_lasso, '.g',label='Variance-Test (LASSO)')

fig = plt.gcf()
fig.set_size_inches(10.0, 6.0)

#plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
#           linewidth=3, label='Optimum on test')
plt.legend(loc='lower left',fontsize=16)
#plt.ylim([-0.01, 1.01])
plt.xlim([min(lmbdas), max(lmbdas)])
plt.xlabel(r'$\lambda$',fontsize=16)
plt.ylabel('Bias-Variance',fontsize=16)
plt.tick_params(labelsize=16)
plt.show()
```

```python
#bootstrap:
```

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

```
np.random.seed(1) # shuffle random seed generator

# Ising model parameters
L=40 # linear system size
J=-1.0 # Ising interaction
T=np.linspace(0.25,4.0,16) # set of temperatures
T_c=2.26 # Onsager critical temperature in the TD limit
```

```python
##### prepare training and test data sets
import pickle,os
from sklearn.model_selection import train_test_split

###### define ML parameters
num_classes=2
train_to_test_ratio=0.5 # training samples

# path to data directory
path_to_data=os.path.expanduser('.')+'/data/'

# load data
file_name = "Ising2DFM_reSample_L40_T=All.pkl" # this file contains 16*10000 samples taken in T=np.arange(0.25,4.0001,0.25)
data = pickle.load(open(path_to_data+file_name,'rb')) # pickle reads the file and returns the Python object (1D array, compressed bits)
data = np.unpackbits(data).reshape(-1, 1600) # Decompress array and reshape for convenience
data=data.astype('int')
data[np.where(data==0)]=-1 # map 0 state to -1 (Ising variable can take values +/-1)

file_name = "Ising2DFM_reSample_L40_T=All_labels.pkl" # this file contains 16*10000 samples taken in T=np.arange(0.25,4.0001,0.25)
labels = pickle.load(open(path_to_data+file_name,'rb')) # pickle reads the file and returns the Python object (here just a 1D array with the binary labels)

# 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:]

#X_ordered[np.where(X_ordered==0)]=-1 # map 0 state to -1 (Ising variable can take values +/-1)
#X_critical[np.where(X_critical==0)]=-1 # map 0 state to -1 (Ising variable can take values +/-1)
#X_disordered[np.where(X_disordered==0)]=-1 # map 0 state to -1 (Ising variable can take values +/-1)
del data,labels

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

# full data set
X=np.concatenate((X_critical,X))
Y=np.concatenate((Y_critical,Y))

print('X_train shape:', X_train.shape)
print('Y_train shape:', Y_train.shape)
print()
```

```
X_train shape: (65000, 1600)
Y_train shape: (65000,)

65000 train samples
30000 critical samples
65000 test samples
```

Code 1.8: Plotting some states

```
##### plot a few Ising states

# set colourbar map
cmap_args=dict(cmap='plasma_r')

# plot states
fig, axarr = plt.subplots(nrows=1, ncols=3)

axarr[0].imshow(X_ordered[20001].reshape(L,L),**cmap_args)
axarr[0].set_title('$\\mathrm{ordered\\ phase}$',fontsize=16)
axarr[0].tick_params(labelsize=16)

axarr[1].imshow(X_critical[10001].reshape(L,L),**cmap_args)
axarr[1].set_title('$\\mathrm{critical\\ region}$',fontsize=16)
axarr[1].tick_params(labelsize=16)

im=axarr[2].imshow(X_disordered[50001].reshape(L,L),**cmap_args)
axarr[2].set_title('$\\mathrm{disordered\\ phase}$',fontsize=16)
axarr[2].tick_params(labelsize=16)

fig.subplots_adjust(right=2.0)

plt.show()
```

Code 1.9: Accuracy

```
###### apply logistic regression
from sklearn import linear_model
from sklearn.neural_network import MLPClassifier

# define regularisation parameter
lmbdas=np.logspace(-5,5,11)

# preallocate data
train_accuracy=np.zeros(lmbdas.shape,np.float64)
test_accuracy=np.zeros(lmbdas.shape,np.float64)
critical_accuracy=np.zeros(lmbdas.shape,np.float64)

train_accuracy_SGD=np.zeros(lmbdas.shape,np.float64)
test_accuracy_SGD=np.zeros(lmbdas.shape,np.float64)
critical_accuracy_SGD=np.zeros(lmbdas.shape,np.float64)

# loop over regularisation strength
for i,lmbda in enumerate(lmbdas):

    # define logistic regressor
    logreg=linear_model.LogisticRegression(C=1.0/lmbda,random_state=1,verbose=0,max_iter=1E3,tol=1E-5)

    # fit training data
    logreg.fit(X_train, Y_train)

    # check accuracy
    train_accuracy[i]=logreg.score(X_train,Y_train)
    test_accuracy[i]=logreg.score(X_test,Y_test)
    critical_accuracy[i]=logreg.score(X_critical,Y_critical)

    print('accuracy: train, test, critical')
    print('liblin: %0.4f, %0.4f, %0.4f' %(train_accuracy[i],test_accuracy[i],critical_accuracy[i]) )

    # define SGD-based logistic regression
    logreg_SGD = linear_model.SGDClassifier(loss='log', penalty='l2', alpha=lmbda, max_iter=100,
                                           shuffle=True, random_state=1, learning_rate='optimal')

    # fit training data
    logreg_SGD.fit(X_train,Y_train)

    # check accuracy
    train_accuracy_SGD[i]=logreg_SGD.score(X_train,Y_train)
    test_accuracy_SGD[i]=logreg_SGD.score(X_test,Y_test)
    critical_accuracy_SGD[i]=logreg_SGD.score(X_critical,Y_critical)

    print('SGD: %0.4f, %0.4f, %0.4f' %(train_accuracy_SGD[i],test_accuracy_SGD[i],critical_accuracy_SGD[i]) )

    print('finished computing %i/11 iterations' %(i+1))

# plot accuracy against regularisation strength
plt.semilogx(lmbdas,train_accuracy,'*-b',label='liblinear train')
plt.semilogx(lmbdas,test_accuracy,'*-r',label='liblinear test')
plt.semilogx(lmbdas,critical_accuracy,'*-g',label='liblinear critical')

plt.semilogx(lmbdas,train_accuracy_SGD,'*--b',label='SGD train')
```

```python
###### apply logistic regression
import logisRegresANA
import importlib
importlib.reload(logisRegresANA)
lr = 0.5
epochs = 300
weights= logisRegresANA.logistic_reg(X_train, Y_train, epochs, lr)
```

```python
###### apply logistic regression
import logisRegresANA
import importlib
importlib.reload(logisRegresANA)
weights=logisRegresANA.stocGradAscentA(X_train,Y_train)
```

```python
###### apply logistic regression
import logisRegresANA
import importlib
importlib.reload(logisRegresANA)
weights = logisRegresANA.steepest_descent_auto(X_train, Y_train, alpha =0.001)
error_train = logisRegresANA.simptest(weights, X_train, Y_train)
error_test = logisRegresANA.simptest(weights, X_test, Y_test)
```

```python
###### apply logistic regression
import logisRegresANA
import importlib
importlib.reload(logisRegresANA)
weights = logisRegresANA.logistic_reg(X_train, Y_train, epochs=100, lr=0.001)
error_train = logisRegresANA.simptest(weights, X_train, Y_train)
error_test = logisRegresANA.simptest(weights, X_test, Y_test)
```

```python
###### apply logistic regression
import logisRegresANA
import importlib
importlib.reload(logisRegresANA)
weights = logisRegresANA.gradDscent(X_train, Y_train, alpha= 0.01)
```

```python
import importlib
import logisRegresANA
importlib.reload(logisRegresANA)

weights2 = logisRegresANA.sgd(X_train, Y_train)
print(weights2)
weights2 = weights2.flatten()
error_train = logisRegresANA.simptest(weights2, X_train, Y_train)
error_test = logisRegresANA.simptest(weights2, X_test, Y_test)
```

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

1.6 Part e) Classifying the Ising model phase using neural networks.

```python
import importlib
import logisRegresANA
importlib.reload(logisRegresANA)

in_layer = X_train.shape[1] #number of neurons in the input layer

if (len(Y_train.shape)==1):
    out_layer = 1   #number of neurons in the output layer
else: out_layer = Y_train.shape[1]
biasesnn, weightsnn= logisRegresANA.neuralnetwork([in_layer, 10, out_layer], X_train, Y_train,
                                                                validation_x= X_test, validation_y=Y_test,
                                                                verbose=True,
                                    epochs= 30, mini_batch_size = 10, lr= 0.5, C='ce')
```

```
mini_batch  [[-1 -1 -1 ..., -1 -1  1]
 [-1 -1 -1 ..., -1 -1  0]
 [ 1  1  1 ...,  1  1  1]
 ...,
 [ 1  1  1 ..., -1 -1  0]
 [-1 -1 -1 ..., -1 -1  1]
 [ 1  1 -1 ...,  1 -1  0]]
nabla_w from weights -shape  (2,)
nabla_b from weights -shape  (2,)
x  [-1 -1 -1 ..., -1 -1 -1]
y  1
activations:  [array([-1, -1, -1, ..., -1, -1, -1])]
f 0 w in loop [[ 0.50892274 -0.28898022 -1.20827711 ..., -0.456469   -
0.65965346
   0.10241962]
 [ 2.3897599   1.29156472  1.27318914 ..., -0.55426639 -0.94425406
  -0.90373304]
 [ 0.97948496 -0.55957186 -0.1711543  ...,  0.08315642 -0.76727449
  -0.1474379 ]
 ...,
 [-0.75724226 -0.53077548 -1.08823808 ..., -0.28865485  0.48977372
   1.44011092]
 [-0.01027559  0.83852329 -0.11277865 ..., -0.85940721  0.45309176
  -0.42789696]
```

```
 [ 0.26413263  1.05422235 -0.36335701 ..., -1.30188984 -0.40503143
  -0.68552322]]
activation :  [-1 -1 -1 ..., -1 -1 -1]
w_a =  numpy.dot(w , activation):  [-13.49316819 -39.12589526 -
10.07647132  37.48403681 -26.31504817
  15.09851548 -13.01447302  18.43597532 -29.68447902  -0.04488525]
b : [-0.9073413  -1.36768501 -0.35959421 -0.89205373 -1.35507506 -
0.08539942
 -1.56692176  0.74175751  0.30745289 -0.95706228]
z = w_a + b :  [-14.40050949 -40.49358026 -10.43606554  36.59198309 -
27.67012324
  15.01311605 -14.58139478  19.17773283 -29.37702613  -1.00194753]
f 1 w in loop [[ 0.74683728 -1.36658155  1.73000956 -0.99789933 -
0.8230627  -0.840913
  -0.48278465  0.37804188  0.20443838  0.55033394]]
activation :  [  5.57106144e-07   2.59335242e-18   2.93536138e-05
1.00000000e+00
   9.61652500e-13   9.99999698e-01   4.64922440e-07   9.99999995e-01
   1.74469556e-13   2.68558686e-01]
w_a =  numpy.dot(w , activation):  [-1.31292226]
b : [ 0.6802191]
z = w_a + b :  [-0.63270316]
ACTIVATIONS :  [-1 -1 -1 ..., -1 -1 -1]
[  5.57106144e-07   2.59335242e-18   2.93536138e-05   1.00000000e+00
   9.61652500e-13   9.99999698e-01   4.64922440e-07   9.99999995e-01
   1.74469556e-13   2.68558686e-01]
[ 0.34689786]
in last layer, y  1 activations[-1 ] [ 0.34689786]
zs stored : [array([-14.40050949, -40.49358026, -10.43606554,
36.59198309,
       -27.67012324,  15.01311605, -14.58139478,  19.17773283,
       -29.37702613,  -1.00194753]), array([-0.63270316])]
DELTA :  [-0.65310214]
in backprop: numpy.array(activations[-2]).T  [  5.57106144e-07
2.59335242e-18   2.93536138e-05   1.00000000e+00
   9.61652500e-13   9.99999698e-01   4.64922440e-07   9.99999995e-01
   1.74469556e-13   2.68558686e-01]
in backprop: nabla_b_backprop[-1]  [-0.65310214]
in backprop: nabla_w_backprop[-1]  [ -3.63847217e-07  -1.69372402e-18  -
1.91709081e-05  -6.53102143e-01
  -6.28057308e-13  -6.53101946e-01  -3.03641842e-07  -6.53102140e-01
  -1.13946441e-13  -1.75396253e-01]
ENTRA NESTE LOOP, k =  1
sp  [  5.57105834e-07   2.59335242e-18   2.93527522e-05   2.22044605e-
16
   9.61652500e-13   3.01916105e-07   4.64922224e-07   4.69047088e-09
   1.74469556e-13   1.96434918e-01]
numpy.array(weights[-1- (k-1)]).T  [[ 0.74683728]
 [-1.36658155]
 [ 1.73000956]
 [-0.99789933]
 [-0.8230627 ]
 [-0.840913  ]
 [-0.48278465]
 [ 0.37804188]
```

```
 [ 0.20443838]
 [ 0.55033394]]
delta  -0.653102142795
delta in loop k [ -2.71734513e-07   2.31461199e-18  -3.31648807e-05
1.44713173e-16
   5.16930545e-13   1.65812954e-07   1.46593551e-07  -1.15807709e-09
  -2.32950262e-14  -7.06034783e-02]
testyy [-1 -1 -1 ..., -1 -1 -1]
```

```
    ---------------------------------------------------------------------------
```

```
    ValueError                                Traceback (most recent
↪ call last)
```

```
    <ipython-input-29-0374de8f71b8> in <module>
         11
↪ validation_x=X_test, validation_y=Y_test,
         12                                                         verbose=
↪ True,
    ---> 13                              epochs= 30, mini_batch_size =
↪ 10, lr= 0.5, C='ce')
```

```
    ~/Documents/FYS-STK4155/Project2/logisRegresANA.py in neuralnetwork
↪ (sizes, X_train, Y_train, validation_x, validation_y, verbose,
↪ epochs, mini_batch_size, lr, C)
        138     #print('initial weights ', weights)
        139     biases, weights = SGD(X_train, Y_train, epochs,
↪ mini_batch_size, lr, C, sizes, num_layers,
    --> 140         biases, weights, verbose, validation_x,
↪ validation_y)
        141     return biases, weights
        142 def SGD(X_train, Y_train, epochs, mini_batch_size, lr, C,
↪ sizes, num_layers,
```

```
    ~/Documents/FYS-STK4155/Project2/logisRegresANA.py in SGD(X_train,
↪ Y_train, epochs, mini_batch_size, lr, C, sizes, num_layers,
↪ biases, weights, verbose, validation_x, validation_y)
        152         #feed-forward (and back) all mini_batches
        153         for k, minib in enumerate(mini_batches):
    --> 154             biases, weights = update_mini_batch(minib, lr,
↪ C, sizes, num_layers,biases,weights)
        155
        156         if(verbose):
```

```
    ~/Documents/FYS-STK4155/Project2/logisRegresANA.py in
↪ update_mini_batch(minibatch, lr, C, sizes, num_layers, biases,
↪ weights)
        185             print('y ', y)
        186             #backpropagation for each observation in
↪ mini_batch
    --> 187             delta_nabla_b, delta_nabla_w = backprop(x, y, C
↪ ,sizes,num_layers,biases,weights)
        188             #print(delta_nabla_b.shape, 'delta_nabla_b ',
↪ delta_nabla_b)
        189             print(delta_nabla_w.shape, 'delta_nabla_w ',
```

```
↪ delta_nabla_w)
```

```
~/Documents/FYS-STK4155/Project2/logisRegresANA.py in backprop(x, y
↪ , C, sizes, num_layers, biases, weights)
    261             testyy = numpy.array(activations[-1 - (k+1)]).T
    262             print('testyy', testyy)
--> 263             nabla_w_backprop[-1 -(k)] = numpy.multiply(
↪ delta , testyy)
    264
    265     return nabla_b_backprop, nabla_w_backprop
```

```
ValueError: operands could not be broadcast together with shapes
↪ (10,) (1600,)
```

1.7 Part f) Critical evaluation of the various algorithms.

2 References