Coordinate Descent

Abstract

Coordinate descent is a simple yet effective optimization method used to minimize a multivariate objective function. It is particularly efficient in high-dimensional problems where updating the entire set of features weights simultaneously may be computationally expensive. Through this project, we explore how each step in coordinate descent can itself be modelled as an optimization problem and devise effective strategies in this regard.

1 Introduction

Coordinate descent is an optimization problem where at each step, the update is performed only in the perspective of one coordinate or feature. This is repeated iteratively until convergence of the loss. To study methods of efficient coordinate selection and update for coordinate descent through the course of this paper, we use the $wine$ dataset. Since the objective is to study coordinate descent for binary classification, we consider the data corresponding to classes 1 and 2 in the dataset and map them to binary labels. This dataset then contains 130 training samples with each of the points having 13 features. With this data we explore selection and descent algorithms that effectively choose a coordinate in each iteration and perform the optimum descent. We generate baselines through random selection and compare the performance to provide a critical evaluation of the models.

2 Coordinate Descent Method

In this section, we first discuss how to update the weight for a given coordinate in an iteration and then proceed to discuss the strategy to select the coordinate for update in each iteration.

2.1 Weight Update Strategy

Let $L(.)$ denote the loss function and w be the weights vector where w_j denotes the weight corresponding to the j^{th} coordinate. Given a fixed index j, we design the weight update strategy such that the update in w_j maximizes the decrease in loss function w.r.t w_i. Consider a differential loss function $L(.)$, the update to w_j can be performed with a suitable step size in the gradient descent direction as follows.

$$w_j \leftarrow w_j - \alpha \frac{\partial L(w)}{w_j}$$

The step size α can be chosen as a suitable constant or linearly decreasing or can be obtained by performing an exact line search. For the purpose of this project, we choose a fixed learning rate and then proceed to use higher order information to do away with this hard coding of learning rate. If the loss function $L(.)$ is doubly differentiable and its Hessian is invertible, the appropriate update can be computed using the Newton's descent method

For our case of binary classification problem, we choose the loss function to be the log-loss. Let $x^{(i)}$ denote the feature vector for one data entry (with intercept term added), and $y^{(i)}$ denote it's true label. Let $p^{(i)}$ denote the predicted label. The log loss with weights vector w and N data points is then given by:

$$LL(w) = -\frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)}log(p^{(i)}) + (1-y^{(i)})log(1-p^{(i)})\right)$$

where

$$p^{(i)} = \sigma(w^T x^{(i)}) = \frac{1}{1+e^{-w^T x^{(i)}}}$$

Since the log-loss is a differential function, for maximum update for a coordinate j, we set the partial derivative of of above log loss w.r.t w_j to 0.

$$\frac{\partial LL(w)}{w_j} = -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y^{(i)}}{p^{(i)}} - \frac{(1-y^{(i)})}{(1-p^{(i)})}\right)\frac{\partial p^{(i)}}{w_j}$$

$$\frac{\partial LL(w)}{w_j} = -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{(y^{(i)}-p^{(i)})}{p^{(i)}(1-p^{(i)})}\right)\frac{\partial p^{(i)}}{w_j}$$

where

$$\frac{\partial p^{(i)}}{w_j} = \frac{\partial}{\partial w_j}\sigma(w^T x^{(i)}$$

$$\frac{\partial p^{(i)}}{w_j} = \sigma(w^T x^{(i)})(1-\sigma(w^T x^{(i)}))\frac{\partial(w^T x^{(i)})}{\partial w_j}$$

$$\frac{\partial p^{(i)}}{w_j} = p^{(i)}(1-p^{(i)})x_j^{(i)}$$

Therefore,

$$\frac{\partial LL(w)}{w_j} = \frac{1}{N}\sum_{i=1}^{N}(p^{(i)} - y^{(i)})x_j^{(i)}$$

w_j is then updated as:

$$w_j \leftarrow w_j - \alpha \frac{\partial LL(w)}{w_j}$$

From the first order derivative of the log loss function, we observe that the loss function is doubly differentiable whose Hessian entries are obtained as:

$$\frac{\partial^2}{\partial w_k \partial w_j} LL(w) = \frac{1}{N} \sum_{i=1}^{N} p^{(i)}(1-p^{(i)}) x_j^{(i)} x_k^{(i)}$$

For the dataset X, the Hessian of log loss can therefore be represented as:

$$H = X^T D X$$

where D is the diagonal matrix with entries

$$D_{ii} = \frac{1}{N} p^{(i)}(1-p^{(i)})$$

This Hessian is positive semi-definite since the diagonal entries on D are greater than or equal to 0 (since $0 \leq p^{(i)} \leq 1$). The magnitude of update for weight of the j coordinate w_j can thus be obtained from the j^{th} entry in the $H^{-1}g$ vector where g is the gradient vector computed as outlined above.

$$w_j \leftarrow w_j - (H^{-1}g)[j]$$

Algorithm 1 Coordinate Descent (1)

$X \rightarrow$ input data with intercept term added
$y \rightarrow$ true labels for input data
$w \rightarrow$ weights initialized from standard normal
$\alpha \rightarrow$ learning rate

For $iteration$ in max_iter:

- Compute $y_{prob} = \sigma(w^T X)$
- Gradient $g = X^T.(y_{prob} - y)/n\_samples$
- Pick co-ordinate j (random/argmax of g)
- Update $w_j \leftarrow w_j - \alpha g[j]$
- Compute and report log_loss

2.2 Coordinate Selection

For both the methods outlined for update in section 2.1 (gradient descent and Newton descent), we now define a strategy to pick which coordinate to update in each iteration. We design the selection such that at each iteration, the coordinate that provides the maximum numerical update in corresponding weight is chosen. This is because intuitively greater the update in weight for a feature, the more likely the prediction will differ from the previously iteration and potentially lead to improvement in model performance. Also since we are performing descent methods on loss function w.r.t each of the weights, maximum possible decrease in loss value is expected by this selection method.

For the gradient descent method, we compute partial derivatives of the loss function with the weight vector and choose the coordinate with maximum absolute value of the partial derivative i.e j is chosen as:

$$j = argmax_k \frac{\partial LL(w)}{\partial w_k}$$

For update using the Newton method, we compute the $H^{-1}g$ vector and choose the coordinate as the index which has the maximum absolute value in the vector.

$$j = argmax_k (H^{-1}g)[k]$$

To compare the quality of the selection methods outlined above, we also perform an update by picking a random coordinate at each iteration for both the methods. The results are depicted in section 3.

Algorithm 2 Coordinate Descent (2)

$X \rightarrow$ input data with intercept term added
$y \rightarrow$ true labels for input data
$w \rightarrow$ weights initialized from standard normal

For $iteration$ in max_iter:

- Compute $y_{prob} = \sigma(w^T X)$
- Gradient $g = X^T.(y_{prob} - y)/n\_samples$
- Compute the Hessian H
- Pick co-ordinate j (random/argmax of $H^{-1}g$)
- Update $w_j \leftarrow w_j - (H^{-1}g)[j]$
- Compute and report log_loss

3 Experimental Results

We first standardize the data by by removing the mean and scaling to unit variance.To get a reference optimum loss value for evaluating the model performance, we fit a logistic regression model without regularization on the data. The training loss L^*(log loss) for this model is observed to be $7.29e-07$.

We then implement the first coordinate descent method outlined in 2.1 where gradient descent is used for optimizing loss w.r.t the chosen coordinate and the coordinate in accordance with the gradient values in an iteration. Initial weights are sampled from a normal distribution with mean 0 and standard deviation 0.01. The learning rate is set to 0.1 and log loss value across $2e5$ iterations are observed. The final log-loss is observed to be $1.03-e5$.To show that this selection strategy is credible, we also perform random selection with the same learning rate and number of iterations. The log-loss at the end of iterations for random selection is $1.98-e4$. The comparative analysis across iterations is shown in Figure 1 and 2. (The plots for iterations 0 to 1000 and 1000 to $1e5$ to clearly depict convergence rates.)

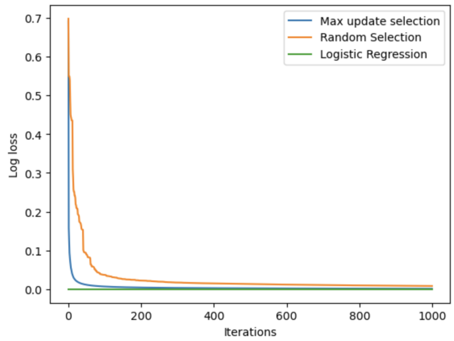


Figure 1: Log loss vs iterations - method 1

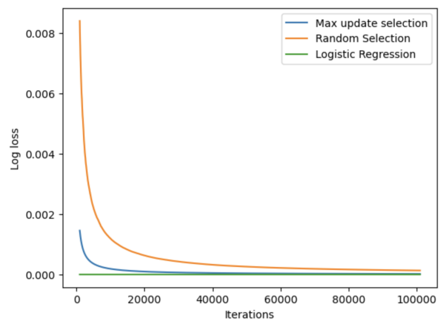


Figure 2: Log loss vs iterations - method 1

We further implement the second coordinate descent method outlined in 2.1 where second order information is used for optimizing loss w.r.t the chosen coordinate and the coordinate in accordance with the $H^{-1}g$ values in an iteration. Initial weights are set to small values sampled from a normal distribution with mean 0 and standard deviation 0.01. The performance across iterations is observed and the final log loss is $6.83e-5$. To show that this selection strategy is better, we also perform random selection for the number of iterations. The log-loss at the end of iterations for random selection is 0.317. It is observed that random selection does not perform well with this second order design and converges to value much higher than our selection method. The comparative analysis across iterations is shown in Figure 3.

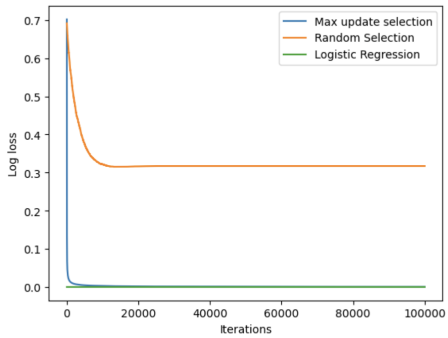


Figure 3: Log loss vs iterations - method 2

4 Convergence

It is observed from the experimental results that the log loss for our selection strategy in both the methods converges asymptotically to the log loss observed in logistic regression. In the first order method although random selection converges close to the logistic log loss, the selection strategy ensures converges to a lower value at a faster rate. In the second order method, the selection strategy ensures convergence to the optimum as opposed to random selection where the log loss saturates at a high value. Therefore, after experimentation it is observed that the rate of convergence value converging value depend on the order of coordinates selected for update and are optimum when selection is ordered by magnitude of update. Also, the convergence is more sensitive to the selection strategy when higher order information is used to compute the descent. Also, since the optimization is performed only from the perspective of one coordinate at a time, a consistent behaviour of convergence is more likely if the loss function is convex as is the case with the log loss function used in the project.

5 Critical Evaluation

The coordinate selection and descent method outlined in this project converge to reasonably low log loss value. However, the is scope of improvement further experiments can be conducted can conducted in this regard. For instance in the first order method, we choose a fixed learning rate α after selecting coordinate with maximum gradient value. However, exact line search can be performed to determine more accurately the the coordinate resulting in maximum decrease in loss as well as the step size for such a decrease. Although this method may be more accurate, it is computationally expensive. The selection of the model design is therefore a trade off between computational cost and degree of accuracy. Also, both the first order and second order methods discussed are only applicable when the chosen loss function is differentiable and doubly-differentiable respectively. Hence, there is a scope for more efficient and universal design for a different choice of a loss function.

In [478…
```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
from sklearn.preprocessing import MinMaxScaler,StandardScaler
import numpy as np
import math
from ucimlrepo import fetch_ucirepo
from scipy.optimize import line_search
import matplotlib.pyplot as plt
```

In [469…
```
wine = fetch_ucirepo(id=109)

X = wine.data.features
y = wine.data.targets
```

In [470…
```
X = X[:130]
y = y[:130]
y['class'].value_counts()
```

Out[470]:
```
class
2    71
1    59
Name: count, dtype: int64
```

In [471…
```
#scaler = MinMaxScaler()
scaler = StandardScaler()

df_norm = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)
df_norm.describe()
```

Out[471]:

	Alcohol	Malicacid	Ash	Alcalinity_of_ash	Magnesium	Total_phenols	
count	1.300000e+02	1.300000e+02	1.300000e+02	1.300000e+02	1.300000e+02	1.300000e+02	1.3
mean	7.651999e-16	1.708035e-17	2.732857e-16	1.639714e-16	-3.825999e-16	-1.639714e-16	2.
std	1.003868e+00	1.003868e+00	1.003868e+00	1.003868e+00	1.003868e+00	1.003868e+00	1.0
min	-2.161957e+00	-1.400991e+00	-3.312182e+00	-2.416453e+00	-1.951429e+00	-2.618653e+00	-2.6
25%	-8.093764e-01	-5.225744e-01	-5.758509e-01	-6.673009e-01	-7.766557e-01	-6.905910e-01	-6.6
50%	7.446037e-02	-2.715983e-01	-6.912290e-02	-6.948922e-02	-1.240038e-01	6.866973e-02	1.0
75%	8.848793e-01	1.020139e-01	6.318509e-01	6.390283e-01	5.286480e-01	7.635083e-01	6.9
max	2.130157e+00	4.371460e+00	3.005027e+00	3.310730e+00	4.052968e+00	2.498304e+00	3.5

In [472…
```
X = df_norm.to_numpy()
y = np.array([i-1 for i in list(y['class'])])
```

In [473…
```
model = LogisticRegression(penalty = None,max_iter=200).fit(X,y)
y_prob = model.predict_proba(X)
loss_lreg = log_loss(y, y_prob)
loss_lreg
```

Out[473]:
```
7.292437743938115e-07
```

In [474…
```
def logistic(y):
    return 1.0/(1+np.exp(-y))

def logloss(X,w,y):
```

```
    y_prob = logistic(np.dot(X,w))
    return log_loss(y, y_prob)
```

In [475…

```
def hessian(X,y_prob):
    return np.matmul(np.matmul(X.T,np.diag(y_prob)),X)
```

In [476…

```
def coordinate_descent_gradient(X,y,max_iter,lr=.1):
    X = np.insert(X,0,1,axis=1)
    n_samples = X.shape[0]
    n_features = X.shape[1]
    weights = np.random.normal(0, 0.01, n_features)
    initial_loss = logloss(X,weights,y)
    ls = [initial_loss]
    print("Iteration: 0" + "   loss: "+str(initial_loss))
    for i in range(max_iter):
        y_prob = logistic(np.dot(X,weights))
        grad = np.dot(X.T,(y_prob-y))#/n_samples
        pick = np.argmax(np.abs(grad))
        weights[pick] -= lr*grad[pick]
        loss = logloss(X,weights,y)
        ls.append(loss)
        if((i+1)%10000==0):
            print("Iteration: "+ str(i+1) + "   loss: "+str(loss))
        #print(weights)
    return ls

#coordinate_descent_gradient(X,y,)
loss1s = coordinate_descent_gradient(X,y,200000)
```

```
Iteration: 0   loss: 0.6903904603886847
Iteration: 10000   loss: 0.00018854732457440748
Iteration: 20000   loss: 9.734707455094677e-05
Iteration: 30000   loss: 6.58356747070647e-05
Iteration: 40000   loss: 4.980841648981822e-05
Iteration: 50000   loss: 4.008866283634282e-05
Iteration: 60000   loss: 3.35596095926181e-05
Iteration: 70000   loss: 2.8869116232329535e-05
Iteration: 80000   loss: 2.5335113910391005e-05
Iteration: 90000   loss: 2.257607786351572e-05
Iteration: 100000   loss: 2.0361828921225497e-05
Iteration: 110000   loss: 1.8545217197030153e-05
Iteration: 120000   loss: 1.702776040368502e-05
Iteration: 130000   loss: 1.5741048647411e-05
Iteration: 140000   loss: 1.4636073261160829e-05
Iteration: 150000   loss: 1.367680028495526e-05
Iteration: 160000   loss: 1.283614067466917 6e-05
Iteration: 170000   loss: 1.2093335300771136e-05
Iteration: 180000   loss: 1.1432206959812491e-05
Iteration: 190000   loss: 1.0839961556357441e-05
Iteration: 200000   loss: 1.0306347164872382e-05
```

In [498…

```
def coordinate_descent_random_gradient(X,y,max_iter,lr=.1):
    X = np.insert(X,0,1,axis=1)
    n_samples = X.shape[0]
    n_features = X.shape[1]
    weights = np.random.normal(0, 0.01, n_features)
    initial_loss = logloss(X,weights,y)
    ls = [initial_loss]
    print("Iteration: 0" + "   loss: "+str(initial_loss))
    for i in range(max_iter):
        y_prob = logistic(np.dot(X,weights))
        grad = np.dot(X.T,(y_prob-y))#/n_samples
        pick = np.random.randint(0, 13)
        weights[pick] -= lr*grad[pick]
        loss = logloss(X,weights,y)
```

```
        ls.append(loss)
        if((i+1)%10000==0):
            print("Iteration: "+ str(i+1) + "   loss: "+str(loss))
    return ls

loss1r = coordinate_descent_random_gradient(X,y,200000)
```

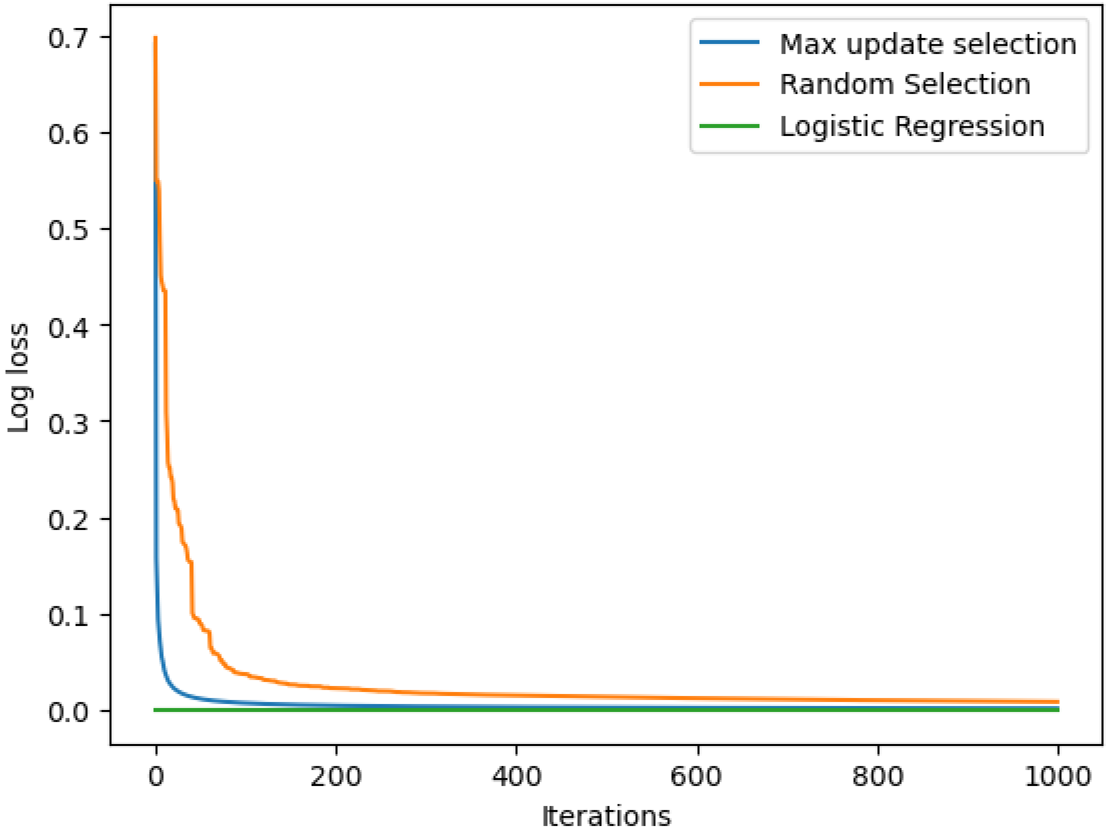
```
Iteration: 0   loss: 0.6971138504294091
Iteration: 10000   loss: 0.0011934181297882263
Iteration: 20000   loss: 0.0006310008130747647
Iteration: 30000   loss: 0.0004297510687675009
Iteration: 40000   loss: 0.0003227398015914253
Iteration: 50000   loss: 0.00026030449853506777
Iteration: 60000   loss: 0.0002181882535222798
Iteration: 70000   loss: 0.0001883242600103548
Iteration: 80000   loss: 0.00016566028925278477
Iteration: 90000   loss: 0.00014801165778837336
Iteration: 100000   loss: 0.0001338638662150378
Iteration: 110000   loss: 0.0001221032658136578
Iteration: 120000   loss: 0.0001122567221525208
Iteration: 130000   loss: 0.00010370774317911805
Iteration: 140000   loss: 9.631099610261816e-05
Iteration: 150000   loss: 8.997952721573228e-05
Iteration: 160000   loss: 8.453116860916722e-05
Iteration: 170000   loss: 7.962259483470823e-05
Iteration: 180000   loss: 7.540252341931665e-05
Iteration: 190000   loss: 7.156985410463349e-05
Iteration: 200000   loss: 6.81038624080378e-05
```

In [526…

```
n = 1000
start = 0
x = [i+start for i in range(n)]
lreg = [loss_lreg for i in range(n)]
plt.plot(x,loss1s[start:n+start], label = "Max update selection")
plt.plot(x,loss1r[start:n+start], label = "Random Selection")
plt.plot(x,lreg[:n], label = "Logistic Regression")

plt.xlabel("Iterations")
plt.ylabel("Log loss")

plt.legend()
plt.show()
```

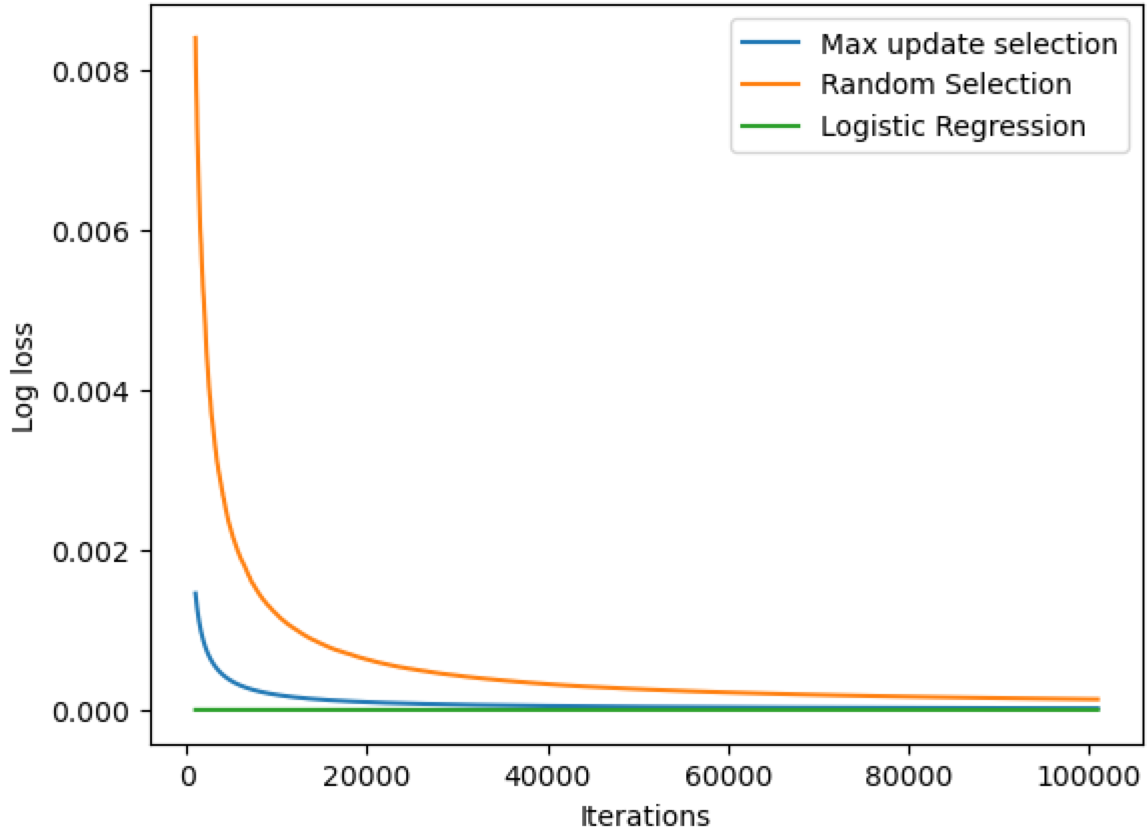


In [527…

```
n = 100000
start = 1000
x = [i+start for i in range(n)]
lreg = [loss_lreg for i in range(n)]
plt.plot(x,loss1s[start:n+start], label = "Max update selection")
plt.plot(x,loss1r[start:n+start], label = "Random Selection")
plt.plot(x,lreg[:n], label = "Logistic Regression")

plt.xlabel("Iterations")
plt.ylabel("Log loss")

plt.legend()
plt.show()
```



In [518…

```
def coordinate_descent_hessian(X,y,max_iter):
    X = np.insert(X,0,1,axis=1)
    n_samples = X.shape[0]
    n_features = X.shape[1]
    weights = np.random.normal(0, 0.01, n_features)
    initial_loss = logloss(X,weights,y)
    ls = [initial_loss]
    print("Iteration: 0" + "    loss: "+str(initial_loss))
    for i in range(max_iter):
        y_prob = logistic(np.dot(X,weights))

        grad = np.dot(X.T,(y_prob-y))#/n_samples
        H = (hessian(X,y_prob))
        H_inv = np.linalg.inv(H)

        delta = np.matmul(H_inv,grad)
        pick = np.argmax(np.abs(delta))
        #pick = np.argmax(np.abs(grad))

        weights[pick] -= delta[pick]
        loss = logloss(X,weights,y)
        ls.append(loss)

        if((i+1)%10000==0):
            print("Iteration: "+ str(i+1) + "    loss: "+str(loss))
    return ls

loss2s = coordinate_descent_hessian(X,y,200000)
```

```
Iteration: 0    loss: 0.7017863340494186
Iteration: 10000    loss: 0.0027498909207778486
Iteration: 20000    loss: 0.0015844752332797526
Iteration: 30000    loss: 0.0011213378633768215
Iteration: 40000    loss: 0.0008709421571232945
Iteration: 50000    loss: 0.0007135448621648047
Iteration: 60000    loss: 0.000605236313882924
Iteration: 70000    loss: 0.00052602698425397
Iteration: 80000    loss: 0.000465544031636318
```

```
Iteration: 90000   loss: 0.00041777390259912085
Iteration: 100000   loss: 0.0003790970111569904
Iteration: 110000   loss: 0.0003471119941356405
Iteration: 120000   loss: 0.0003202153606048875
Iteration: 130000   loss: 0.0002972687711021797
Iteration: 140000   loss: 0.0002774619953674894
Iteration: 150000   loss: 0.0002601772434921201
Iteration: 160000   loss: 0.00024497049505741884
Iteration: 170000   loss: 0.00023147699964640086
Iteration: 180000   loss: 0.00021942481893382726
Iteration: 190000   loss: 0.00020859394738823473
Iteration: 200000   loss: 0.00019880183760410223
```

In [521…

```
def coordinate_descent_random_hessian(X,y,max_iter):
    X = np.insert(X,0,1,axis=1)
    n_samples = X.shape[0]
    n_features = X.shape[1]
    weights = np.random.normal(0, 0.001, n_features)
    initial_loss = logloss(X,weights,y)
    ls = [initial_loss]
    print("Iteration: 0" + "   loss: "+str(initial_loss))
    for i in range(max_iter):
        y_prob = logistic(np.dot(X,weights))

        grad = np.dot(X.T,(y_prob-y))/n_samples
        H = (hessian(X,y_prob))
        H_inv = np.linalg.inv(H)

        delta = np.matmul(H_inv,grad)
        pick = np.random.randint(0, 13)
        #print(pick)
        #pick = np.argmax(np.abs(delta))

        weights[pick] -= delta[pick]
        loss = logloss(X,weights,y)
        ls.append(loss)

        if((i+1)%10000==0):
            print("Iteration: "+ str(i+1) + "   loss: "+str(loss))
    return ls

loss2r = coordinate_descent_random_hessian(X,y,200000)
```

```
Iteration: 0   loss: 0.691557503663448
Iteration: 10000   loss: 0.32168843239672995
Iteration: 20000   loss: 0.31631499610803193
Iteration: 30000   loss: 0.3174075101235929
Iteration: 40000   loss: 0.31738904181189115
Iteration: 50000   loss: 0.3172671650175636
Iteration: 60000   loss: 0.31722294791055533
Iteration: 70000   loss: 0.3172052958045988
Iteration: 80000   loss: 0.31719880244312537
Iteration: 90000   loss: 0.3171962389483934
Iteration: 100000   loss: 0.3171952866487816
Iteration: 110000   loss: 0.3171949445027117
Iteration: 120000   loss: 0.3171948404498702
Iteration: 130000   loss: 0.31719480666039485
Iteration: 140000   loss: 0.3171947958329806
Iteration: 150000   loss: 0.3171947908788463
Iteration: 160000   loss: 0.3171947892840135
Iteration: 170000   loss: 0.3171947887699575
Iteration: 180000   loss: 0.31719478858771843
Iteration: 190000   loss: 0.31719478851677796
Iteration: 200000   loss: 0.31719478849299515
```

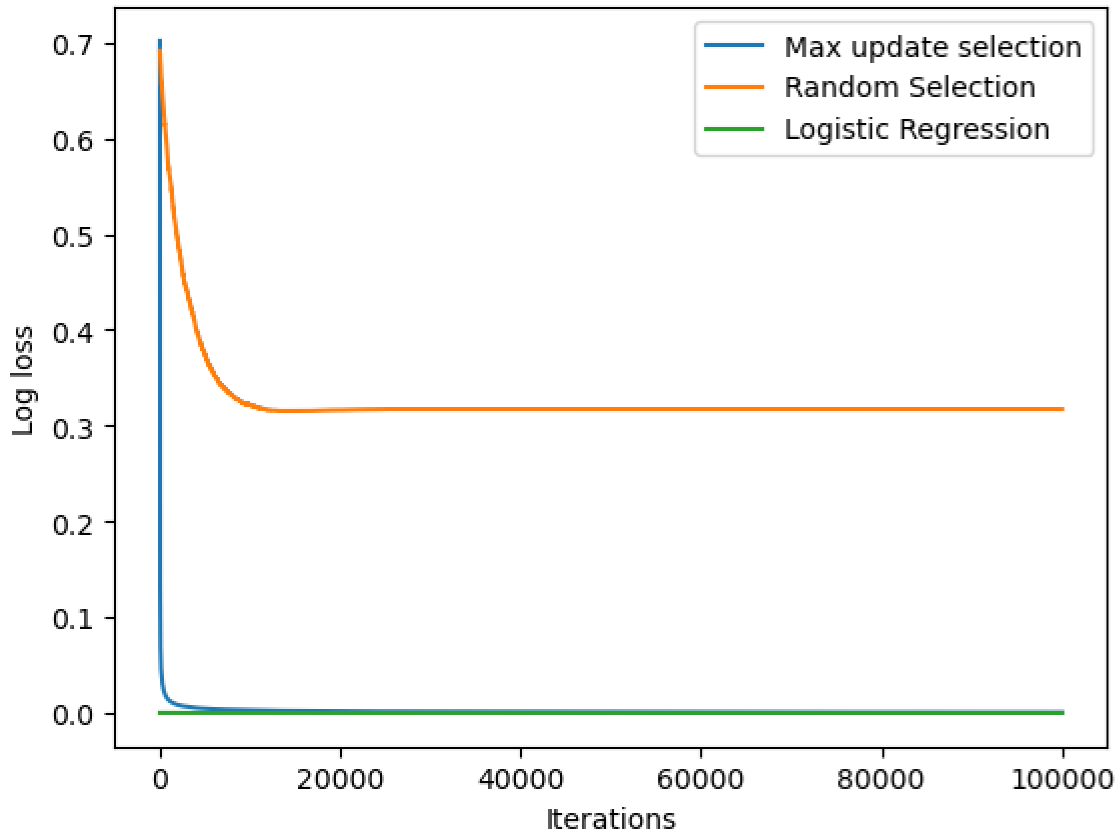
In [528…

```
n = 100000
```

```python
start = 0
x = [i+start for i in range(n)]
lreg = [loss_lreg for i in range(n)]
plt.plot(x,loss2s[start:n+start], label = "Max update selection")
plt.plot(x,loss2r[start:n+start], label = "Random Selection")
plt.plot(x,lreg[:n], label = "Logistic Regression")

plt.xlabel("Iterations")
plt.ylabel("Log loss")

plt.legend()
plt.show()
```




```
In [ ]:
```