Michaelmas Term 2022

Optimisation for Regression and Classification Models

Suyash Agarwal

1 Introduction

This project is an investigation into computational optimisation methods for learning the optimal parameters of a model. The aim is to find the parameters which minimise the error in regression or classification over a training dataset, and then use the same parameters to make certain predictions for given data points. This is the basis for machine learning.

This report begins by optimising a linear regression model in Sections 2 and 3. Sections 4-7 will focus on optimising a classification model, through both logistic regression and support vector machines. Throughout the report, different optimisation methods will be utilised including a closed-form analytical solution, gradient descent (GD), stochastic gradient descent (SGD) and linear programming.

2 Linear regression using analytical solution

2.1 Motivation and theory

In this section, a closed-form analytical solution to the optimisation problem is used. The linear regression model is $\bar{\boldsymbol{y}} = \hat{\boldsymbol{X}}\boldsymbol{\theta}$ and the objective function which the optimisation methods seek to minimise is mean-squared error (MSE),

$$J_{mse}(\boldsymbol{\theta}, \hat{\boldsymbol{X}}, \boldsymbol{y}) = \frac{1}{n}(\hat{\boldsymbol{X}}\boldsymbol{\theta} - \boldsymbol{y})^T(\hat{\boldsymbol{X}}\boldsymbol{\theta} - \boldsymbol{y}). \tag{1}$$

The desired model parameters, $\boldsymbol{\theta}*$, are found by setting the derivative of the objective function to 0, minimising the MSE. An analytical solution can be found by solving the following system of linear equations

$$\boldsymbol{\theta}* = (\hat{\boldsymbol{X}}^{\boldsymbol{T}}\hat{\boldsymbol{X}})^{-1}\hat{\boldsymbol{X}}^{\boldsymbol{T}}\boldsymbol{y}. \tag{2}$$

2.2 Methodology and MATLAB Implementation

The training and test datasets were created using the $create\_data$ function, which randomly samples each x from one of two different 2-dimensional Gaussian distributions, each with different parameters. The implementation was straightforward as $\boldsymbol{\theta}*$ was calculated in a single line of code. When running the experiments to see how changing the seed or number of training samples affects the error, for loops were added to collect all the relevant data at once and simply output the relevant summary statistics, i.e. the mean and standard deviation of the error for each number of training samples.

2.3 Results and Discussion

The model parameters computed using 1000 training samples were $\boldsymbol{\theta}* = (0.3083, 0.6894, 1.9942)$. These values are expected, as in $create\_data$, the generated regression targets were $(0.3, 0.7, 2)$ for the weights and bias respectively. The MSE was 0.0418 with 1000 samples. Reducing the number of training samples to 100 increased the MSE to 0.0492, which was expected as the model had less training data to work with to learn the regression parameters. However, increasing the number of training samples to 10,000 also increased the MSE. This was unexpected, however, trying the same experiment with a different random number generator (RNG) seed yielded the expected trend. This suggests that there is a certain random dependence between the MSE and the seed, which justifies the later experiment where variation in MSE over different seeds is investigated.

With 10 training samples and a seed of 12345, the model parameters found were $\boldsymbol{\theta}* = (0.4741, 0.6992, 1.7319)$. The reason they are further from the regression targets than the parameters found using 1000 training samples even though the function finds the global optimum is that $\boldsymbol{\theta}*$ depends directly on the training data $\hat{\boldsymbol{X}}$ and $\boldsymbol{y}$ used (as per Eq.2), which are randomly sampled from the 2-dimensional Gaussians that the model is trying to predict. Taking more samples results in a more accurate depiction of these distributions, which gives model parameters closer to the regression targets.

This trend is investigated in Table 1. Since the RNG seed was shown to affect the MSE achieved, this effect was removed by using 10 different random seeds and reporting the mean and standard deviation (SD) over these seeds. As expected, the mean-squared error present in the test data decreases as the number of training samples increases. Also, the standard deviation in mean-squared error generally decreases as the testing dataset increases in size. This is because increasing the number of data points reduces the effect that the stochastic nature of individual data points has on the outcome MSE. So, with a larger test set, there are more consistent results (i.e. less noise/variance) for the MSE. Furthermore, increasing the number of training points makes it more difficult to overfit the data. Overfitting would mean that the MSE over training samples is very small, as the model seeks to match the training data too accurately – it tracks the noise as well as

		Number of training samples used					
		4	10	20	100	1000	10000
Mean ± SD of MSE tested on:	Training samples	0.0162 ± 0.0182	0.0347 ± 0.0099	0.0340 ± 0.0122	0.0420 ± 0.0067	0.0392 ± 0.0014	0.0398 ± 0.0006
	Test samples	0.0970 ± 0.0544	0.0553 ± 0.0164	0.0455 ± 0.0055	0.0408 ± 0.0019	0.0398 ± 0.0017	0.0394 ± 0.0019

Table 1: Variation of MSE with number of training samples used

the underlying trend. This would lead to much higher MSE over test samples, as the model is too specific to the training samples. This also helps to explain the trends seen in the table above, as the discrepancy between training and test error decreases as we increase the number of training samples.

Finally, the effect of the size of the training dataset on runtime is investigated in Table 2. Once again, the results are averaged over different RNG seeds. There is a general increase in runtime as the number of training samples increases because the calculation of $\boldsymbol{\theta}*$ is a very computationally expensive step, as it involves inverting a matrix product. The size of the matrices involved in the computation of Eq.2 is directly proportional to the number of training samples used, so the runtime and memory requirements will increase rapidly as the training dataset increases. The exact time and space complexities of the operation will depend on the implementation used by MATLAB, however the data in Table 2 justifies the need for alternative optimisation algorithms that don't rely on an analytical solution of $\boldsymbol{\theta}*$ for large datasets. This is what the rest of this report will focus on.

No. of training samples	Average runtime
10	0.0002
100	0.0001
1000	0.1306
10000	1.3308

Table 2: Runtime for different training dataset sizes

3 Optimising linear regression model with Gradient Descent

3.1 Motivation and theory

It is often not possible to find an analytical solution that will optimise the objective function in question so GD is used to iteratively approach the optimal model parameters. To compute the gradient of the MSE objective function, first rewrite Eq.1 as

$$J_{mse}(\boldsymbol{\theta}, \hat{\boldsymbol{X}}, \boldsymbol{y}) = \frac{1}{n}[(\hat{\boldsymbol{X}}\boldsymbol{\theta})^T(\hat{\boldsymbol{X}}\boldsymbol{\theta}) - \boldsymbol{y}^{\boldsymbol{T}}\hat{\boldsymbol{X}}\boldsymbol{\theta} - (\hat{\boldsymbol{X}}\boldsymbol{\theta})^{\boldsymbol{T}}\boldsymbol{y} + \boldsymbol{y}^{\boldsymbol{T}}\boldsymbol{y}], \tag{3}$$

and then differentiating this with respect to $\boldsymbol{\theta}$ using rules of matrix and vector differentiation gives

$$\nabla_\theta J_{mse}(\boldsymbol{\theta}, \hat{\boldsymbol{X}}, \boldsymbol{y}) = \frac{2}{n}(\hat{\boldsymbol{X}}^{\boldsymbol{T}}\hat{\boldsymbol{X}}\boldsymbol{\theta} - \hat{\boldsymbol{X}}^{\boldsymbol{T}}\boldsymbol{y}). \tag{4}$$

3.2 Methodology and MATLAB Implementation

Since there are now additional parameters to define, it is necessary to randomly divide the input data into a training dataset (used to learn $\boldsymbol{\theta}*$) and a validation dataset (used to find the optimal learning rate, λ, and number of iterations of the algorithm). This was implemented in the $divide\_data$ function. The GD function utilised a for loop to calculate $\nabla_\theta J_{mse}$ and update the value of $\boldsymbol{\theta}$ at each iteration.

3.3 Results and Discussion

For successful GD convergence, the optimum value of λ must be found. The measure of performance was MSE over validation data after 1000 iterations, and as Table 3 shows, the optimum value was $\lambda = 0.1$. The optimum λ value resulted in an MSE of 0.0397 over the test data, lower than the 0.0424 observed over validation data. This was unexpected, as the λ was chosen to optimise error over validation data, not test data. This is likely down to noise as this was only tested using 1 seed. With only 200 validation samples, the MSE values in Table 3 are likely to exhibit large variance. Going to 10,000 iterations, the optimum value was $\lambda = 0.01$. The MSE values for the 10,000 iteration experiment suggest an inverse relationship between the number of iterations and optimal λ. This could be because there is a set distance to traverse from final to initial theta. If the estimated model parameters never overshoot the optimum theta value then this distance must be equal to the distance travelled in each step (learning rate) multiplied by the number of steps. In practice, fewer iterations and larger λ is preferable as the algorithm runtime is proportional to the number of iterations, so the optimal values are 1000 iterations and $\lambda = 0.1$.

λ value	MSE over validation data
0.00001	10.2608
0.0001	1.4020
0.001	0.1996
0.01	0.0484
0.1	0.0424
1	NaN

Table 3: MSE over validation data for different λ values

The closed-form model from Sec.2 achieved an MSE of 0.0398 with the same training and validation data - lower than the 0.0424 achieved by GD. This is expected as the closed-form solution should give the exact $\boldsymbol{\theta}$* that minimises the error over the training data, whereas GD approaches this solution iteratively. Finally, the effect of increasing the number of training samples is considered. Fig.1 shows that there is still poor time complexity as the training database grows in size. This is because all the matrix operations (particularly computation of $\nabla_\theta J_{mse}$) take longer as their size increases. Stochastic gradient descent speeds up gradient computation, which is especially important for larger databases.

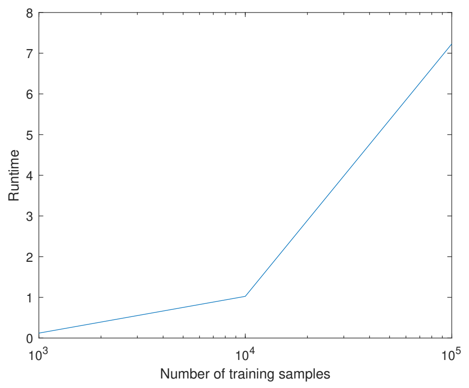


Figure 1: Runtime vs size of training dataset

4 Logistic regression for classification using Gradient Descent

4.1 Motivation and theory

In this binary classification task, the two Gaussian distributions mentioned in Sec.2.2 are each associated with a class and the goal is to predict which class a given data point came from by establishing a decision boundary between the two classes, with parameters contained in $\boldsymbol{\theta}$. As in Sec.3, GD will be used but for logistic regression the objective function is the mean log-loss ($\mathcal{L}_C$) in order to preserve its convex nature. To differentiate $\mathcal{L}_C$, first note that the derivative of $\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})$, using the quotient rule and simplifying, is $\hat{\boldsymbol{x}}\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})(1-\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta}))$. By the chain rule,

$$\begin{aligned}\nabla_\theta \mathcal{L}_C(\boldsymbol{\theta}, \hat{\boldsymbol{x}}, y) &= -y(1+e^{-\hat{\boldsymbol{x}}\boldsymbol{\theta}})\hat{\boldsymbol{x}}\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})(1-\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})) + (1-y)\frac{1+e^{-\hat{\boldsymbol{x}}\boldsymbol{\theta}}}{e^{-\hat{\boldsymbol{x}}\boldsymbol{\theta}}}\hat{\boldsymbol{x}}\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})(1-\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})) \\ &= -y\hat{\boldsymbol{x}}(1-\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})) + (1-y)e^{\hat{\boldsymbol{x}}\boldsymbol{\theta}}\hat{\boldsymbol{x}}(1-\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta})) \\ &= (\sigma(\hat{\boldsymbol{x}}\boldsymbol{\theta}) - y)\hat{\boldsymbol{x}}.\end{aligned} \tag{5}$$

4.2 Methodology and MATLAB Implementation

The implementation was broadly similar to Sec.3, except for the definitions of the objective function and its gradient. There was also a need for a new performance metric (instead of MSE), implemented in err_perc, which outputs the percentage of data points that were incorrectly classified.

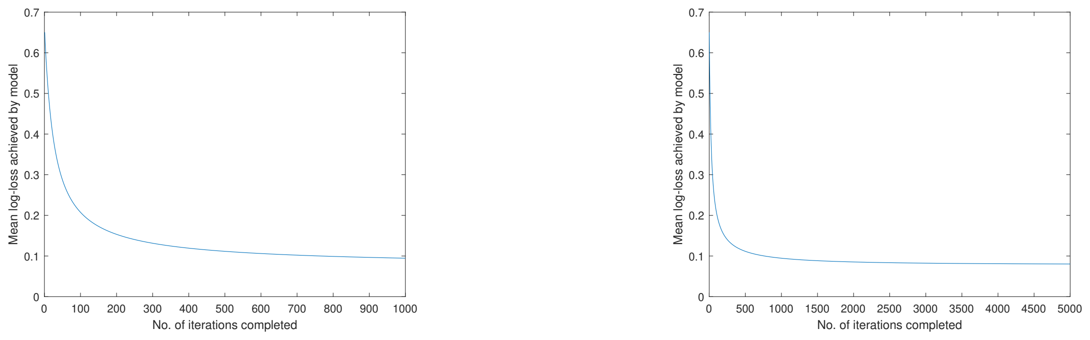
4.3 Results and Discussion

Table 4 shows that $\lambda = 1$ minimises the error percentage, so this is the optimal learning rate.

Testing for convergence, Figure 2a shows that despite the error percentage reaching a steady state, the mean log-loss has not converged after 1000 iterations. After 5000 iterations, Figure 2b seems to indicate that the cost has converged so this is the new number of iterations. With 20,000 test samples, the error percentage achieved was 3.39%, validation error was 4.00% and training error was 3.00%. As expected, the training error is lowest as the model is tested on the same data it used to learn $\boldsymbol{\theta}$.

λ value	Error percentage on validation data
0.00001	45.50
0.0001	45.50
0.001	45.50
0.01	17.50
0.1	5.00
1	4.00
10	4.50

Table 4: Error percentage for different λ values



(a) 1000 iterations *(b) 5000 iterations*

Figure 2: Mean log-loss plotted after each iteration with $\lambda = 1$

Table 5 shows that mean error percentage on test and validation data decreases as the number of training samples increases, as does standard deviation across all sets of data (with the exception of 10 samples tested on training data, which the model classified perfectly every time). These are both expected trends which were observed in Sec.2.3. With 10 training samples, there is a large discrepancy in training error versus test and validation error, which could indicate overfitting. This discrepancy is absent with 1000 samples because overfitting is more difficult with more training samples.

		Number of training samples			
		10	20	100	1000
Mean ± SD of Error tested on:	Test data	7.6115 ± 3.6761	4.6260 ± 0.8421	4.1270 ± 0.7304	3.5255 ± 0.1406
	Validation data	20.0000 ± 34.9603	11.0000 ± 6.4235	5.0000 ± 4.7140	3.3500 ± 0.7091
	Training data	0.0000 ± 0.0000	1.2500 ± 3.9529	2.2500 ± 2.2669	3.5625 ± 0.5344

Table 5: Error percentages averaged over different seeds

5 Logistic regression with Stochastic Gradient Descent

5.1 Motivation and theory

The GD algorithm implemented in Sec.4 has poor performance with large training databases, due to the expensive computation of the gradient in each iteration. This can be improved with SGD, where the gradient is calculated using a stochastic batch of samples rather than the whole training dataset.

5.2 Methodology and MATLAB Implementation

The only difference in implementation between this and normal GD is that in each iteration, a random batch of samples for gradient computation is taken using MATLAB function $randsample$.

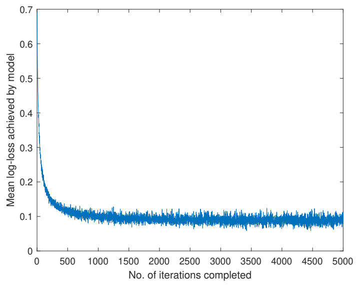
5.3 Results and Discussion

There is now a new hyper-parameter that must be configured - batch size, i.e. the number of samples used to compute the gradient. There is an expected trade-off between accuracy and speed, as increasing batch size makes the gradient computations more accurate but slower, as the algorithm is closer to standard GD. Table 6 shows that error over test data monotonically decreases as batch size increases. Over training and validation error, there is also a general decrease in error, though these results are subject to large variance due to the small number of samples used in testing.

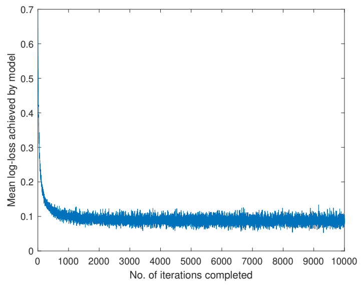
Over both 5000 and 10,000 iterations, the model appears to converge well albeit with some additive noise. The stochasticity adds variance to the cost plot because the samples that the cost is calculated over are different in each iteration. The final error percentage was the same in Fig.3a and Fig.3b, so 5000 iterations is adequate. Fig.3c shows that using a smaller batch size results in greater noise being added to the plot. The error on test data was 3.40%, and on validation data it was 4.00%, compared to GD errors of 3.53% on test data and 3.35% on validation data. GD performed better on validation data, as expected. However, SGD appeared to perform better on test data. This may be because the more accurate gradient computation of GD isn't necessary when generalising the model for test data. Also, the test error with SGD was lower than validation error, which is unexpected but may be because the datasets used are small and SGD values are subject to large variance, especially for errors

		Batch size				
		1	10	20	50	100
Mean ± SD of Error tested on:	Test data	4.1370 ± 0.7489	3.5305 ± 0.1214	3.5285 ± 0.1098	3.5255 ± 0.1575	3.5030 ± 0.1194
	Validation data	3.9500 ± 0.8317	3.2000 ± 0.9775	3.1500 ± 0.8182	3.3500 ± 0.8835	3.3500 ± 0.7472
	Training data	4.0000 ± 0.8858	3.5125 ± 0.5727	3.4875 ± 0.4268	3.6000 ± 0.5231	3.4750 ± 0.5798

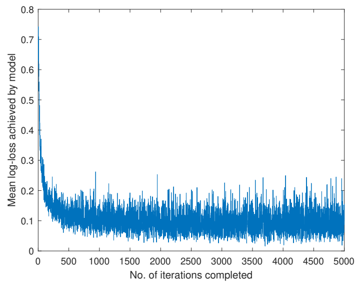
Table 6: Error performance of SGD with different batch sizes



(a) 5000 iterations



(b) 10000 iterations



(c) Batch size of 10

Figure 3: Mean log-loss plotted after each iteration for SGD

over validation data, as Table 6 showed. Fig.4 shows how increasing the batch size increases runtime. However, all batch sizes ran faster than normal GD, and similar trends are expected for memory requirements. Overall, SGD completes each iteration faster but each step is more noisy so more iterations may be required.

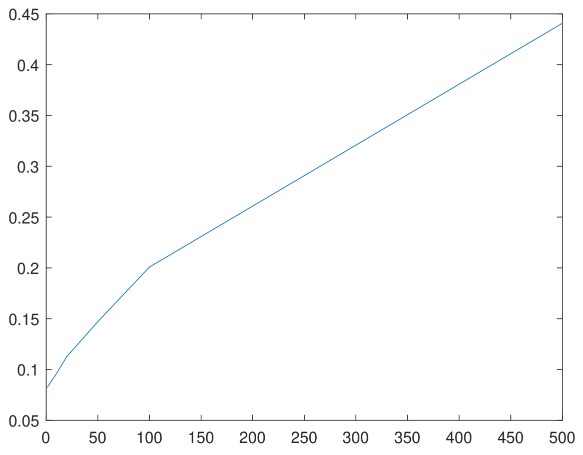


Figure 4: Runtime (y-axis) vs batch size (x-axis)

6 Support vector machine and Linear Programming

6.1 Motivation and theory

A Support Vector Machine (SVM) is an alternative classification model. $\boldsymbol{\theta}$ contains the parameters of the linear decision boundary that minimises the classification error. Since $\bar{y} = \boldsymbol{x\theta}$ and the boundary is defined at $\bar{y} = 0$, the equation of the line is

$$x_2 = \alpha x_1 + \beta = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}. \tag{6}$$

6.2 Methodology and MATLAB Implementation

Since the objective function must be optimised over two variables, $\boldsymbol{\theta}$ and $\boldsymbol{\xi}$, these are combined into $\boldsymbol{\psi} = [\boldsymbol{\theta}, \boldsymbol{\xi}]^T$. The code below shows how the optimisation problem was formulated into the $linprog$ form, within function $train\_SVM\_linear\_progr$.

```
n = length(X_train);                % Set n to no. of samples for convenience
y_matrix = diag(y_train);           % So we can easily multiply with X_train
iden = eye(n);
A = -[y_matrix*X_train iden];       % A and b define our constraint
b = - ones(n,1);
f = [0;0;0;ones(n,1)];              % Objective function sums slack values
lb = [-Inf;-Inf;-Inf;zeros(n,1)];   % No lower bounds on theta, 0 for slack
```

```
% Now we can call linprog. No upper bounds or equality condition so set these to [].
psi_opt = linprog(f,A,b,[],[],lb,[]);
theta_opt = psi_opt(1:3);                      % Extract theta values from psi
```

6.3 Results and Discussion

The training error using this SVM model was 3.20% and the test error was 3.28%. The test error is slightly better than the 3.40% achieved by logistic regression, so the $linprog$ function outperforms GD. The error cannot be 0 as it is impossible to draw a straight line that perfectly divides the data into the two classes, as Fig.5 shows. Using Eq.6, the lines are defined by $(\alpha, \beta) = (-0.9167, 2.8727)$ for SVM and $(\alpha, \beta) = (-0.7642, 2.6247)$ for GD. The values are similar, which is expected as the same problem is being solved with two different methods. The maximum number of training samples that the model could use was between 40,000 and 50,000. While SGD can avoid this issue to some extent by computing gradients on smaller stochastic batches, GD will face similar issues to $linprog$ when training on large databases.

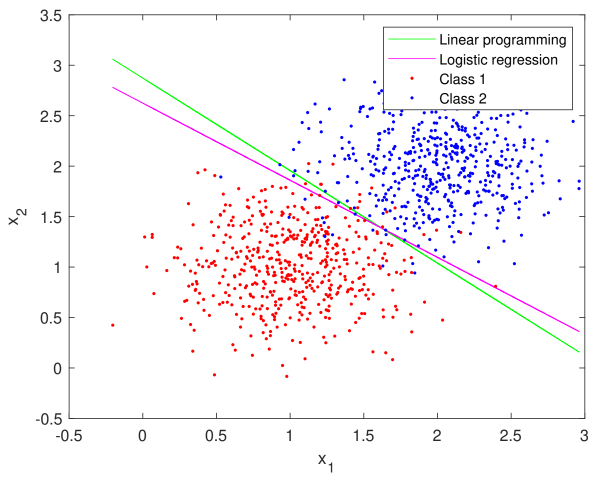


Figure 5: Decision boundaries plotted with input data

7 Support vector machine and Gradient Descent

7.1 Motivation and theory

In the final section, the SVM is optimised using GD rather than MATLAB's $linprog$ function. The objective function is now hinge loss, $\mathcal{L}_H$, which is piecewise differentiable.

7.2 Methodology and MATLAB Implementation

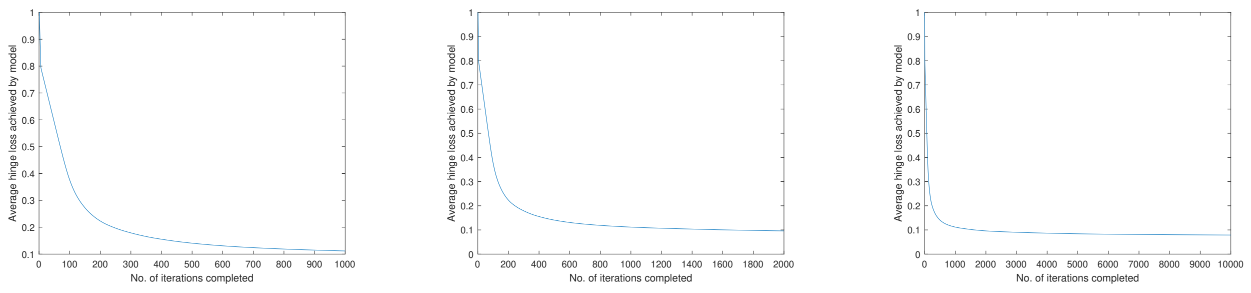
The main difference in implementation was the gradient computation, which now depends on the value of $\mathcal{L}_H$. The following code was repeated for each iteration of the algorithm. Code not relating to the gradient computation has been removed. At the end of this code, $\boldsymbol{\theta}$ would be updated using the $gradient$ value.

```
gradient = 0;                 % Reset gradient each time as we sum it over each sample
```

```
for idx = 1:n               % Iterate through samples in the data
    x = X_train(idx,:);     % Extract the sample
    y = y_train(idx);
    % Compute gradient. If hinge loss is not positive, don't update gradient.
    if cost_per_sample(idx)>0
         gradient=gradient-y*x;
    end
end
gradient = gradient/n;      % Divide by n to get average gradient rather than sum
```

7.3 Results and Discussion

Using the same methodology as before, the optimal learning rate was $\lambda = 0.1$. Fig.6 illustrates the convergence investigation. All three plots achieved the same final validation error rate of 4.00%. However, they appear to show that the hinge loss doesn't reach a steady state as the number of iterations increases. 2000 was the chosen number of iterations as error percentage is the main criteria for optimising hyper-parameters, though it is important to bear in mind that more iterations are needed to reach a steady $\boldsymbol{\theta}*$. Testing the model with 50,000 iterations gives $\boldsymbol{\theta}* = (-12.4579, 3.6979, 4.6921)$, compared to $(-12.4851, 3.9843, 4.3461)$ learned by linear programming. The training and test error were



(a) 1000 iterations *(b) 2000 iterations* *(c) 10,000 iterations*

Figure 6: Cost plotted after each iteration for GD on SVM

3.00% and 3.31% respectively. Linear programming achieved a slightly better test error of 3.28%, which makes intuitive sense as $linprog$ should perform better but is limited in its applicability. The decision boundary learned by this algorithm has coefficients $(\alpha, \beta) = (-0.7881, 2.6551)$ whereas $linprog$ learned $(\alpha, \beta) = (-0.9167, 2.8727)$. The discrepancies in these coefficients and in $\boldsymbol{\theta}*$ values suggest that even more GD iterations may be required to reach convergence.