

# CMPS242 Homework #3 – Logistic Regression

Benjamin Sherman

&

Zayd Hammoudeh

October 23, 2017

# 1 Homework Objective

Develop a logistic regression-based learner that can identify SMS text messages as either SPAM or ham (i.e., not SPAM). Additional requirements include:

- Learner should implement a batch gradient descent
- Create a Jupyter (i.e., IPython) notebook to document all work

Four extra credit tasks were also possible for this homework. They are:

1. Support a bias term in the weight vector ( $\mathbf{w}$ ) that is not regularized
2. Experiment with different regularizers, such as  $\lambda \|\mathbf{w}\|_1$ .
3. Implement the stochastic gradient descent algorithm
4. Implement the exponentiated gradient algorithm ( $\text{EG}^\pm$ )

# 2 Gradient Descent Update Rule

Logistic regression is a subcategory of binary classification. Define the logistic (sigmoid) function as shown in Eq. (1).

$$\sigma(a) = \frac{e^a}{1 + e^a} \quad (1)$$

As shown in Figure 1, this function is bounded between 0 and 1 making it ideal for binary classification since it can be used to represent probabilities without additional normalization. The following notes show the derivation of the logistic update rule.<sup>1</sup>

Define  $Z$  as a random variable that maps one classification value (e.g., ham) to 0 and the other classification result (e.g., SPAM) to 1. We could then define the probability of a 0 or a 1 as shown in Eq. (2) and (3) respectively. This can be written compactly as shown in Eq. (4).

$$\Pr\{Z = 0|a\} = 1 - \sigma(a) \quad (2)$$

$$\Pr\{Z = 1|a\} = \sigma(a) \quad (3)$$

$$\Pr\{Z = z|a\} = \sigma(a)^z (1 - \sigma(a))^{1-z} \quad (4)$$

Given a target classification vector  $\mathbf{y}$  and the relation  $a = \mathbf{w}^T \mathbf{x}$ , the likelihood ( $L$ ) is in turn defined as shown in Eq. (5). Since each element is independent, this probability can be transformed into a product as shown in Eq. (6), where  $n = |\mathbf{y}|$ .

$$L(\mathbf{w}) = \Pr\{\mathbf{y}|\mathbf{w}^T \mathbf{x}\} \quad (5)$$

$$= \prod_{i=1}^n \Pr\{y_i|\mathbf{w}^T \mathbf{x}_i\} \quad (6)$$

As with homework #1, the value that maximizes the likelihood of a non-negative function also maximizes the log likelihood. Hence, Eq. (6) can be transformed to Eq. (11).

---

<sup>1</sup>This proof is based on the lecture notes of Andrew Ng.

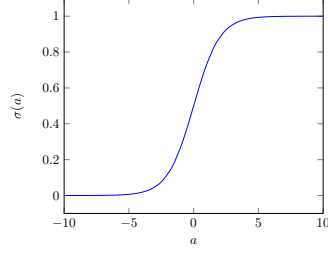


Figure 1: Plot of the Logistic Function

$$\ln L(\mathbf{w}) = \ln \left( \prod_{i=1}^n \Pr\{y_i | \mathbf{w}^T \mathbf{x}_i\} \right) \quad (7)$$

$$= \sum_{i=1}^n \ln \Pr\{y_i | \mathbf{w}^T \mathbf{x}_i\} \quad (8)$$

$$= \sum_{i=1}^n \ln \left( \sigma(\mathbf{w}^T \mathbf{x}_i)^{y_i} (1 - \sigma(\mathbf{w}^T \mathbf{x}_i))^{1-y_i} \right) \quad (9)$$

$$= \sum_{i=1}^n \ln \left( \sigma(\mathbf{w}^T \mathbf{x}_i)^{y_i} + \ln \left( (1 - \sigma(\mathbf{w}^T \mathbf{x}_i))^{1-y_i} \right) \right) \quad (10)$$

$$= \sum_{i=1}^n y_i \ln \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \ln(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \quad (11)$$

To find the maximizing likelihood, the derivative is taken and set equal to 0. We use the identity  $\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a))$  without proof, but the proof is trivial. Hence the maximizing value is shown in Eq. (15); note that the *chain rule* was used for this derivative.

$$0 = \sum_{i=1}^n \left( y_i \frac{1}{\sigma(\mathbf{w}^T \mathbf{x}_i)} \frac{\partial \sigma(\mathbf{w}^T \mathbf{x}_i)}{\partial \mathbf{w}} - (1 - y_i) \frac{1}{1 - \sigma(\mathbf{w}^T \mathbf{x}_i)} \frac{\partial \sigma(\mathbf{w}^T \mathbf{x}_i)}{\partial \mathbf{w}} \right) \quad (12)$$

$$= \sum_{i=1}^n \left( y_i \frac{1}{\sigma(\mathbf{w}^T \mathbf{x}_i)} \sigma(\mathbf{w}^T \mathbf{x}_i)(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \mathbf{x}_i - (1 - y_i) \frac{1}{1 - \sigma(\mathbf{w}^T \mathbf{x}_i)} \sigma(\mathbf{w}^T \mathbf{x}_i)(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \mathbf{x}_i \right) \quad (13)$$

$$= \sum_{i=1}^n \left( y_i(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) - (1 - y_i)\sigma(\mathbf{w}^T \mathbf{x}_i) \right) \mathbf{x}_i \quad (14)$$

$$= \sum_{i=1}^n (y_i - \sigma(\mathbf{w}^T \mathbf{x}_i)) \mathbf{x}_i \quad (15)$$

Eq. (15) represents the maximizing gradient for the log likelihood. This can then be substituted into the update rule for stochastic gradient descent, where  $\eta$  is the hyperparameter, learning rate.

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \eta \nabla \ln L(\mathbf{w}) \quad (16)$$

We can then substitute into Eq. (16), using tensor notation since this is batch update.

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \eta \mathbf{X}^T (\mathbf{y} - \hat{\mathbf{y}})^T \quad (17)$$

Note that that  $\mathbf{w}_t$  was used in the gradient since it is not known how to maximize on  $w_{t+1}$ . In addition, the learning rate,  $\eta$  uses a constant configured by the user an aging term as shown in Eq. (18).

$$\eta = \frac{\eta_0}{t^{0.9}} \quad (18)$$

Our implementation uses the update rule and learning rate in Eq. (17) and (18) respectively for all batch gradient descent experiments in this paper.

### 3 Text Preprocessor

Before learning can be attempted the natural language text in the form of SMS messages much be turned into a feature vectors. We accomplish this through a multistep process described below (in order).

- *Import CSV File* – Pandas’ native CSV support as part of the **DataFrame** object was used.
- *Invalid Character Removal* – The original text files (before my team cleaned them for the class) had invalid, unsupported characters. We cleaned the
- *Punctuation Removal* – The punctuation set is defined by Python’s **string** package (i.e., “string.punctuation”). Punctuation removal entailed replacing all punctuation symbols with the empty string (e.g., “couldn’t” became “couldnt”).
- *Capitalization Correction* – Standardizes case so that words with different capitalization are not treated differently.
- *Stop Words Removal* – The set of “stop words” is non-standard. For our experiments, we used the English “corpus” dictionary bundled with the Natural Language Toolkit (NLTK).

## 4 Jupyter Notebook Implementation

Jupyter notebooks serve as a intermediary layer that insulates users from the underlying code implementation. We included multiple features into our Jupyter notebook to improve the user experience and provide external learner configuration and data visualization.

### 4.1 Specifying Hyperparameters

Since Jupyter notebook masks the code from the user, some may find it difficult to experiment with different configurations. To address this limitation, we added a series of widgets to our notebook. They leverage the **ipywidgets** library. The specific hyperparameters that are controllable by widgets are:

- Cross-validation fold count ( $k$ )
- Learning algorithm (e.g., standard gradient descent, exponentiated gradient descent, and stochastic gradient descent)
- Regularizer selection (e.g.,  $L_2$  or  $L_1$  norm)
- Error calculation formula (e.g.,  $1 - Accuracy$  or root-mean square)
- Maximum number of training epochs
- Learning rate ( $\alpha$ )
- $\lambda$  range – These are the powers of 2 to test.

Figure 2 shows a screenshot from our Jupyter notebook with the widgets included.

Number of Folds:  10

Select the Learning Algorithm: ☒ Batch Gradient Descent  
☐ EG+/-  
☐ Stochastic Gradient Descent

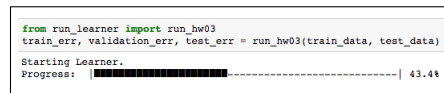
Select the Regularizer: ☐ L1 Norm  
☒ L2 Norm

Max. Number of Epochs:  25

Learning Rate ( $\eta$ ):  20.00

Range of  $\lambda$  in Form  $2^k$ :  -10 - 10

Select the Validation Error Calculation: ☒ Accuracy  
☐ RMS



## 4.2 Learner Progress Indicator

As with most iterative learning algorithms, our learner does provide instantaneous results. Depending on the user specified configuration described in the previous section, it may take several minutes before a run of the algorithm completes. Our implementation provides visual feedback on the learners progress via a text-based progress bar, which is shown in Figure 3.<sup>2</sup>

### 4.3 Embedded Results Table

To improve the Jupyter notebook experience, we built in support for a table that will display the latest execution’s results. The minimum training, validation and test errors are bolded and highlighted in yellow as shown in Figure 4.

#### 4.4 Embedded Result Graphing

We also implemented graphing into our Jupyter notebook. This uses Python's `matplotlib` package. An example graph is shown in Figure 5.

## 5 Experiment Setup

Our solver is implemented in Python 2.7.13. It requires a set of external packages, including:

$\lambda$	0	$2^{-10}$	$2^{-9}$	$2^{-8}$	$2^{-7}$	$2^{-6}$	$2^{-5}$	$2^{-4}$	$2^{-3}$	$2^{-2}$	$2^{-1}$	$2^0$
Training	0.0053	0.0052	0.0051	0.0055	0.0077	0.0113	0.0199	0.0433	0.0495	0.0906	0.1390	0.1390
Validation	0.0300	0.0300	0.0297	0.0297	0.0313	0.0410	0.0590	0.0700	0.0727	0.1067	0.1390	0.1390
Test	0.0233	0.0229	0.0241	0.0257	0.0276	0.0385	0.0525	0.0614	0.0634	0.0956	0.1283	0.1283

Figure 4: Example Jupyter notebook table of experimental results

<sup>2</sup>The underlying implementation used for this feature was from an online resource and not developed natively by our team.

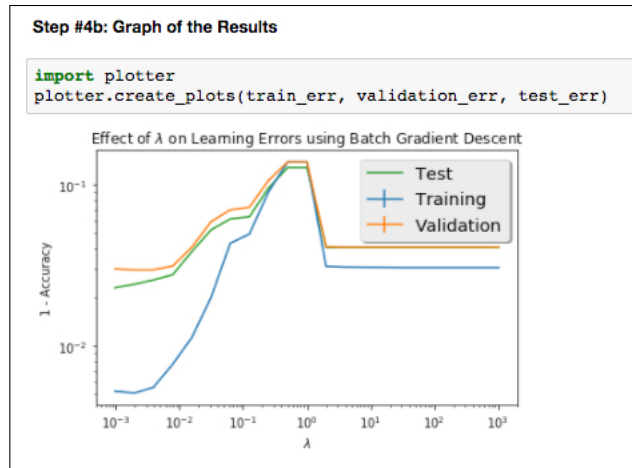


Figure 5: Example Jupyter notebook graph of experimental results

- Pandas
- Scikit-Learn
- NumPy
- IPython
- Matplotlib
- ipywidgets.

The default experiment setup was:

- *Performance Metric*: Accuracy which rounds  $\hat{y}$  to the closest integer.
- *Learning Algorithm*: Batch gradient descent
- *Regularizer*:  $L_2$  norm
- *Cross-validation Fold Count ( $k$ )*: 10
- *Number of Training Epochs/Rounds*: 15
- *$\lambda$  Range*:  $0, 2^{-10}, 2^{-9}, \dots, 2^9, 2^{10}$
- *Learning Rate ( $\eta_0$ )*: 20
- *Graph Error Bars*: These are included in all graphs. However, the variance between folds is low meaning they are often not visible.

## 6 Extra Credit #1: Bias Term Support

A bias term in the learning algorithm can only improve the learner's performance by accounting for shifts in the data. As such, our implementation only supports training with a bias; we deliberately chose that this feature could not be disabled.

The bias term is added to the input tensor,  $X$ , as a column of value 1. This is done using NumPy. When calculating the regularization error, we ignore the error caused by the bias term by setting its error to 0 as shown in Figure 6.

```
def l2_norm_regularizer(lambda_val, w):
    reg_err = np.multiply(lambda_val, w)
    reg_err[0, 0] = 0 # Exclude the bias
    return reg_err
```

Figure 6: Implementation of Bias Regularizer Error Removal

Table 1: Effect of the  $L_2$  Regularizer on Batch Gradient Descent Performance

$\lambda$	0	$2^{-10}$	$2^{-9}$	$2^{-8}$	$2^{-7}$	$2^{-6}$	$2^{-5}$	$2^{-4}$	$2^{-3}$	$2^{-2}$
Training	0.008	0.007	0.007	<b>0.006</b>	0.008	0.013	0.023	0.043	0.058	0.099
Validation	<b>0.027</b>	0.028	0.029	0.029	0.032	0.039	0.057	0.072	0.081	0.113
Test	<b>0.024</b>	0.024	0.024	0.026	0.028	0.034	0.051	0.060	0.061	0.098

## 7 Batch Gradient Descent Learning Performance

Our batch gradient descent implementation followed the experiment setup described in Section 5. The best error was observed when there was no regularizer as shown in Table 1 and Figure 7. It is also important to note that as the regularizer increased, the error drops precipitously and remains largely unchanged. This occurs because the regularizer begins to saturate the sigmoid function causing it to return a *yhat* of 0 for all but the most obvious cases of spam.

## 8 Extra Credit #2: Multiple Regularizer Experimentation

An extra credit portion of the assignment was to investigate the performance of the learner with another regularizer. As the homework proposed, we use the  $L_1$ . Unlike the  $L_2$  from Section ??, the derivative of the  $L_1$  norm is a vector of ones multiplied by *lambda*. As such, the value of  $\mathbf{w}$  does not affect the regularization. As expected, the affect of this norm on the learner's performance is far more muted as shown in Table 2 and Figure 8. The clamping of the logistic function was not observed and there was not significant change in the learner's performance for small values of  $\lambda$  (i.e., less than 1). This behavior is fully expected.

## 9 Extra Credit #3: Stochastic Gradient Descent

In each training round/epoch, batch gradient descent performs a single update of the weight vector based off *all* the training samples. In contrast, given a dataset of size  $m$ , stochastic gradient descent (SGD) performs  $m$  updates of the weight vector in each epoch. This generally should cause the trainer to require less epochs and train faster on average.

Since a single element at a time is used to update the weight vector, the learning rate,  $\eta_0$  should be smaller to prevent over-correction based on a single outlier. As such, we reduced the learning rate to 0.1 for

Table 2: Effect of the  $L_1$  Regularizer on Batch Gradient Descent Performance

$\lambda$	0	$2^{-10}$	$2^{-9}$	$2^{-8}$	$2^{-7}$	$2^{-6}$	$2^{-5}$	$2^{-4}$	$2^{-3}$	$2^{-2}$
Training	<b>0.007</b>	<b>0.007</b>	<b>0.007</b>	0.008	0.008	0.008	0.008	0.008	0.008	0.008
Validation	<b>0.030</b>	<b>0.030</b>	<b>0.030</b>	<b>0.030</b>	<b>0.030</b>	<b>0.030</b>	<b>0.030</b>	0.031	0.031	0.031
Test	<b>0.024</b>	<b>0.024</b>	<b>0.024</b>	<b>0.024</b>	<b>0.024</b>	<b>0.024</b>	<b>0.024</b>	<b>0.024</b>	<b>0.024</b>	0.026

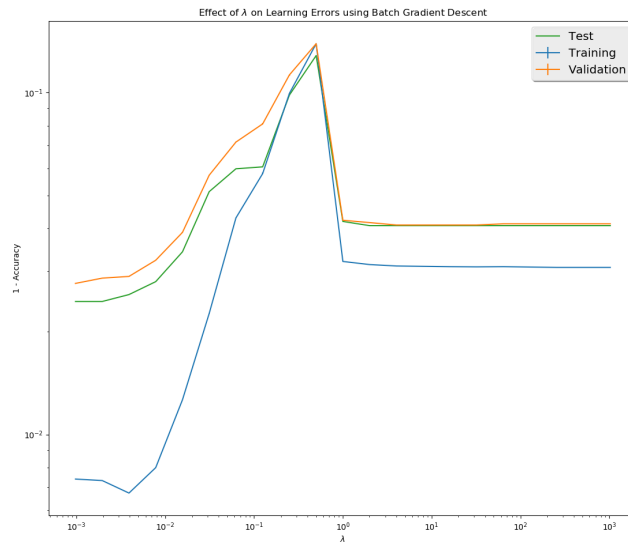


Figure 7: Effect of regularizer  $\lambda$  on batch gradient descent performance using  $L_2$  norm

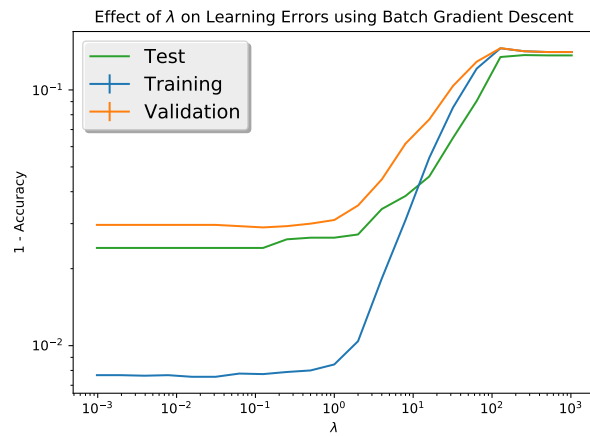


Figure 8: Effect of regularizer  $\lambda$  on batch gradient descent performance using  $L_1$  norm



Table 3: Effect of the  $L_1$  Regularizer on Batch Gradient Descent Performance

$\lambda$	0	$2^{-10}$	$2^{-9}$	$2^{-8}$	$2^{-7}$	$2^{-6}$	$2^{-5}$	$2^{-4}$	$2^{-3}$	$2^{-2}$
Training	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	0.001	0.002	0.005	0.013	0.024	0.046
Validation	0.026	<b>0.025</b>	0.029	0.030	0.030	0.032	0.032	0.034	0.040	0.061
Test	<b>0.018</b>	0.019	0.019	0.021	0.021	0.022	0.020	0.026	0.037	0.056

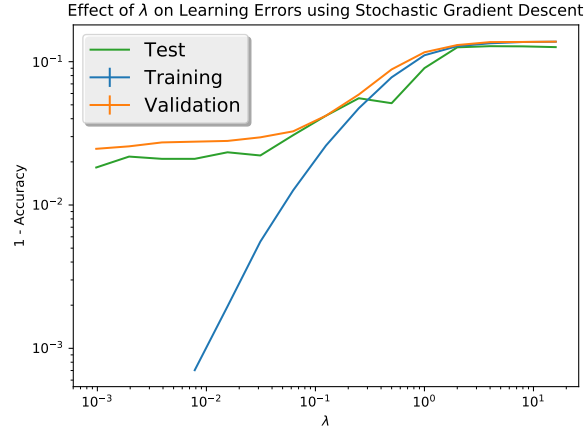


Figure 9: Stochastic Gradient Descent Performance using the  $L_2$  norm and a learning rate of 0.1

this experiment. Figure 3 and Table 3 show that SGD had better performance than batch gradient descent. Although the selected regularizer ( $2^{-10}$ ) did not have the minimum test error, the difference is quite small. Like despite also using the  $L_2$  norm, SGD did not have the same clamped error for larger values of  $\lambda$ .

While stochastic gradient descent takes far few epochs to converge (5 versus more than 15 for batch gradient descent), the running time is actually longer for SGD by about a factor of 3 times. This is due primarily to the inefficiencies of Python and how it interacts with NumPy. When all the matrix multiplications are performed at once, the just-in-time (JIT) compiler is able to optimize them. It is not able to do the same level of optimization when multiplications are made one at a time in SGD.