Assignment 02

Learning from Data, Related Challenges, Classification

submitted for

EN3150 - Pattern Recognition

Department of Electronic and Telecommunication Engineering
University of Moratuwa

Udugamasooriya P. H. J.
220658U
Progress on GitHub

20 August 2025

1 Linear Regression

Question 01 The main reason is the presence of outliers.

Based on the inliers, we see that the desired line is an almost horizontal one. To see why a slanted line has been produced instead of a horizontal line, we look at the behavior of the squaring function;

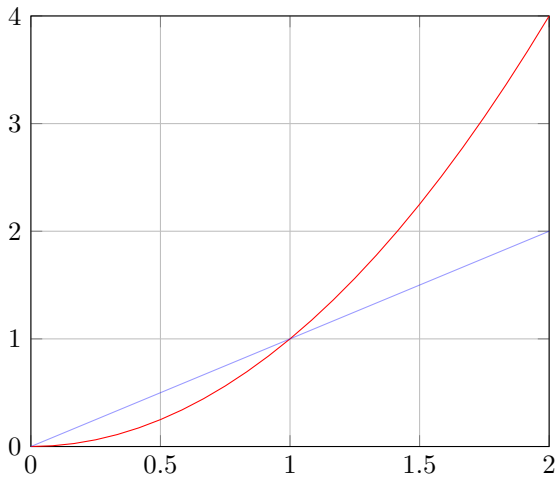

Figure 1: $y = x^2$

Note that for $|x| < 1$, $x^2 < |x| < 1$, i.e., the squares of numbers less than 1 are less than the numbers themselves (in absolute value). This means that the squared loss function can remain small enough for prediction errors that are just slightly below 1, but increases rapidly as it increases beyond 1.

In the given situation, even though a horizontal line would give near-zero error for the inliers, the error due to outliers would be very large. It is reasonable then, that a more optimal, i.e., with a lesser mean squared error, could be achieved with a slanted line,with lesser distance between outliers and the corresponding predictions.

Such a line would increase the gap between the inliers and the corresponding predictions and therefore increase the loss, but *not by too much*, due to the behavior of the square function. Instead, the reduction in the already-large gap between the outliers and the corresponding predictions, *will* cause a *rapid* decrease in the loss, *exceeding* the increase in the loss due to the inliers.

Question 02 Scheme 1.

The issue we wish to mitigate is the rapid increase in loss due to outliers, while keeping the loss due to inliers significant, with the intention that a model that better fits the inliers and is not strongly affected by the outliers as described above would be preferred.

With Scheme 1, we do not completely neglect the effect of outliers, but significantly scale down their contribution to the loss function, so they do not dominate the loss when the loss due to the inliers is small.

Scheme 2 is obviously not suitable, as it amplifies the loss from outliers over inliers; minimizing such a function would require the error due to outliers to be significantly reduced, causing more slanted line to be preferred, as described in the previous problem.

Question 03 In this case, the prediction to be made is of a categorical nature, i.e., a probability distribution over several possible classes, rather than a single continuous value.

In linear regression, we try to fit a continuous linear model to a given set of data points. In doing so, we make the following assumptions:

- the error between an actual data point and the corresponding prediction is normally distributed
- the data points vary continuously

However, in the case of categorical data, the variation of data points tends to be more discrete in nature, and the error between continuous-valued predictions and the data points is no longer normally distributed.

More specifically, if each voxel is treated as a feature, we expect the data matrix $\mathbf{X}$ to have almost linearly dependent rows/columns, as we expect a high correlation between the voxels due to spatial smoothness in the images. This would lead to instability in the computation of $\left(\mathbf{X}^\top\mathbf{X}\right)^{-1}$ required for the ordinary least squares weights.

On a final note, linear regression performs best when there are fewer features than data points for training. In our case, we expect the input images (data points) to consist of large numbers of voxels (features), but that there would be fewer such images than voxels. This might also lead to unstable coefficients, and sometimes overfitting, in the absence of regularization.

Question 05

2 Logistic Regression

Question 02 The following is a snippet of an error that was encountered:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

----> logreg.fit(X_train, y_train)

ValueError: could not convert string to float: 'Adelie'
```

The error indicates that not all data items given to `logreg.train` could be converted to the `float` data type. Clearly, all data *must* be *numeric* in order to enable the computation of weights and gradients required for logistic regression.

The error arises from the line `X = df_filtered.drop(['class_encoded'], axis=1)` in Listing 1 provided; `X` is prepared by dropping only the '`class_encoded`' column, and therefore will retain the '`species`', '`island`', and '`sex`' columns from `df_filtered`, whose entries are strings.

In our case, '`species`' is not a feature—it is the label we intend to predict—and therefore mustn't even be contained in `X` at all.

We have two options regarding '`island`' and '`sex`'; if we believe the label we are trying to predict, i.e., the species of the penguin, depends on either of these features, then we could form `X` by replacing these columns with numerically encoded versions of them, or, if we believe they have little or no influence onf the label, we could drop them altogether.

It is unlikely that the sex of a penguin has any influence on its species, so we will drop this feature entirely. A penguin's habitat ('`island`'), on the other hand, may be considered as influencing its species, as different species may have adaptations for different habitats. Whether or not to include it as a feature then depends on one's exact requirements. For the purposes of this problem, we will look at both options.

Hence, we resolve the error as follows. We drop the '`species`' and '`sex`' columns entirely from '`df_filtered`' to form `X_`. Then, from `X_` we form `X_1` by one-hot encoding the contents of the '`island`' column and including them as new features, and `X_2` by dropping the column entirely. The relevant lines of code are given in Listing 1.

```
X_ = df_filtered.drop(['species', 'sex', 'class_encoded'], axis=1)

# Form X_1 by one-hot encoding the island names and including them as new
    features
ohe = OneHotEncoder(sparse_output=False, drop='first', handle_unknown='ignore')
islands_encoded = ohe.fit_transform(X_[['island']])
X_1 = pd.concat(
    [
        X_.drop('island', axis=1).reset_index(drop=True),
        pd.DataFrame(islands_encoded, columns=ohe.get_feature_names_out(['island'
            ]))
    ], axis=1
)

# Form X_2 by simply dropping the 'island' column
X_2 = X_.drop(['island'], axis=1)
```

Listing 1: Fix for the error

We then run the code given in Listing 2 to train the logistic regression model with both `X_1` and `X_2`. The outputs generated by each one respectively, are as follows:

```
Accuracy : 0.5813953488372093
[[ 2.75836572e-03 -8.48924277e-05  4.43272550e-04 -2.85530563e-04
   1.84666104e-04 -1.04725548e-04]] [-8.62161382e-06]

Accuracy : 0.5813953488372093
[[ 2.76291513e-03 -8.24983988e-05  4.82711005e-04 -2.87644233e-04]]
 [-8.35428389e-06]
```

The same accuracy has been achieved in both cases, and the coefficients for common features in both cases are very similar. We further note that in both cases, the following warning was raised: `ConvergenceWarning: The max_iter was reached which means the coef_ did not converge.`

Question 03 It is clear that SAGA (Stochastic Average Gradient Ameliorated) has performed poorly on the current dataset, as indicated both by the relatively low accuracy of about 0.581 on its perfor-

mance on the data, and the fact that the coefficients did not converge.

Being a stochastic method, SAGA performs best when the size of the dataset is large, ideally with more than tens of thousands of data points. This is because statistically, the gradient computed using a minibatch of the data points approaches the true gradient only when averaged over sufficiently many samples.

Hence, the main reason for the poor performance of SAGA on our specific dataset is its size. Our dataset consists only of 214 data points—too few for SAGA to perform optimally.

More specifically, we attribute the poor performance of SAGA to the following issues that may arise due to a dataset of this scale:

- the gradient computation has a high tendency to be noisy
- a lot more iterations will be needed in order for any convergence to occur, as indicated by the `ConvergenceWarning`
- the overhead associated with each iteration is not "offset" by a fewer number of iterations, as is usually the case with larger datasets
- sparsity?

Question 04 The same code snippet in Listing 2 given was run using the '`liblinear`' solver in place of '`saga`' solver with training and test sets constructed using both `X_1` and `X_2` as described above, yielding respectively the following outputs;

```
Accuracy : 1.0
[[ 1.51189823 -1.38940877 -0.14156163 -0.00378048  0.72408487 -0.5530247 ]]
 [-0.07450833]

Accuracy : 1.0
[[ 1.59665154 -1.42501103 -0.15238046 -0.003951  ]] [-0.0755452]
```

In each case, an accuracy of 1.0 is reported.

Question 05 '`liblinear`' is a coordinate-descent method which uses all data points to compute the partial derivative with respect to a single coefficient at a time, and updates it over a number of iterations. Hence, it performs better on smaller datasets, with several hundreds of, or a few thousand data points, such as in our case.

As such, '`liblinear`' does not incur as much overhead per iteration as a stochastic method such as SAGA. Thus, when the number of data points is small, it will converge faster than a stochastic method, with lesser memory and resource usage. Therefore, '`liblinear`' outperforms SAGA on our current dataset.

Question 06 The `random_state` parameter passed to a `LogisticRegression` instance from `sklearn` controls how the coefficients will be initialized, and in the case of a stochastic method, how data points will be shuffled and selected randomly for iterations.

Hence, performing different runs with different seed values for `random_state` is expected to give different results and different accuracies, as each value would result in different coefficient initializations, and different minbatches of data points being used for updates.

Question 07 We implement standard scaling for the feature values and re-ran the logistic regression model with both '`liblinear`' and SAGA solvers. The results obtained are as follows:

```
Accuracy with saga : 0.9767441860465116
[[ 3.90443923 -0.82356688  0.18543687 -0.73559796]] [-1.96731639]

Accuracy with liblinear : 0.9767441860465116
[[ 3.77819685 -0.75341497  0.17248526 -0.71597049]] [-1.72205563]
```

Equal accuracies of around 0.976 are reported in both cases; a significant improvement over the

previous value of 0.581 with SAGA, but a marginal decrease compared to the previous value of 1.0 with '`liblinear`'. We also note that both models have led to comparable coefficients.

To see why this leads to a performance increase with SAGA, we note that standard scaling the features of the data points transforms them to be centered around zero, and restricts their variation along comparable scales. With stochastic methods such as SAGA, having all features on comparable scales helps stabilize gradient computation. Had this not been done, a randomly chosen minibatch may have features of a vastly different magnitude range than other batches, and therefore sudden variations/magnitude changes in the gradient, leading to uneven and haphazard steps being taken, slowing down convergence.

Further, scaling ensures that the model learns to compare relative magnitudes *across* features rather than be dominated by features that vary on very large scales, ignoring other features.

In fact, the reason for the slight decrease in accuracy with '`liblinear`' with scaling may be due to the model starting to factor in other features more significantly; the previous accuracy of 1.0 may have been due to some level of overfitting to the data, and it is possible that the model is now starting to exchange some of the lower bias it had previously achieved with a lower variance now, in an attempt to generalize better to unseen data.

Question 08 The stated approach is incorrect.

Label-encoding a feature whose possible values are colors (with more than two colors) would imply some *ordering* and relative *weighting* among the colors, and is therefore unsuitable, unless such an ordering or weighting actually does exist (e.g., 'green' - 'least risk', 'red' - 'highest risk'/'twice as high a risk as green' etc.).

Feature-scaling tends to preserve such an ordering and weighting. It would not address the problem of how exactly to assign labels to the colors.

A more suitable approach would be to use one-hot encoding. Here, we treat each color as a binary-valued feature, and append them as extra features to the existing features. Doing so we avoid the implication of any form of ordering or weighting. Further, the values of these features will always be either 0 or 1, and therefore vary on comparable scales among each other.

3 First and Second Order Methods for Logistic Regression

Question 01 Figure 3 shows a plot of the synthetically generated data.

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8096\4135035922.py in ?()
      2 X_train, X_test, y_train, y_test = train_test_split (X, y, test_size=0.2, random_state=42)
      3
      4 # Train the logistic regression model. Here we use sagasolver to learn weights.
      5 logreg = LogisticRegression(solver='saga')
----> 6 logreg.fit(X_train, y_train)
      7
      8 # Predict on the testing data
      9 y_pred = logreg.predict(X_test)

d:\UoM\Semester 5\EN3150 - Pattern Recognition\Assignment 02\.venv\Lib\site-packages\sklearn\base.py in ?(estimator, *args, **kwargs)
   1361                 skip_parameter_validation=(
   1362                     prefer_skip_nested_validation or global_skip_validation
   1363                 )
   1364             ):
-> 1365                 return fit_method(estimator, *args, **kwargs)

d:\UoM\Semester 5\EN3150 - Pattern Recognition\Assignment 02\.venv\Lib\site-packages\sklearn\linear_model\_logistic.py in ?(self, X, y, sample_weight)
   1243             _dtype = np.float64
   1244         else:
   1245             _dtype = [np.float64, np.float32]
   1246
-> 1247         X, y = validate_data(
...
   2169         else:
   2170             arr = np.array(values, dtype=dtype, copy=copy)
   2171
ValueError: could not convert string to float: 'Adelie'
```

Figure 2: Data generated

Question 02 We simply initialize the weights to zero.

The binary cross entropy loss is a convex function. Hence, with a suitable learning rate, we can guarantee convergence to the global optimum, irrespective of the initial weights chosen.

With neither prior information about the nature of the weights, nor any constraints placed on the initial weights by the nature of the loss function, we have no reason to prefer any specific initialization over another.

Therefore, for simplicity, we simply initialize the weights to zeros in both cases.

Question 03 We use the binary cross entropy loss;

$$\text{BCE}\left(\mathbf{y}, \hat{\mathbf{y}}\right) = -\sum_{i=1}^{N} y_i \log\left(\hat{y}_i\right) + \left(1 - y_i\right) \log\left(1 - \hat{y}_i\right) = -\mathbf{y}^\top \log\left(\hat{\mathbf{y}}\right) - \left(\mathbf{1} - \mathbf{y}\right)^\top \log\left(\mathbf{1} - \hat{\mathbf{y}}\right),$$

where $\mathbf{y}_{N\times 1}$ and $\hat{\mathbf{y}}_{N\times 1}$ contain respectively the actual class labels and the predicted probabilities that a data point belongs to the positive class, and $\mathbf{1}_{N\times 1}$ is a vector of all 1s.

We choose it mainly for the following reasons:

- it follows naturally from the expression for the likelihood of observing the data points given the predicted probabilites, and hence has a probabilistic foundation
- it penalizes incorrect predictions more heavily
- it is convex

Question 04 The implementation is given in Listing .

Question 05 The required plot is given in Figure 3.

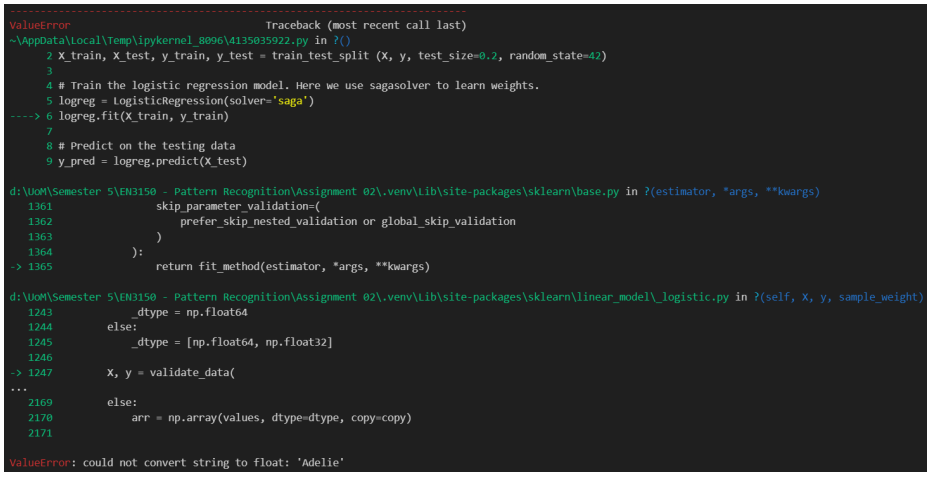

Figure 3: Loss vs iteration count with batch gradient descent and Newton's Method

Clearly, in both cases, the loss can be seen to be decreasing with the number of iterations, indicating that the algorithm is indeed seeking an optimal set of weights. However, the loss decreases far more slowly with batch gradient descent, when compared with Newton's Method.

Newton's Method is derived from a quadratic approximation to the loss function, and hence convergences quadratically, whereas batch gradient descent is derived from a linear approximation and therefore converges only linearly, more slowly than Newton's Method.

Gradient descent looks only at the slope of the function at a point, and does not take into account the variation of the slope within the its neighborhood. Newton's Method, on the other hand, uses higher-order derivatives in the Hessian as well, in deciding the best direction to take a step in.

For example, although the gradient in some direction a may be the steepest compared to another direction b, it may be the case that traversing further along b leads to a path of much steeper descent

towards a minimum—a path that would not have been discovered had the exploration been done in the direction of a.

We note however that Newton's Method comes at the increased computational cost of computing and inverting the Hessian of the loss function, and is therefore unsuitable for complicated, high-dimensional functions.

Question 06 The following stopping criteria may be used:

- a specified maximum number of iterations is reached
- the loss function is below a certain threshold
- the percentage reduction of the loss function is less than a certain threshold
- the percentage change in the norm of the weights vector is less than a certain threshold
- whichever of the above happens first

Question 07 The newly generated data points appear as shown in Figure 3.

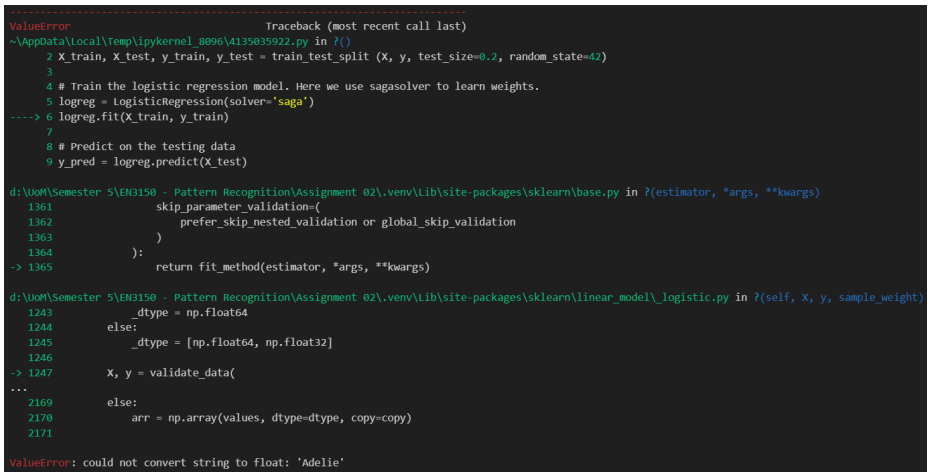

Figure 4: Data generated with different centers

Observe that in this case there is no clear separation between the classes. There is significant overlap between the two classes, and there is no clear boundary dividing them.

We ran batch gradient descent on this set of data with various values for the learning rate. The variation of the loss with several chosen values for the learning rate is illustrated in Figure 3.

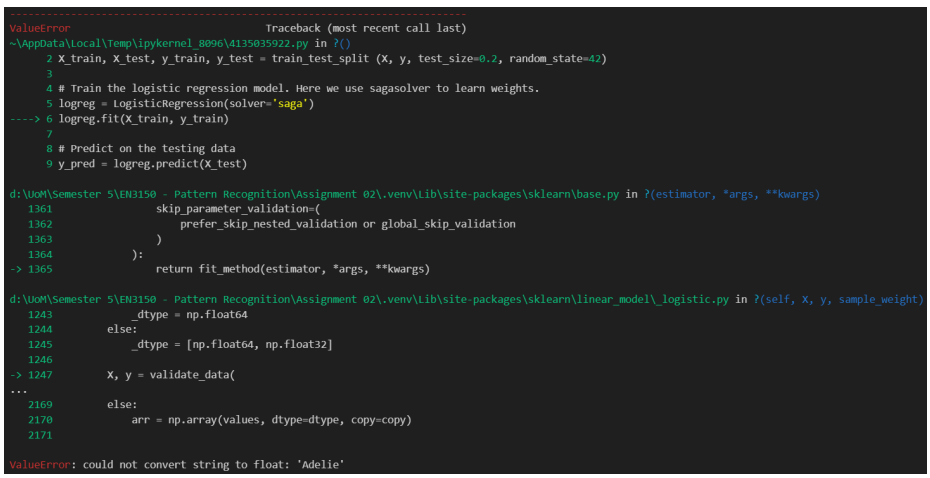

Figure 5: Loss against iteration count with different learning rates

It can be seen that for all choices of the learning rate, the loss seems to saturate just below 1000, and does not seem to decrease to zero.

To see why this happens, note that from how the problem is set up, we are trying to discover a linear separation/boundary between the two classes. However, as obviously visible in Figure 3, no single straight line can separate the two classes; any straight line chosen as a boundary will always lead to misclassifications.

From a Bayesian standpoint, we classify a data point $\mathbf{x}$ as belonging to Class $y = 1$, if the posterior probability

$$p\left(y = 1|\mathbf{x}\right) = \frac{p\left(\mathbf{x}|y = 1\right)p\left(y = 1\right)}{p\left(\mathbf{x}|y = 1\right)p\left(y = 1\right) + p\left(\mathbf{x}|y = 0\right)p\left(y = 0\right)} > p_{\text{thresh}},$$

where p_{thresh} is some threshold probability.

However, in the case of overlapping classes, $p\left(\mathbf{x}|y = 1\right)$ and $p\left(\mathbf{x}|y = 0\right)$ are both of comparable magnitude; specifying the class does not make it more or less likely to see $\mathbf{x}$ within that class, as the classes overlap and $\mathbf{x}$ is almost equally likely to be seen in either class.

In such a case,

$$p\left(y = 1|\mathbf{x}\right) \approx \frac{p\left(y = 1\right)}{p\left(y = 1\right) + p\left(y = 0\right)} = p\left(y = 1\right),$$

i.e., the data point will have little to no effect at all on the prediction being made. Given some arbitrary $\mathbf{x}$ then, the model would simply predict it as belonging to Class 1 with the same probability that an observation from Class 1 is made in general.

As a result, as long as data points from Class 0 still exist, *any* model will *always* make misclassifications. This has the effect of imposing a strict positive lower bound on the loss achievable; no loss function can be minimized beyond this bound.

More concretely, in the case of batch gradient descent, the loss function at such points will remain more or less constant—almost independent of the parameters we differentiate with respect to, leading to almost 0 gradients, and hence a stagnation of the loss.