Logistic Regression with Regularization

Zeehan Rahman

Aim: Optimize the weight vector using gradient descent

Challenge: the loss function of the logistic regression is not convex (weight vector can get trapped in local minima and never reach the global minimum)

Solution:

The derivative of the cross-entropy error loss for logistic regression is convex. Thus, wherever we start in the gradient descent, we will always end up at the global minimum. The following algorithm is based on the logistic regression algorithm from the lecture (Logistic Regression). The only difference being, we have added regularizer to avoid overfitting

initialize $\mathbf{w}_0$

For $t = 0, 1, \cdots$

1. compute

$$\nabla E_{\text{in}}(\mathbf{w}_t) = \left(\frac{1}{N} \sum_{n=1}^{N} \theta\left(-y_n \mathbf{w}_t^T \mathbf{x}_n\right)\left(-y_n \mathbf{x}_n\right) \right) + \frac{2\lambda}{N} w_t$$

2. update by

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta \nabla E_{\text{in}}(\mathbf{w}_t)$$

...until $\nabla E_{\text{in}}(\mathbf{w}_{t+1}) = 0$ or enough iterations

return last $\mathbf{w}_{t+1}$ as g

As the derivate of the loss function gives the direction of the steepest ascent, the negative of that gradient gives us the direction to decrease the function most quickly. Therefore, we multiply the gradient with a negative value, and then update the weight vector.

Step 1: Achieve optimum weight vector using gradient descent with regularization, with learning rate set to 0.03 and total iterations equal 10, 000.

(During the implementation of the program, I have used cross validation to arrive at the value for the learning rate and the total number of iterations)

Step 2: Calculate the in-sample error, using the in-sample set

a. Get the dot product of the weight vector and the training data (let's call the product M)
b. Pass each element of M, as a parameter to the sigmoid function (let's call it M1)
c. Using the threshold value of 0.5, classify all elements of M1 as 1(positive) when they are at least 0.5. Otherwise, classify the elements as -1(negative).
d. Consequently, count the number of mistakes by comparing the result with the label, and get the average error rate

Repeat step a to d, using 5-fold cross validation, and get the average error rate

Step 3: Repeat step 2, this time using the validation set, to obtain E_val

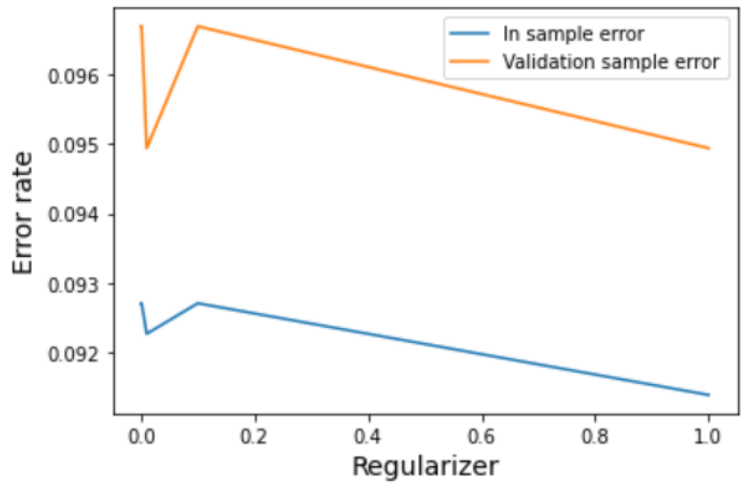
Step 4: Repeat step 1 using a different value for regularization

Step 5: Plot the performance plots for various regularizer for both E_in and E_val. (Figure below)

Experiment:

We set the training and testing data using 5-fold cross validation. In 5-fold cross validation, we run five iterations of training, summing the in-sample error, testing our model and summing the validation error. At the end of the five iterations, we divide the summed up in sample and validation error by five, to get the average error rate. By making sure every set of data has a chance of appearing the training and testing phase, the model is less biased

Result and Discussion:




```
E_in sample array: [0.09270677 0.09270677 0.09226721 0.09270677 0.09138809]
E_val array: [0.09669306 0.09669306 0.09493867 0.09669306 0.09493867]
```

Regulizer array: [0,0.001, 0.01, 0.1,1]

According to the plot, both in sample and validation error reaches their minimum when the regularizer equals to 1 (when learning rate is set to 0.03)

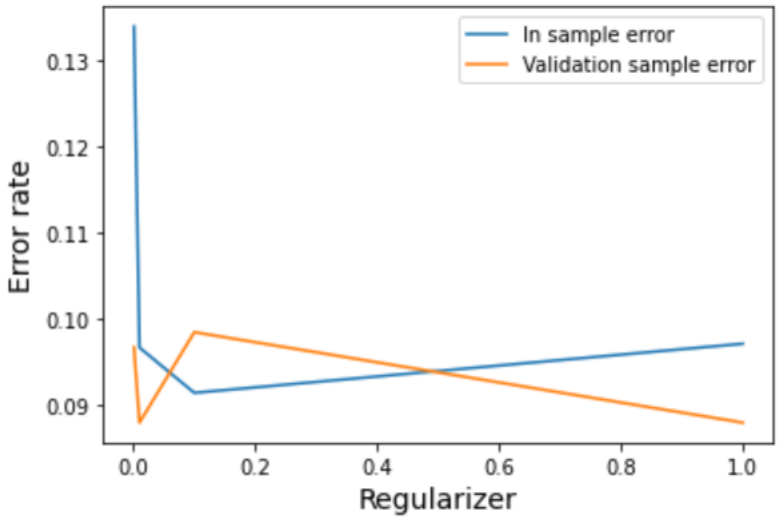
As we do not know the complexity of the target function, we might use a too complex g(x) and face overfitting. When using gradient descent, we might optimize to weight vector that causes the lowest in sample error, however that same weight vector might not perform well in the validation set. Therefore, we add some constraint during the gradient descent, by adding a regularizer: $\frac{2\lambda}{N}\mathbf{w}_{REG}$

The lambda in the regularizer adds constraint to the derivative of E_in. The value of the lambda determines how much the length of the weight vector is allowed. With the added constraint, the algorithm finds us the weight vector with that produces the lowest error rate with the constrained length of the weight vector.

As difference between the in-sample error and the validation error is not high (about 0.04), there is no issue of overfitting.

Repeat Step 1 using feature transform with degree set to 2 with the learning rate 0.03

When degree is set to 2, the number of dimensions increases from 30 to 496 features



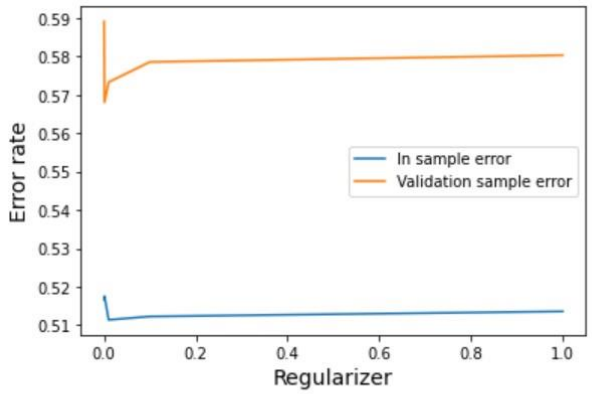
We notice with regularizer set to 0 (no constrain), there is a considerable difference between the in-sample error and the validation error (about 0.14 – 0.09 = 0.05), compared to the previous plot when no feature transform was applied.

As we are increasing the model complexity, d_vc increases (as we increases the number of hypotheses). Moreover, according to the Hoeffding inequality

$$\mathbb{P}_{\mathcal{D}}\Big[\underbrace{|E_{\text{in}}(g) - E_{\text{out}}(g)| > \epsilon}_{\textbf{BAD}}\Big] \quad \leq \quad \underbrace{4(2N)^{d_{\text{VC}}} \exp\left(-\tfrac{1}{8}\epsilon^2 N\right)}_{\delta}$$

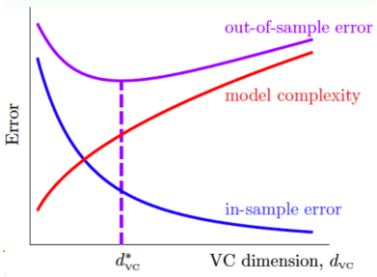
there is less chance of E_in and E_val being close to each other. Therefore, we a similar trend in the figure above (when regularizer is 0).

Degree 3, learning rate 0.001




```
E_in sample array: [0.51665992 0.51753904 0.51138519 0.51226335 0.51358203]
E_val array: [0.58907002 0.56801739 0.57328055 0.5785437  0.58029809]
Regularizer array:  [0, 0.001, 0.01, 0.1, 1.0]
```

The validation error drops when regularizer is 0.001, barely tackling the problem of overfitting. Also, since we are increasing the dimension to 5456 dimensions (degree=3), we are increasing the model complexity as the d_vc goes up (more ways to shatter the points), therefore we should at least observe an improvement in the sample error. However, the in-sample error is greater than 0.50, whereas the in sample error when degree equals 2 with the learning rate (0.1), is around 0.09 (page 9). This contradicts the graph where the in-sample error rate should drop for increasing model complexity.



As the time to train the model with degree set to 3, was taking too long, I only trained it once for learning rate 0.001

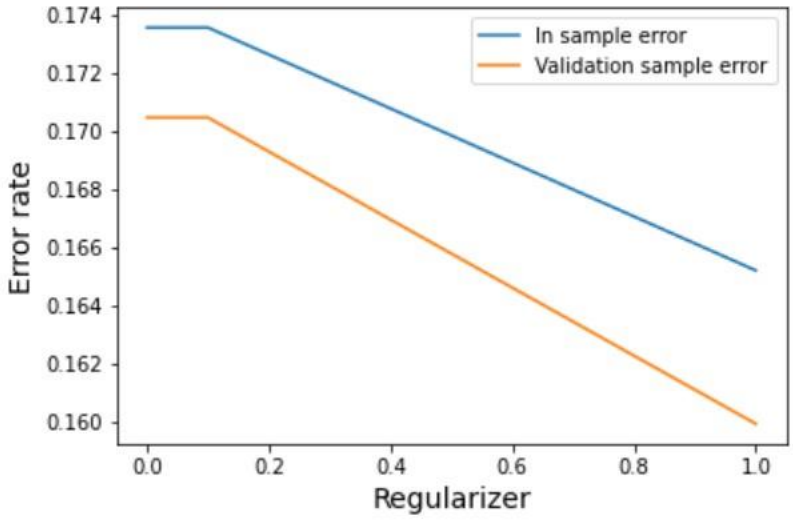
When degree set to 5, the training time was impractical.

When the degree for feature transformation is set to 10, an error is thrown:

```
MemoryError: Unable to allocate 3.51 TiB for an array with shape (569, 847660528) and data type float64
```

I also ran the program with different learning rate and degree of transformation:

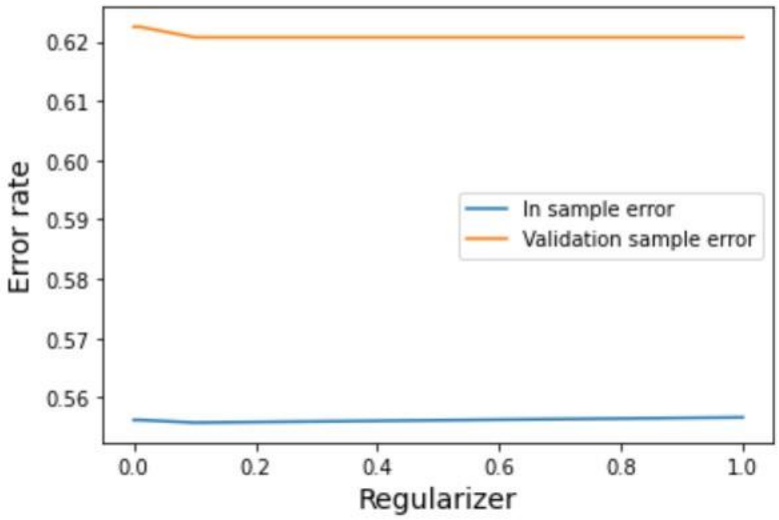
Degree 1, learning rate 0.001




```
E_in sample array: [0.17354733 0.17354733 0.17354733 0.17354733 0.16519761]
E_val array: [0.1704549  0.1704549  0.1704549  0.1704549  0.15992858]
Regularizer array:  [0, 0.001, 0.01, 0.1, 1.0]
```

Observation: The odd thing about this is that in sample error is greater than the validation error, for all the value of the regularizer.

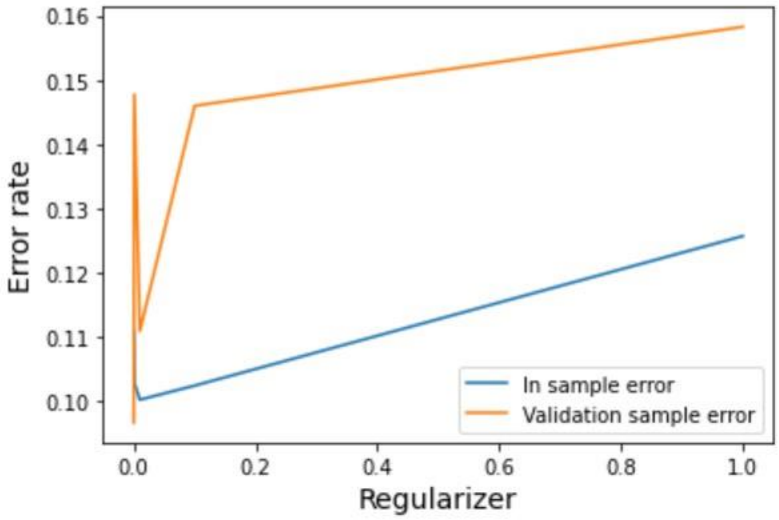
Degree 2, learning rate 0.001




```
E_in sample array: [0.55619433 0.5561953  0.5561953  0.55575574 0.55663389]
E_val array: [0.62244993 0.62244993 0.62244993 0.62068002 0.62068002]
Regularizer array:  [0, 0.001, 0.01, 0.1, 1.0]
```

Observation: The in-sample error is lower than the validation error. The graph also shows some sign of overfitting, as there is a noticeable gap between the in-sample and validation error rate. Moreover, validation error drops a little for values of regularizer greater than 0.01, trying to tackle the issue of overfitting. This example complies with what we have learnt: regularizer prevents overfitting. Most importantly, the error rate is considerably higher than the other values of the learning rate

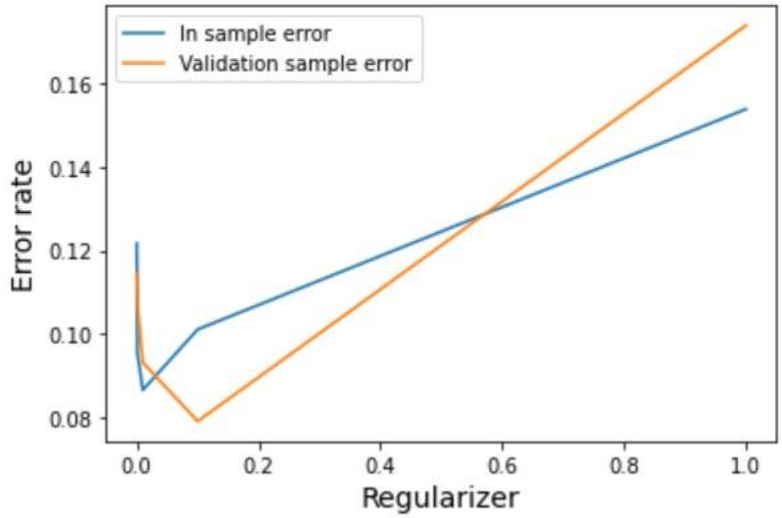
Degree 1, learning rate 0.1




```
E_in sample array: [0.11073453 0.10281184 0.10017447 0.10237228 0.12565934]
E_val array: [0.09661543 0.14770998 0.1108834  0.14594007 0.1582363 ]
Regularizer array:  [0, 0.001, 0.01, 0.1, 1.0]
```

Observation: We see the ideal value of the regularizer is somewhere around 0.001, where the difference between the in sample and validation error is the least. One interesting property to note about the graph is the steep drop of the validation error when the regularizer is around 0.001. Consequently, there is a steep increase in the validation error for regularizer values greater than 0.001. Also, the in-sample error has linear increase for increasing values of regularizer.

Degree 2, learning rate 0.1



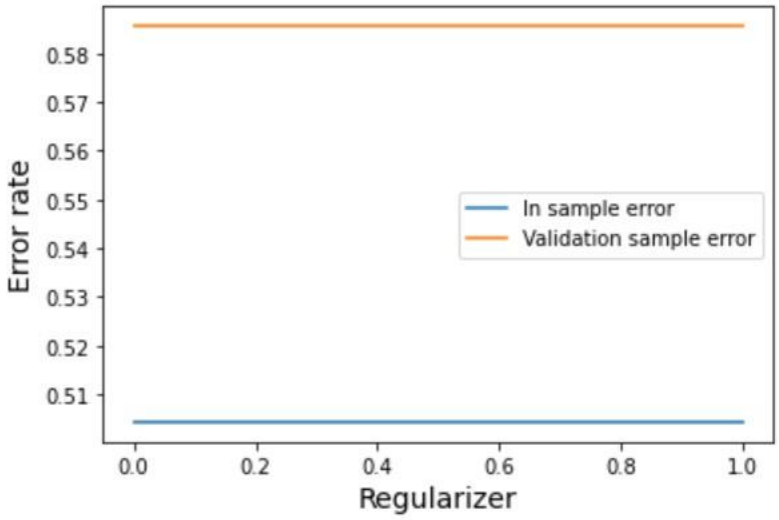

```
E_in sample array: [0.12170715 0.09534991 0.0865587  0.10106227 0.15381338]
E_val array: [0.11448533 0.10721938 0.09313771 0.07907157 0.17383947]
Regularizer array:  [0, 0.001, 0.01, 0.1, 1.0]
```

Observation: We notice the in-sample error is higher when there is no constraint. When the constraint increases, we see validation error becomes much lower than the in-sample error when regularizer is 0.1. The validation error becomes greater than the in-sample error when the regularizer is 1. However, it is surprising to notice the validation error is smaller than the in-sample error for some values of the regularizer. The behavior of the graph above is unexpected.

PCA:

I also ran PCA on the original dataset (no feature transform). The first dimension compromised about 98% of the invariance. Therefore, I reduced the dimension to 1 and ran Logistic Regression. Here is the graph of the result:

```
[0.98204467]
```




```
E_in sample array: [0.50435512 0.50435512 0.50435512 0.50435512 0.50435512]
E_val array: [0.58556125 0.58556125 0.58556125 0.58556125 0.58556125]
Regularizer array:  [0, 0.001, 0.01, 0.1, 1.0]
```

The error rate for both the in-sample error and the validation error remains constant for changing values of regularizer. Even though the first-dimension accounts for about 98% of the variance, we do not see any improvement in the error rate.

I ran PCA after feature transforming:

Applying feature transform of degree 2, the first dimension compromised 97.84% of the variance

Applying feature transform of degree 3, the first dimension compromised 97.60% of the variance