Gauss-Southwell coordinate selection for Greedy coordinate descent

Prasad Madhava Kamath
pkamath@ucsd.edu

Abstract

In this paper we describe a method for selection of a coordinate for coordinate descent algorithm applied to an unconstrained optimization problem. Here we investigate and compare the differences between different weight update/optimization schemes such as gradient descent, coordinate descent with random coordinate selection and coordinate descent using a greedy coordinate selection algorithm. The greedy based algorithm 'greedily' selects the coordinate to be updated and is proven to converge faster than choosing random coordinates for problems with sparse solution, although the computations per iteration increases due to added checks for update.The greedy algorithm here uses a Gauss-Southwell update rule which assumes that the function is smooth, convex and differentiable.

1 Introduction

In this paper we explore the coordinate descent algorithm applied to a unconstrained optimization problem, in this the logistic regression problem. coordinate descent stands out as a popular algorithm for problems with sparse solution due to its low computational complexity as compared to other weight update methods where the closed form solution to the objective function doesn't exist.
We first consider the standard gradient update scheme where the gradient is computed across all 'n' dimensions and all 'n' weights are updated simultaneously to minimize the objective function at each iteration. In the gradient descent scheme, the weight update is done once over all the samples in the training data set, which is computationally expensive. As a alternative to this, the stochastic gradient descent scheme updates the weight after each train sample if fed to the system. This tends to be computationally less expensive and is shown to converge to the global minima of the objective function.
In contrast to the schemes described above the coordinate descent scheme is an optimization scheme which minimizes the objective function at every iteration by updating only one of the weights. The algorithm uses a selection rule to determine which coordinate to update at each iteration so as to minimize the objective function (loss). In this paper we describe the greedy coordinate descent algorithm. The Greedy coordinate descent algorithm has a selection rule based on the magnitude of partial derivatives or minimization of the objective function. Assuming the objective function is smooth The Gauss-Southwell selection rule has very fast convergence in terms of number of iterations (Shi et al., 2017) .

In the following sections we describe coordinate descent algorithms in greater detail and bring out the differences

2 Methodology

2.1 Dataset

In this paper we consider the Wine dataset. Wine dataset is a multi-class classification dataset consisting of data on wine classes described by 13 attributes which is described in (Dua and Graff, 2017). The dataset consists of 3 classes and a total of 179 data samples each being a 13 dimensional vector. The split of samples amongst the 3 classes is as follows.

1. class label 0 : 59 sample points
2. class label 1 : 71 sample points
3. class label 2 : 48 sample points

For the experiments described in this paper we consider only class 0 and class 1 sample points such that we have a binary classification problem. The pseudo code below illustrates the steps in dataset preparation for our experimental setup.

Algorithm 1 Dataset Preparation

for $(sample, label)$ *in wine dataset* **do**
  if $label == 0$ *or* $label == 1$ **then**
    $sample = (sample - mean)/std$
    $data.append(sample, label)$
  end if
end for

where the mean and std are the mean and standard deviation over the samples in the two classes under considerations. The mean normalised data is stored and split into train and test data for our experiments.

2.2 Objective Function

To classify the two wine classes we use a supervised binary classifier specifically in this case we use a logistic regression algorithm to classify the data. The objective function for the logistic regression algorithm is described below. The function is smooth, convex and differentiable everywhere.

$$L(w,b) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(\hat{y}^{(i)})+(1-y^{(i)})log((1-\hat{y}^{(i)}))]$$

Where L is the logistic losss, m is the number of training samples in the dataset. $y^{(i)}$ is the label of i^th sample in the dataset and $\hat{y}^{(i)}$ is the predicted output for the i^th input sample. w is the weight vector and b is the bias, here the bias is absorbed into the weight vector.$\hat{y}^{(i)}$ is computed as follows

$$\hat{y}^{(i)} = (1/(1 + exp(-w^T x^{(i)})))$$

2.3 Coordinate descent

We implement the coordinate descent algorithm with Gauss-Southwell coordinate selection as a greedy algorithm. In the Gauss-Southwell update rule, we find the maximum of the absolute value of the gradient of the objective function with respect to the weights. This is the coordinate direction in which we perform the update. Only this particular weight is updated in the weight vector keeping all other weights constant in the given iteration thus forming the coordinate descent step. For the greedy coordinate descent using Gauss-Southwell rule, the weights can be initialized to zeros. This has empirically been proven to provided the fastest convergence and independent of dimension n (Karimireddy et al., 2019). The underlying assumptions here are that the objective function is smooth, convex and differentiable everywhere. The pseudo code for the same is described below

Algorithm 2 Greedy coordinate descent with Gauss Southwell selection rule

Input: function $L(w)$,(X, Y)
$w = [0, 0....0]$
for $t = 1, 2...K$ **do**
  Coordinate selection:select k according to rule
  $\nabla L(w^{t-1}) = \frac{1}{n}(X^T(\hat{Y} - Y))$
  $k = argmax(|\nabla_k L(w^{t-1})|)$
  $w_k^t = w_k^{t-1} - \eta \nabla_k L(w^{t-1})$
end for

3 Experimental results

As a baseline reference for accuracy and convergence to compare the coordinate descent algorithm's performance we implement the gradient descent and the coordinate descent algorithm using randomized index selection for weight update for objective function optimization. We load and process the wine dataset as described in 2.1 from sklearn (Pedregosa et al., 2011). We then compute the accuracy metric and loss vs epoch (convergence rate) over the partitioned test dataset for the three schemes- Gradient Descent, Coordinate descent with random selection, Greedy coordinate descent.

3.1 Experiment 1: Baseline reference

3.1.1 Gradient Descent

To establish a baseline reference we first implement the gradient descent algorithm in which all the weights are simultaneously updated in one iteration over the entire dataset. The initial weights are randomly initialized and the learning rate η is set to 0.01. The results of the gradient descent based optimization for logistic regression based binary

	precision	recall	f1-score	support
0	0.96	1.00	0.98	22
1	1.00	0.95	0.98	21
accuracy			0.98	43
macro avg	0.98	0.98	0.98	43
weighted avg	0.98	0.98	0.98	43

Final loss= 0.016669597417297247

Figure 1: Accuracy and loss of Logistic regression using gradient descent scheme

	precision	recall	f1-score	support
0	0.92	1.00	0.96	22
1	1.00	0.90	0.95	21
accuracy			0.95	43
macro avg	0.96	0.95	0.95	43
weighted avg	0.96	0.95	0.95	43

Final loss= 0.11207624754815076

Figure 2: Accuracy and loss of Logistic regression using coordinate descent scheme with random feature selection

classifier over the wine dataset is as shown in 1. The loss after 10000 epochs is found to be 0.0167 and the accuracy of the classifier is 98%.

3.1.2 Coordinate Descent using random feature selection

We implement the coordinate descent scheme using random coordinate selection in which a particular weight w_k is randomly chosen updated in each iteration over the entire dataset. The initial weights are randomly initialized and the learning rate η is set to 0.01. The results of this scheme for logistic regression based binary classifier over the wine dataset is as shown in 2. The loss after 10000 epochs is found to be 0.112 and the accuracy of the classifier is 95%.

3.2 Experiment 2 : Greedy coordinate descent

The procedure for the greedy-coordinate descent algorithm is outlined in 2.3. Here, the weight initialization is empirically found to be of great importance to the rate of convergence. For this experiment we initialize the weights randomly. The learning rate η = 0.01. The accuracy of the classifier is nearly 100% after 10000 epochs and the loss is 0.06. The results of the experiment are summarised in the figure 3 A comparison of accuracy

	precision	recall	f1-score	support
0	1.00	1.00	1.00	22
1	1.00	1.00	1.00	21
accuracy			1.00	43
macro avg	1.00	1.00	1.00	43
weighted avg	1.00	1.00	1.00	43

Final loss= 0.06066325422932337

Figure 3: Accuracy and loss of Logistic regression using coordinate descent scheme with Gauss-Southwell selection rule

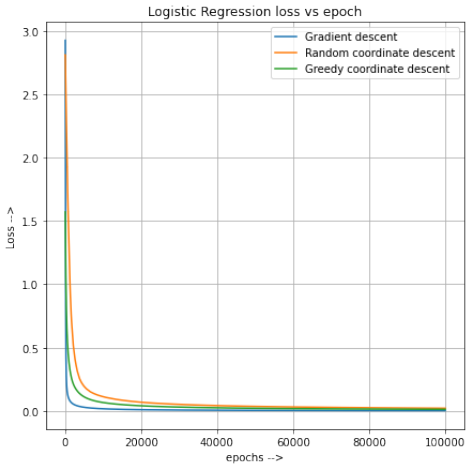


Figure 4: Comparison of convergence rate for the 3 different schemes

and the convergence of the various algorithms for the same step-size $\eta = 0.01$ and epochs =100000 is as shown in figure4 and in table 1. It can be inferred that the rate of convergence of the greedy coordinate descent algorithm is faster than the rate of convergence of random feature selection coordinate descent.

3.2.1 Effect of weight initialization on convergence

In this experiment we see the effect of weight initialization on the convergence of the greedy coordinate descent algorithm. The convergence appears to be comparable and appear to converge to same

Algorithm\learning rate	0.01		0.001	
epochs =100000	Accuracy	loss L*	Accuracy	Loss L*
Gradient Descent	98	0.016	98	0.016
Random Feature Selection	95	0.112	95	0.109
GS Feature Selection	100	0.06	100	0.066

Table 1: Comparison of accuracy and loss for the 3 different schemes

Algorithm\epochs	100000		200000	
LR=0.01	Accuracy	loss L*	Accuracy	Loss L*
Gradient Descent	98	0.016	98	0.001
Random Feature Selection	95	0.112	95	0.012
GS Feature Selection	100	0.06	100	0.007

Table 2: Analysis of convergence of the 3 schemes with number of epochs

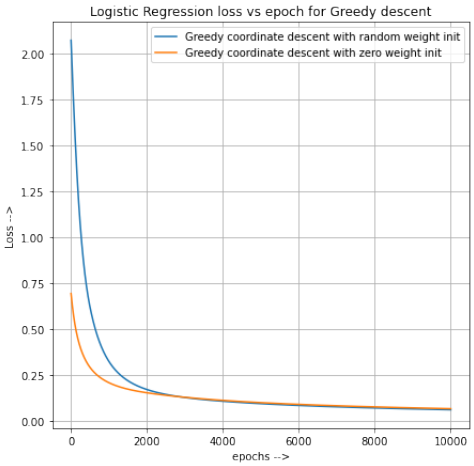


Figure 5: Convergence curves for greedy coordinate descent for different weight initialization

loss value finally at comparable rates. However as per literature, initializing with zero has the inherent advantage of introducing sparsity in features since the update only happens in certain coordinates (Karimireddy et al., 2019).Figure 5 is indicative of the convergence in both cases.

3.2.2 Effect of learning rate on convergence

In this experiment we see the effect of learning rate on the convergence of the greedy coordinate descent algorithm. The convergence appears to be generally faster in the case of a larger step size. As per literature the greedy coordinate descent can be accelerated by using a line search algorithm to choose an optimal step size, this would be a good area of implementation for future work.Figure 6 is indicative of the convergence in both cases.

3.2.3 Introduction of sparsity

The greedy descent method is supposed to be very efficient for problems with sparse solutions. The reason why zero initialization is effective in this case is because during the course of update of coordinates during Gauss-Southwell update, a lot of them remain zeros and unmodified throughout.This in turn leads to reducing the dimension of the problem since only few features are considered. Thus the convergence of the algorithm is quick(Shi et al., 2017).

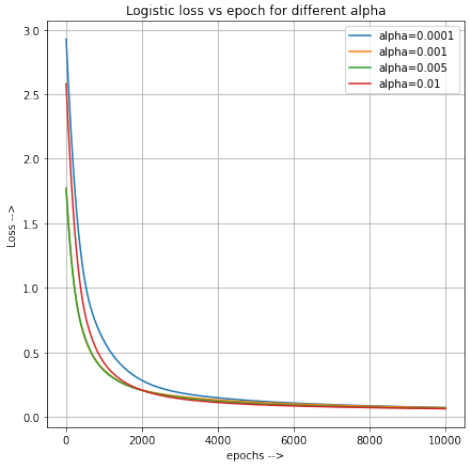


Figure 6: Convergence curves for greedy coordinate descent for different learning rates

4 Conclusion

From the experiments conducted it is evident that the greedy coordinate descent algorithm using the Gauss-Southwell coordinate selection rule outperforms the random feature selection coordinate descent scheme. It was also seen that for larger learning rate the convergence of the greedy coordinate scheme was better and it could be inferred that the scheme would benefit from a learning rate optimization scheme such as line search, which is an area of improvement.Although we discuss greedy coordinate descent here with regards to a smooth convex and differentiable function, it is possible to extend this scheme to non smooth and not differentiable objective functions using sub gradients as discussed in (Karimireddy et al., 2019). Further, there are other advantages of greedy coordinate descent algorithms when implemented non-trivially, such as all gradients do not need to be computed to find the update coordinate, there are nearest neighbor based update schemes. greedy coordinate descent using Gauss-Southwell selection also has the property that it can be parallalized and run improving run-time and computation.

References

Dheeru Dua and Casey Graff. 2017. UCI machine learning repository.

Sai Praneeth Karimireddy, Anastasia Koloskova, Sebastian U. Stich, and Martin Jaggi. 2019. Efficient greedy coordinate descent for composite problems.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Hao-Jun Michael Shi, Shenyinying Tu, Yangyang Xu, and Wotao Yin. 2017. A primer on coordinate descent algorithms.

Coordinate_descent

February 22, 2022

[22]:
```
# -*- coding: utf-8 -*-
"""
@author: prasad kamath
"""

import numpy as np;
import sklearn;
import sklearn.datasets as datasets
from sklearn.model_selection import train_test_split
from  sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.metrics.pairwise import euclidean_distances
from sklearn import metrics
from sklearn.datasets import load_wine
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix
```

1. Wine dataset preparation

[23]:
```
eps=100000
np.random.seed(1992)
# Get data
features, target = load_wine(return_X_y=True)
#normalize data
scaler = StandardScaler()
scaler.fit(features)
features=scaler.transform(features)
```

[24]:
```
x=[]
y=[]
#keep only the first two classes
for i,label in enumerate(target):
    if(label==0 or label==1):
        x.append(features[i])
        y.append(label)
x=np.array(x)
y=np.array(y)
```

[25]:
```
xtrain,xtest,ytrain,ytest=train_test_split(x, y, test_size=0.33,␣
↪random_state=42, shuffle=True)
```

[26]:
```
bias=np.ones((xtrain.shape[0],1))
xtrain=np.append(xtrain,bias,axis=1)
bias=np.ones((xtest.shape[0],1))
xtest=np.append(xtest,bias,axis=1)
```

2. Logistic regression using gradient descent

2(a). Logistic regression using sklearn

[27]:
```
model=LogisticRegression(penalty='none',random_state=0).fit(xtrain, ytrain)
ypredicted=(model.predict(xtest))
acc=100*accuracy_score(ypredicted,ytest)
print(classification_report(ytest,ypredicted))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       1.00      0.95      0.98        21

    accuracy                           0.98        43
   macro avg       0.98      0.98      0.98        43
weighted avg       0.98      0.98      0.98        43
```

2(b). Logistic regression using gradient descent

[28]:
```
def sigmoid(data):
    return (1.0/(1 + np.exp(-1*data)))
```

[29]:
```
def Lossfunction(weights,data,labels):
    out=np.dot(data,weights)
    ypred=sigmoid(out)
    yk=labels.reshape(labels.shape[0],1)
    op=(yk*np.log(ypred))+((1-yk)*np.log(1-ypred))
    err=-1.0*np.mean(op)
    return err
```

[30]:
```
def WeightUpdateGD(weights,data,labels,stepsize=0.001):
    N=data.shape[0]
    out=np.dot(data,weights)
    ypred=sigmoid(out)
    yk=labels.reshape(ytrain.shape[0],1)
    grad= (1/N)*(np.dot(data.T,(ypred-yk)))
    update=grad*stepsize
    weights-=update
    return weights
```

[31]:
```
def Predict(weights,data):
    yp=[]
    ypred=sigmoid(np.dot(data,weights))
    for y in ypred:
        if(y>=0.5):
            yp.append(1)
        else:
            yp.append(0)
    return yp
```

[32]:
```
epochs=eps
weights=np.random.rand(xtrain.shape[1],1)
loss_GD=[]
for epoch in range(epochs):
    loss=Lossfunction(weights,xtrain,ytrain)
    loss_GD.append(loss)
    weights=WeightUpdateGD(weights,xtrain,ytrain,0.01)
    ypred= Predict(weights,xtest)
    acc=100*accuracy_score(ytest,ypred)
ypred= Predict(weights,xtest)
print(classification_report(ytest,ypred))
print("Final loss= {0}".format(loss))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       1.00      0.95      0.98        21

    accuracy                           0.98        43
   macro avg       0.98      0.98      0.98        43
weighted avg       0.98      0.98      0.98        43

Final loss= 0.0024378625946884476
```

3. Logistic regression using random coordinate descent

[33]:
```
def WeightUpdateRCD(weights,data,labels,stepsize=0.01):
    N=data.shape[0]
    ypred=sigmoid(np.dot(data,weights))
    yk=labels.reshape(ytrain.shape[0],1)
    grad= (1/N)*(np.dot(data.T,(ypred-yk)))
    update=-1.0*grad*stepsize
    #get index of random coordinate and update only that index
    indx=np.random.randint(0,14)
    updatemod=np.zeros((14,1),np.float64)
    updatemod[indx]=update[indx]
    weights=np.add(weights,updatemod)
    return weights
```

[34]:
```
epochs=eps
weights=np.random.rand(xtrain.shape[1],1)
loss_RCD=[]
for epoch in range(epochs):
    loss=Lossfunction(weights,xtrain,ytrain)
    loss_RCD.append(loss)
    weights=WeightUpdateRCD(weights,xtrain,ytrain,stepsize=0.01)
    ypred= Predict(weights,xtest)
    #print(loss)
ypred= Predict(weights,xtest)
print(classification_report(ytest,ypred))
print("Final loss= {0}".format(loss))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       1.00      0.95      0.98        21

    accuracy                           0.98        43
   macro avg       0.98      0.98      0.98        43
weighted avg       0.98      0.98      0.98        43

Final loss= 0.021110415947651052
```

4. Logistic regression using Greedy coordinate descent

4(a). Logistic regression using Greedy coordinate descent vs Gradient descent and Random selection Coordinate descent

[35]:
```
def WeightUpdateCD(weights,data,labels,stepsize=0.01):
    N=data.shape[0]
    ypred=sigmoid(np.dot(data,weights))
    yk=labels.reshape(ytrain.shape[0],1)
    grad= (1/N)*(np.dot(data.T,(ypred-yk)))
    #find max gradient
    max_indx=np.argmax(np.abs(grad))
    update=-1.0*grad*stepsize
    #get index of random coordinate and update only that index
    indx=np.random.randint(0,14)
    updatemod=np.zeros((14,1),np.float64)
    updatemod[max_indx]=update[max_indx]
    weights=np.add(weights,updatemod)
    return weights
```

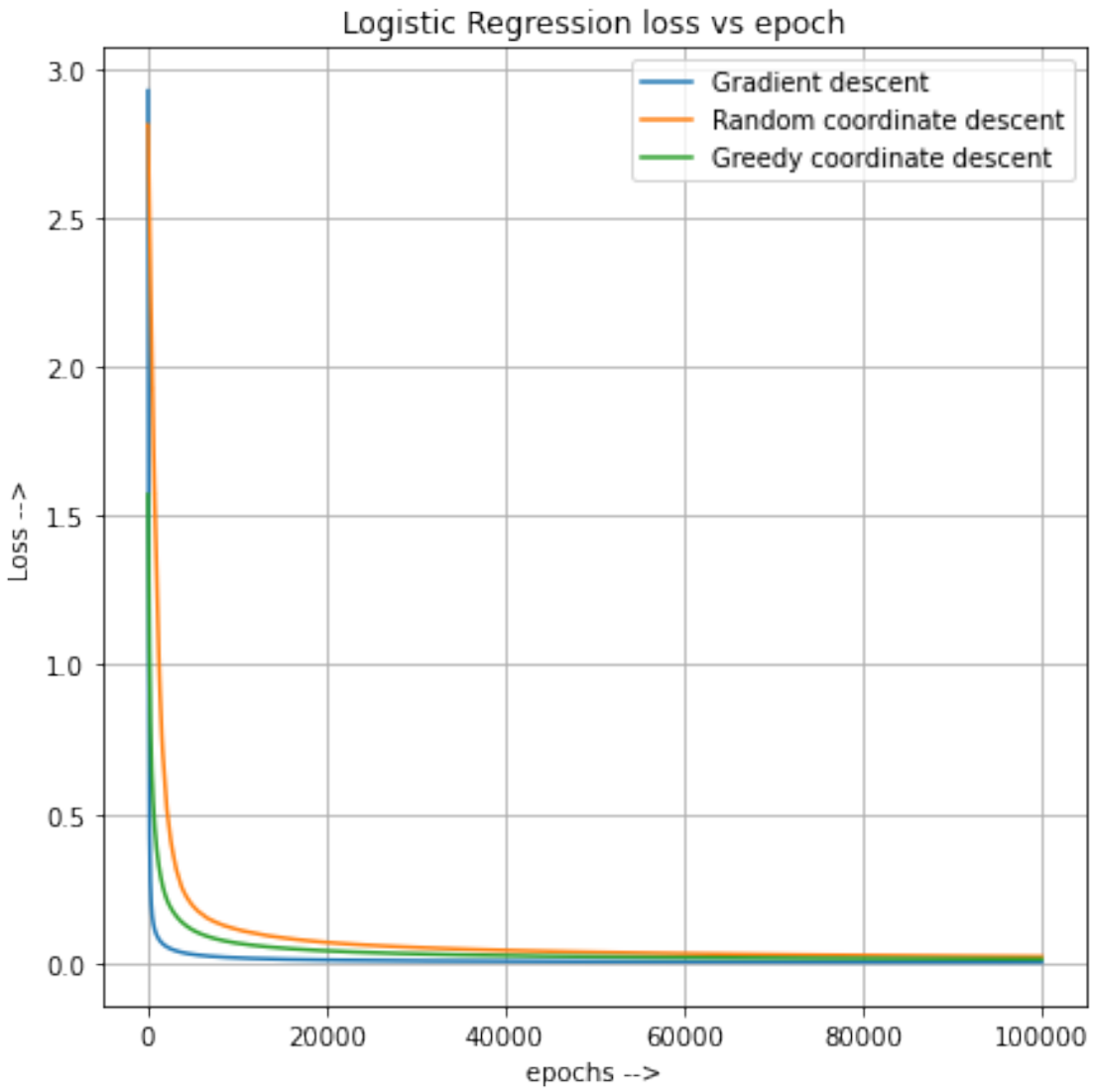
[36]:
```
epochs=eps
weights=np.random.rand(xtrain.shape[1],1)
loss_CD=[]
for epoch in range(epochs):
    loss=Lossfunction(weights,xtrain,ytrain)
```

```
    loss_CD.append(loss)
    weights=WeightUpdateCD(weights,xtrain,ytrain,stepsize=0.01)
    ypred= Predict(weights,xtest)
    #print(loss)
ypred= Predict(weights,xtest)
print(classification_report(ytest,ypred))
print("Final loss= {0}".format(loss))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       1.00      1.00      1.00        21

    accuracy                           1.00        43
   macro avg       1.00      1.00      1.00        43
weighted avg       1.00      1.00      1.00        43

Final loss= 0.012439214869525858
```

[37]:
```
fig = plt.figure(1)
fig.set_figheight(7)
fig.set_figwidth(7)
ax1= fig.add_subplot(1,1,1)
ax1.title.set_text('Logistic Regression loss vs epoch')
plt.xlabel('epochs -->')
plt.ylabel('Loss -->')
curve1,=ax1.plot(loss_GD)
curve2,=ax1.plot(loss_RCD)
curve3,=ax1.plot(loss_CD)
plt.grid(visible='True')
curve1.set_label('Gradient descent')
curve2.set_label('Random coordinate descent')
curve3.set_label('Greedy coordinate descent')
plt.legend()
```

[37]:
```
<matplotlib.legend.Legend at 0x2df136fc2e0>
```



4(b). Logistic regression using Greedy coordinate descent for different weight initializations

[38]:
```
epochs=eps
weights=np.random.rand(xtrain.shape[1],1)
loss_CD_rw=[]
for epoch in range(epochs):
    loss=Lossfunction(weights,xtrain,ytrain)
    loss_CD_rw.append(loss)
    weights=WeightUpdateCD(weights,xtrain,ytrain,stepsize=0.01)
    ypred= Predict(weights,xtest)
    #print(loss)
ypred= Predict(weights,xtest)
```

```
print(classification_report(ytest,ypred))
```

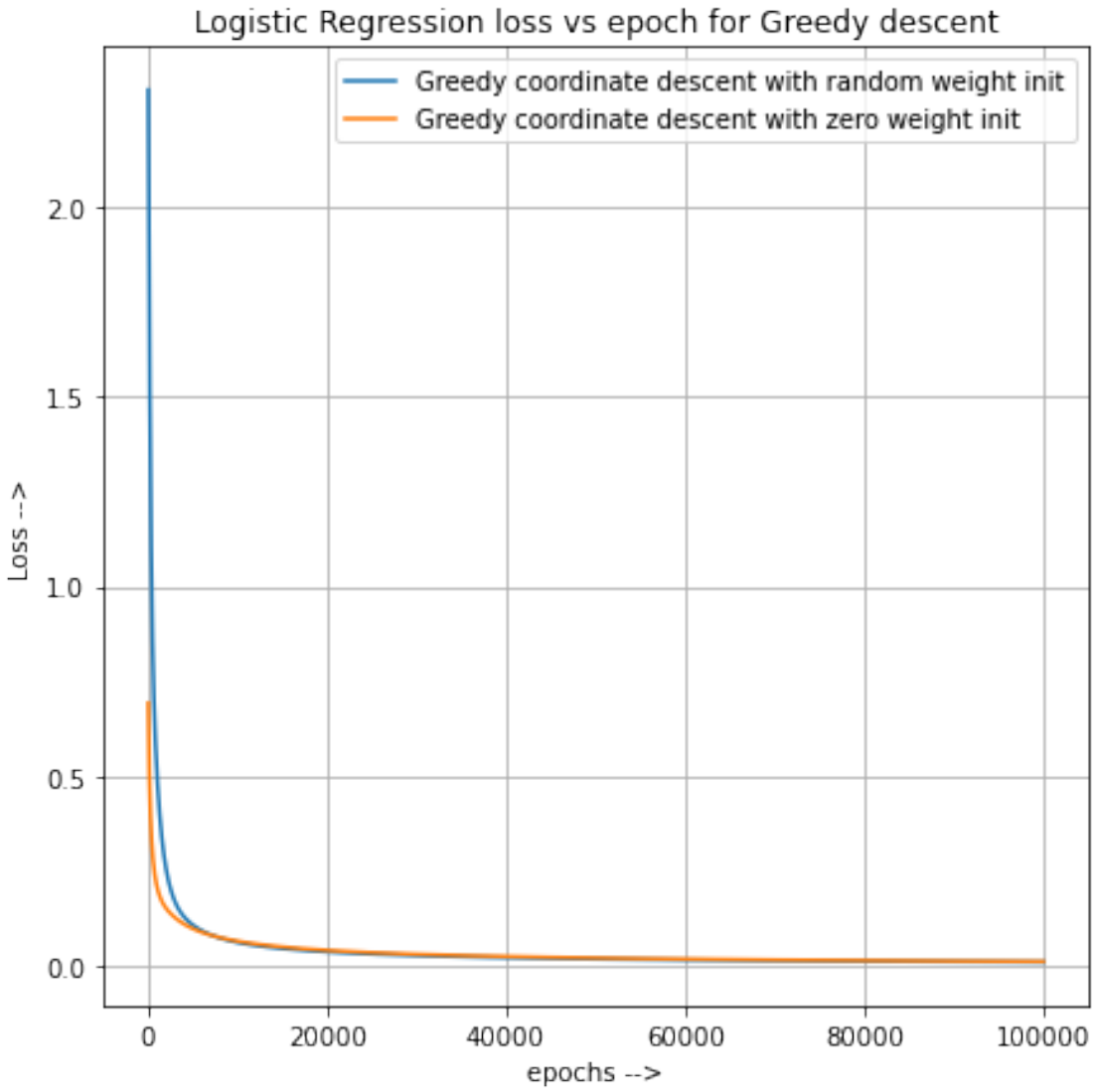
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       1.00      1.00      1.00        21

    accuracy                           1.00        43
   macro avg       1.00      1.00      1.00        43
weighted avg       1.00      1.00      1.00        43
```

[39]:
```
epochs=eps
weights=np.zeros((xtrain.shape[1],1))
loss_CD_zw=[]
for epoch in range(epochs):
    loss=Lossfunction(weights,xtrain,ytrain)
    loss_CD_zw.append(loss)
    weights=WeightUpdateCD(weights,xtrain,ytrain,stepsize=0.01)
    ypred= Predict(weights,xtest)
    #print(loss)
ypred= Predict(weights,xtest)
print(classification_report(ytest,ypred))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       1.00      0.95      0.98        21

    accuracy                           0.98        43
   macro avg       0.98      0.98      0.98        43
weighted avg       0.98      0.98      0.98        43
```

[40]:
```
fig = plt.figure(1)
fig.set_figheight(7)
fig.set_figwidth(7)
ax1= fig.add_subplot(1,1,1)
ax1.title.set_text('Logistic Regression loss vs epoch for Greedy descent')
plt.xlabel('epochs -->')
plt.ylabel('Loss -->')
curve1,=ax1.plot(loss_CD_rw)
curve2,=ax1.plot(loss_CD_zw)
plt.grid(visible='True')
curve1.set_label('Greedy coordinate descent with random weight init')
curve2.set_label('Greedy coordinate descent with zero weight init')
plt.legend()
```

[40]: <matplotlib.legend.Legend at 0x2df19ca5f70>



4(c). Logistic regression using Greedy coordinate descent for different learning rates

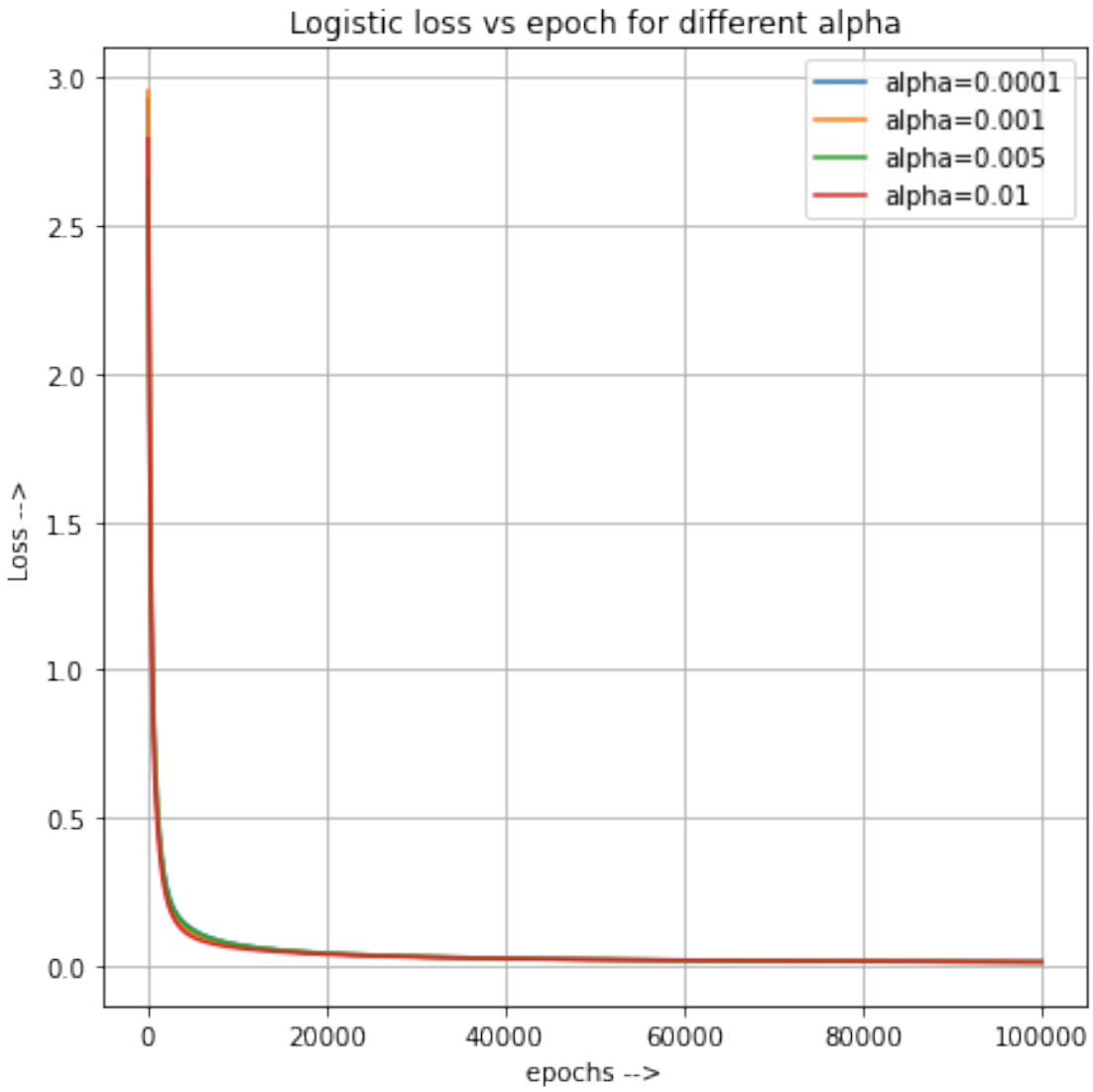
[41]:
```
epochs=eps
step_sizes=[0.0001,0.001,0.005,0.01]
loss_alpha={}
seed=np.random.seed(1992)
for alpha in step_sizes:
    weights=np.random.rand(xtrain.shape[1],1)
    loss_CD_rw=[]
    for epoch in range(epochs):
        loss=Lossfunction(weights,xtrain,ytrain)
```

```
        loss_CD_rw.append(loss)
        weights=WeightUpdateCD(weights,xtrain,ytrain,stepsize=0.01)
        ypred= Predict(weights,xtest)
    ypred= Predict(weights,xtest)
    acc=100*accuracy_score(ytest,ypred)
    print("Test accuracy for Greedy Coordinate descent with alpha ={0} : {1}%".
↪format(alpha,acc))
    loss_alpha[alpha]=loss_CD_rw
```

```
Test accuracy for Greedy Coordinate descent with alpha =0.0001 :
97.67441860465115%
Test accuracy for Greedy Coordinate descent with alpha =0.001 :
97.67441860465115%
Test accuracy for Greedy Coordinate descent with alpha =0.005 : 100.0%
Test accuracy for Greedy Coordinate descent with alpha =0.01 : 100.0%
```

[57]:
```
fig = plt.figure(1)
fig.set_figheight(7)
fig.set_figwidth(7)
ax1= fig.add_subplot(1,1,1)
ax1.title.set_text('Logistic loss vs epoch for different alpha')
plt.xlabel('epochs -->')
plt.ylabel('Loss -->')
for alpha in step_sizes:
    curve1,=ax1.plot(loss_alpha[alpha])
    plt.grid(visible='True')
    curve1.set_label('alpha={0}'.format(alpha))
plt.legend()
```

[57]: <matplotlib.legend.Legend at 0x2df147bdd90>



4(d). Logistic regression convergence comparison for different epochs

[58]:
```
weights=np.random.rand(xtrain.shape[1],1)
loss_GD=[]
diff=999
prev_weights=np.random.rand(xtrain.shape[1],1)
iters=0
max_iters=200000
while(diff>=0.000001 and iters<max_iters):
    loss=Lossfunction(weights,xtrain,ytrain)
    loss_GD.append(loss)
    weights=WeightUpdateGD(weights,xtrain,ytrain,0.01)
    diff=np.mean(np.abs((weights-prev_weights)))
```

```
    prev_weights=np.copy(weights)
    ypred= Predict(weights,xtest)
    acc=100*accuracy_score(ytest,ypred)
    iters+=1
ypred= Predict(weights,xtest)
print(classification_report(ytest,ypred))
print("Final loss= {0}".format(loss))
print("required epochs for Gradient descent={0}".format(iters))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       1.00      0.95      0.98        21

    accuracy                           0.98        43
   macro avg       0.98      0.98      0.98        43
weighted avg       0.98      0.98      0.98        43

Final loss= 0.0012846981636210398
required epochs for Gradient descent=200000
```

[59]:
```
weights=np.random.rand(xtrain.shape[1],1)
loss_RCD=[]
diff=999
iters=0
prev_weights=np.random.rand(xtrain.shape[1],1)
loss=100
max_iters=200000
while(loss>=0.0008 and iters<max_iters):
    loss=Lossfunction(weights,xtrain,ytrain)
    loss_RCD.append(loss)
    weights=WeightUpdateRCD(weights,xtrain,ytrain,stepsize=0.01)
    diff=np.mean(np.abs((weights-prev_weights)))
    prev_weights=np.copy(weights)
    ypred= Predict(weights,xtest)
    iters+=1
    #print(loss)
ypred= Predict(weights,xtest)
print(classification_report(ytest,ypred))
print("Final loss= {0}".format(loss))
print("required epochs for random feature selection coordinate descent={0}".
 ↪format(iters))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       1.00      0.95      0.98        21
```

```
    accuracy                           0.98        43
   macro avg       0.98      0.98      0.98        43
weighted avg       0.98      0.98      0.98        43

Final loss= 0.012864027918244304
required epochs for random feature selection coordinate descent=200000
```

[60]:
```
weights=np.random.rand(xtrain.shape[1],1)
loss_CD=[]
diff=999
iters=0
loss=100
prev_weights=np.random.rand(xtrain.shape[1],1)
max_iters=200000
while(loss>=0.0008 and iters<max_iters):
    loss=Lossfunction(weights,xtrain,ytrain)
    loss_CD.append(loss)
    weights=WeightUpdateCD(weights,xtrain,ytrain,stepsize=0.01)
    diff=np.mean(np.abs((weights-prev_weights)))
    prev_weights=np.copy(weights)
    ypred= Predict(weights,xtest)
    iters+=1
    #print(loss)
ypred= Predict(weights,xtest)
print(classification_report(ytest,ypred))
print("Final loss= {0}".format(loss))
print("required epochs for greedy coordinate descent={0}".format(iters))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        22
           1       1.00      0.95      0.98        21

    accuracy                           0.98        43
   macro avg       0.98      0.98      0.98        43
weighted avg       0.98      0.98      0.98        43

Final loss= 0.007495119175856397
required epochs for greedy coordinate descent=200000
```