Final_submission

December 11, 2024

0.1 setup and import data

[8]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad
from autograd import hessian
import pandas as pd
from scipy.stats import mode
data = pd.read_csv('data.csv')
#import standard functions
%run basic_functions.ipynb
%run poly_k_functions.ipynb
%run RBF_functions.ipynb
```

0.2 import and process data

[9]:
```
#extract task 1 features
task_one = data.iloc[:, 1:19]

#extract diagnosis
diag = data.iloc[:,-1]
diag = diag.map({'P': 1, 'H' : -1})
task_one = task_one.to_numpy().T
diag = diag.to_numpy().T
diag = diag.reshape((1,174))

#all task data
all_task = data.iloc[:,1:-1]
all_task = all_task.to_numpy().T
all_task_norm,means,std = standard_normalise(all_task)

#some stats on the data
count = np.size(diag)
no_alz = np.count_nonzero(diag ==1)
no_healthy = np.count_nonzero(diag == -1)

print(f'Dataset size: {count}')
print(f'# Alzheimer: {no_alz} , {100*round(no_alz/count,2)} %')
```

```
print(f'# Healthy: {no_healthy} , {100*round(no_healthy/count,2)} %')
```

```
Dataset size: 174
# Alzheimer: 89 , 51.0 %
# Healthy: 85 , 49.0 %
```

0.3 Logistic regression classifer

Create a 5 fold data split on training data, optimise each split with gradient descent, do retrospective 'early stopping'by taking the final weights of the model as the ones which produce the lowest validation error during optimisation. Take the 3 best most accurate models per 5-fold cross validation run. Run these top 3 models with the test data, then bag the result (modal output) to calculate accuracy, precision and specifcity. Perform this 20 times, with randomised train (which includes validation) and test data, take a mean of performance staistics to provide an overall evaulation of logistic classification for this dataset.

[12]:
```
runs =20
run_verbose =0
```

[13]:
```
g= reg_softmax
alpha = 0.01
its = 5000
plot = 0

runs_acc = []
runs_pre = []
runs_spc = []

for j in range(runs):

    if (run_verbose ==1):
        print(f'Run: {R}')

    #set x train and test
    x_train,y_train,x_test,y_test = train_split(all_task_norm,diag,0.8)

    #constants for k-fold
    fold_parts = 5
    # generate indices of non-overlapping k fold sections
    num_samples = x_train.shape[1]
    indices = np.arange(num_samples)
    sub_indices = np.array_split(indices,fold_parts)

    acc_h = []
    w_h = []
    for i in range(fold_parts):
            #split training and validation data
            x_train_fold = np.delete(x_train,sub_indices[i],axis = 1)
```

```
            y_train_fold = np.delete(y_train,sub_indices[i],axis = 1)
            x_valid_fold = x_train[:,sub_indices[i]]
            y_valid_fold = y_train[:,sub_indices[i]]

            #set x and y so functions can access them
            x=x_train_fold
            y=y_train_fold #accessed in grad descent in cost function
            x_valid = x_valid_fold
            y_valid = y_valid_fold

            #train
            w_init = np.random.randn(451,1)
            w_history,cost_history,w_f,w_optimal = LR_gradient_descent(g,w_init,its,alpha,plot=0)

            acc,pre,spc = evaluate(w_optimal,x_valid_fold,y_valid_fold,0)
            w_h.append(w_optimal)
            acc_h.append(acc)

    #take 3 highest accuracy models out of the 5 and bag modal result
    best_index = np.argsort(acc_h)[::-1][:3]
    length = np.shape(x_test)[1]
    y_p_sub = np.zeros((3,length))
    for i in range(3):
        y_p_sub[i] = np.sign(model(x_test,w_h[best_index[i]]))
    y_p = mode(y_p_sub,axis=0)[0].reshape((1,length))
    acc,pre,spc = bag_eval(y_test,y_p,verbose =0)

    runs_acc.append(acc)
    runs_pre.append(pre)
    runs_spc.append(spc)

    if (run_verbose ==1): print(f'Run accuracy: {acc}')

print(f'//////////////////////////////')
print(f'Training and testing over')
print(f'Overall classifier performance:')
print(f'Mean accuracy: {round(np.mean(runs_acc),2)}')
print(f'Standard deviation: {round(np.std(runs_acc),2)}')
print(f'Mean precision: {round(np.mean(runs_pre),2)}')
print(f'Mean specificity: {round(np.mean(runs_spc),2)}')
print(f'//////////////////////////////')
```

```
//////////////////////////////
Training and testing over
Overall classifier performance:
Mean accuracy: 75.86
Standard deviation: 3.99
```

```
Mean precision: 82.29
Mean specificity: 71.01
//////////////////////////////
```

0.4 Cross validation of polynomial kernel method with regularisation, to find optimal regularisation parameter

For each step in regularisation parameter sweep, split training data into 5 folds, optimise model for each fold configuration, take mean accuracy of all models. The optimal regularisation parameter is the one which produces the highest mean accuracy of the 5-fold configurations. Optimise a model on all training data, using the optimal parameter found, test with test data, find accuracy. During this final optimisation, take the model which produces the lowesest testing validation error, which is not neccesarily the final model at the end of optimisation. Overall classisification accuracy is the mean of the test accuracy over all runs (20). This system uses polynomial kernel, D = 3.

[14]:
```
runs = 20
run_verbose = 0
```

[15]:
```
runs_acc = []
runs_pre = []
runs_spc = []

#gradient descent parameters
g= reg_k_softmax
L_set = np.linspace(0,5,10)
D = 3
C = 1
its = 1000
alpha = 0.1

for R in range(runs):

    if (run_verbose ==1):
        print(f'Run: {R}')

    #generate training and testing split
    x_train,y_train,x_test,y_test = train_split(all_task_norm,diag,0.8)

    #constants for k-fold
    fold_parts = 5
    # generate indices of non-overlapping k fold sections
    num_samples = x_train.shape[1]
    indices = np.arange(num_samples)
    sub_indices = np.array_split(indices,fold_parts)

    overall_acc = []

    for j in range(len(L_set)):
```

```python
        L  = L_set[j]

        acc_h = []
        spc_h = []
        pre_h = []

        #perform 5 fold cross evlaution to find performance of model
        z_init_set = np.random.randn(200,1)
        for i in range(fold_parts):
            #split training and validation data
            x_train_fold = np.delete(x_train,sub_indices[i],axis = 1)
            y_train_fold = np.delete(y_train,sub_indices[i],axis = 1)
            x_valid_fold = x_train[:,sub_indices[i]]
            y_valid_fold = y_train[:,sub_indices[i]]

            y=y_train_fold #accessed in grad descent in cost function
            y_valid = y_valid_fold

            #generate z_init
            train_size = np.shape(x_train_fold)[1]
            z_init = z_init_set[:train_size+1]

            #generate kernel and valid kernel matrix
            H = poly_k_matrix(x_train_fold,D,C)
            H_valid = make_H_valid(x_train_fold,x_valid_fold)

            #run grad descent for model, record accuracy etc
            w_history,cost_history,z_f,optimal_z =␣
↪kernel_gradient_descent(g,z_init,its,alpha,plot=0)
            acc,pre,spc = evaluate(z_f,H_valid,y_valid_fold,verbose=0)

            #store evaluation history of k-fold
            acc_h.append(acc)

        #find mean accuracy of whole k-fold evaluation
        acc= np.mean(acc_h)
        overall_acc.append(acc)

    #identify optimal L Value
    L_optimal_index = np.argmax(overall_acc)
    L_optimal = L_set[L_optimal_index]

    # train whole traning data and test with test data
    train_size = np.shape(x_train)[1]

    z_init = np.random.randn(train_size+1,1)
    y = y_train
```

```
    y_valid = y_test

    #generate kernel and test kernel matrix
    L =L_optimal
    H = poly_k_matrix(x_train,D,C)
    H_valid = make_H_valid(x_train,x_test)
    w_history,cost_history,z_f,optimal_z =␣
↪kernel_gradient_descent(g,z_init,its,alpha,plot=0)
    acc,pre,spc = evaluate(optimal_z,H_valid,y_valid,verbose=0)
    runs_acc.append(acc)
    runs_pre.append(pre)
    runs_spc.append(spc)

    if (run_verbose ==1):
        print(f'Run accuracy: {acc}')

print(f'//////////////////////////////')
print(f'Training and testing over')
print(f'Overall classifier performance:')
print(f'Mean accuracy: {round(np.mean(runs_acc),2)}')
print(f'Standard deviation: {round(np.std(runs_acc),2)}')
print(f'Mean precision: {round(np.mean(runs_pre),2)}')
print(f'Mean specificity: {round(np.mean(runs_spc),2)}')
#what was the value of L found in the last run, just for reference?
print(f'Optimal L: {round(L_optimal,4)}')
print(f'//////////////////////////////')
```

```
//////////////////////////////
Training and testing over
Overall classifier performance:
Mean accuracy: 86.14
Standard deviation: 4.17
Mean precision: 90.02
Mean specificity: 83.98
Optimal L: 3.3333
//////////////////////////////
```

0.5 Cross validation of RBF kernel

Same as above, but using RBF kernel instead of polynomial kernel. No regulariser. Performing sweep of values for beta (sometimes called gamma).

[16]:
```
runs  = 20
run_verbose = 0
```

[17]:
```
runs_acc = []
runs_pre = []
runs_spc = []
```

```python
#gradient descent parameters
g= k_softmax
its = 1000
alpha = 0.1
steps = 18
betas = np.linspace(0.0005,0.01,steps)

for R in range(runs):

    if (run_verbose ==1):
        print(f'Run: {R}')

    #generate training and testing split
    x_train,y_train,x_test,y_test = train_split(all_task_norm,diag,0.8)

    #constants for k-fold
    fold_parts = 5
    # generate indices of non-overlapping k fold sections
    num_samples = x_train.shape[1]
    indices = np.arange(num_samples)
    sub_indices = np.array_split(indices,fold_parts)

    overall_acc = []

    for j in range(len(betas)):
        beta  = betas[j]

        acc_h = []
        spc_h = []
        pre_h = []

        #perform 5 fold cross evlaution to find performance of model
        z_init_set = np.random.randn(200,1)
        for i in range(fold_parts):
            #split training and validation data
            x_train_fold = np.delete(x_train,sub_indices[i],axis = 1)
            y_train_fold = np.delete(y_train,sub_indices[i],axis = 1)
            x_valid_fold = x_train[:,sub_indices[i]]
            y_valid_fold = y_train[:,sub_indices[i]]

            y=y_train_fold #accessed in grad descent in cost function
            y_valid = y_valid_fold

            #generate z_init
```

```
            train_size = np.shape(x_train_fold)[1]
            z_init = z_init_set[:train_size+1]

            #generate kernel and valid kernel matrix
            H = RBF_matrix(x_train_fold,beta)
            H_valid = make_H_RBF_valid(x_train_fold,x_valid_fold,beta)

            #run grad descent for model, record accuracy etc
            w_history,cost_history,z_f,optimal_z =␣
↪kernel_gradient_descent(g,z_init,its,alpha,plot=0)
            acc,pre,spc = evaluate(z_f,H_valid,y_valid_fold,verbose=0)

            #store evaluation history of k-fold
            acc_h.append(acc)

        #find mean accuracy of whole k-fold evaluation
        acc= np.mean(acc_h)
        overall_acc.append(acc)

    #identify optimal beta Value
    beta_optimal_index = np.argmax(overall_acc)
    beta_optimal = betas[L_optimal_index]

    # train whole traning data and test with test data
    train_size = np.shape(x_train)[1]
    z_init = np.random.randn(train_size+1,1)
    y = y_train
    y_valid = y_test

    #generate kernel and test kernel matrix, take model at point that is␣
↪produces lowest test error
    beta = beta_optimal
    H = RBF_matrix(x_train,beta_optimal)
    H_valid = make_H_RBF_valid(x_train,x_test,beta)
    w_history,cost_history,z_f,optimal_z =␣
↪kernel_gradient_descent(g,z_init,its,alpha,plot=0)
    acc,pre,spc = evaluate(optimal_z,H_valid,y_valid,verbose=0)
    runs_acc.append(acc)
    runs_pre.append(pre)
    runs_spc.append(spc)

    if (run_verbose ==1):
        print(f'Run accuracy: {acc}')

print(f'//////////////////////////////')
print(f'Training and testing over')
print(f'Overall classifier performance:')
```

```
print(f'Mean accuracy: {round(np.mean(runs_acc),2)}')
print(f'Standard deviation: {round(np.std(runs_acc),2)}')
print(f'Mean precision: {round(np.mean(runs_pre),2)}')
print(f'Mean specificity: {round(np.mean(runs_spc),2)}')
print(f'//////////////////////////////')
```

```
//////////////////////////////
Training and testing over
Overall classifier performance:
Mean accuracy: 77.57
Standard deviation: 5.37
Mean precision: 76.16
Mean specificity: 80.82
//////////////////////////////
```

0.6 MLP - boosting

Boost a neural network one unit at a time, choose model with number of units that has lowest validation misclassifications. Then train a NN with this optimal number of units, using all trian data and test with test data. Could be imprived by traing using k-fold cross validation system then bagging output.

[19]:
```
%run NN_functions.ipynb
#this overwrites some functions used previously, so restart kernel if you want␣
↪to run the previous classifiers
```

[18]:
```
NN_verbose = 0
runs =  20
max_units = 20
```

[20]:
```
runs_acc = []
runs_pre = []
runs_spc = []
for R in range(runs):

    if(NN_verbose == 1): print(f'Run: {R}')

    overall_x_train,overall_y_train,x_test,y_test =␣
↪train_split(all_task_norm,diag,0.8)
    x_train,y_train,x_valid,y_valid =␣
↪train_split(overall_x_train,overall_y_train,0.75)

    #set NN parameters
    U_1 = 1
    C =1
    N=450
    layer_sizes = [N, U_1, C]
    #theta_init is always a one unit NN
```

```
    theta_init = network_initializer(layer_sizes,1)

    # run for up to 1 to 30 units
    system_valid_misclass = []
    system_train_misclass = []

    #gradient descent parameters
    its =1000
    alpha = 1
    g=reg_softmax

    x = x_train
    y = y_train

    #run system with one unit, genenerate first set w_f
    theta_init = network_initializer(layer_sizes,1)
    w_history,cost_history,w_f,train_misclass = 
↪NN_gradient_descent(g,theta_init,its,alpha,plot=0)
    w_set = w_f

    valid_misclass = missed_class(w_set, x_valid, y_valid)

    system_valid_misclass.append(valid_misclass)
    system_train_misclass.append(train_misclass)

    # change cost function
    g = boost_softmax


    #run boosting, adding 1 unit at a time
    for i in range(1,max_units):

        #set w_set

        #print(f'# units: {i+1}')

        #run to generate new w_f
        #theta_inti is start values for new unit
        theta_init = network_initializer(layer_sizes,1)
        w_history,cost_history,w_f,train_misclass = 
↪NN_gradient_descent(g,theta_init,its,alpha,plot=0)

        # system to update weights of system incrementally:

        #append internal weights
        new_w_0 = np.append(w_set[0], w_f[0], axis=2)
```

```
        #sum external non-touching weights
        w_set[1][0] = w_set[1][0] + w_f[1][0]

        #append external weights
        new_w_1 = np.append(w_set[1],w_f[1][1])
        new_w_1 = new_w_1.reshape((np.size(new_w_1),1))

        #create new w_set
        w_set = [new_w_0, new_w_1]

        valid_misclass = missed_class(w_set, x_valid, y_valid)
        #print(f'validation misclassifications: {valid_misclass}')

        # append training and validation misclasses to array
        system_valid_misclass.append(valid_misclass)
        system_train_misclass.append(train_misclass)


    #choose system with least valid misclass
    best_no_unit = np.argmin(system_valid_misclass)+1

    if (NN_verbose == 1):
        #display training and validation error
        x_axis = np.arange(1,max_units+1)
        plt.figure()
        plt.plot(x_axis,system_train_misclass,label = 'training')
        plt.plot(x_axis,system_valid_misclass,label = 'validation')
        plt.xlabel('number of neural network units')
        plt.ylabel('misclassifications')
        plt.title(f'Run number: {R}')
        plt.legend()
        print(f'no_units with lowest misclass = {best_no_unit}')



    #train all training data and test with test data
    x = overall_x_train
    y = overall_y_train
    x_valid = x_test
    y_valid = y_test

    U_1 = best_no_unit
    layer_sizes = [N, U_1, C]
    theta_init = network_initializer(layer_sizes,1)

    #train on all training data
```

```
    w_history,cost_history,w_f,train_misclass = NN_gradient_descent(g,theta_init,its,alpha,plot=0)

    #evaluate testing data
    acc,pre,spc = evaluate(w_f,x_test,y_test,verbose=0)

    if(NN_verbose ==1 ):
        print(f'Run accuracy: {acc}')

    runs_acc.append(acc)
    runs_pre.append(pre)
    runs_spc.append(spc)


print(f'//////////////////////////////')
print(f'Training and testing over')
print(f'Overall classifier performance:')
print(f'Mean accuracy: {round(np.mean(runs_acc),2)}')
print(f'Standard deviation: {round(np.std(runs_acc),2)}')
print(f'Mean precision: {round(np.mean(runs_pre),2)}')
print(f'Mean specificity: {round(np.mean(runs_spc),2)}')
print(f'//////////////////////////////')
```

```
//////////////////////////////
Training and testing over
Overall classifier performance:
Mean accuracy: 59.57
Standard deviation: 8.86
Mean precision: 59.46
Mean specificity: 60.93
//////////////////////////////
```