Logistic Regression Classifier

Andrea Pagotto

February 2, 2018

1 Introduction

Logistic regression is a discriminative graphical model, meaning that it directly optimizes for $p(y|x)$. Logistic regression is a modification of linear regression, by substituting the hypothesis function of a linear mapping, to the sigmoid function. The use of the sigmoid function for hypothesis results in outputting values between 0 and 1, which are then translated to either 0 or 1 based on a threshold (of 0.5) for the binary classification task. The hypothesis with the sigmoid function is:

$$p(y = 1|x) = h_\theta = \frac{1}{1 + \exp(-\theta^T x)}$$

$$p(y = 0|x) = 1 - h_\theta$$

Note in our case $y = 1$ corresponds to class 2, and $y = 0$ corresponds to class 1.

To find the parameters θ, the maximum likelihood estimation is used. Training a model to find the parameters can be done by maximizing the conditional log likelihood of the training data. The log is used to ensure the cost function will be convex, with one global optimum point.

The logistic regression cost function is:

$$Cost(h_\theta(x), y) = y * (-\log(h_\theta(x))) + (1 - y) * (-\log(1 - h_\theta(x)))$$

For a training set with m points, the likelihood of the parameters is:

$$L(\theta) = p(Y|X; \theta)$$

Where, the L represents the likelihood, the Y and X represent the class and data for all the points in the training set. Using the chain rules for probabilities and the previously given equation for $p(y^i|x^i; \theta)$, this equation can be written as:

$$L(\theta) = \prod_{i=1}^{m} (h_\theta(x^i)^{y^i})(1 - h_\theta(x^i))^{1-y^i}$$

This can be transformed into the log likelihood:

$$l(\theta) = \log L(\theta) = \sum_{i=1}^{m} y^i \log(h_\theta(x^i)) + (1 - y^i) \log(1 - h_\theta(x^i))$$

The objective will be to maximize the log likelihood, and this can be done using gradient descent towards the global optimum.

In order to optimize the parameters, this can be done iteratively using gradient descent, to update the parameters at each step up the gradient. The update equation is:

$$\theta := \theta + \alpha \nabla_\theta l(\theta)$$

By taking gradient of the log likelihood function, this equation can be simplified to:

$$\theta := \theta + \alpha(y^i - h_\theta(x^i))x^i_j$$

This is the equation that was implemented in the code for gradient descent. The final parameters from the training are taken once the gradient has approached 0. This procedure can be visualized in the following diagram, where J represent the cost function.

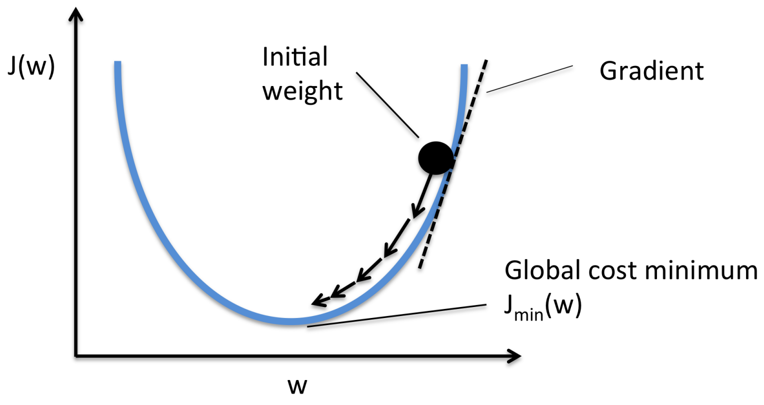


Figure 1: Gradient Descent

2 Training and Testing

2.1 Training

This section will show step by step the code that was used to execute the training of the model. This code was also consolidated into one function "logistic_regression" in the submitted python file. To run this file that was submitted, just run "logisticregression.py" with python 3, and the desired data file. This code is also implemented in a runnable jupyter notebook that was also submitted.

Loading Files First the files are loaded from the matlab data file an formatted into the format required for use in python. The output of the format data function will be the training features and the class IDs in a list. In this code, 0 was used to represent class 1, and 1 was used to represent class 2. The features list, is a list of the feature vectors. Note the format data function also adds an extra column as required to enable setting parameters for the intercept, so that the parameters found are not restricted to passing through the origin.

```
mat = scipy.io.loadmat('classify_d5_k3_saved2.mat')
features, classIDs = format_data(mat)
```

To begin logistic regression, first set the required parameters, of the learning rate and the max number of steps for gradient descent. These parameters were set based on looking at recommendations from research on-line and also from trail and error testing and checking results.

```
num_steps = 50000
learning_rate = 0.0001
```

Gradient descent for the parameters, θ, optimization will be used to update the parameters with the following equation:

$$\theta := \theta + \alpha(y^i - h_\theta(x^i))x^i_j$$

Where α is the learning rate set in the previous section, the x are the points in the training set, the y is the class, and h is the prediction of the class. The predictions h are calculated using the sigmoid function.

This equation will be executed in the following code. The log likelihoods will be printed out as it proceeds to track the progress. Also, as the gradient size decreases, indicating it is approaching the minimum (ie. close to 0), the loop will be terminated if it has not yet reached the maximum number of iterations.

```
# gradient descent
for step in range(0, num_steps):
    scores = np.dot(features, params)
    predictions = sigmoid(scores)

    # Update weights with log likelihood gradient
    errors = classIDs - predictions

    gradient = np.dot(features.T, errors)
    params += learning_rate * gradient

    if np.linalg.norm(gradient) < 0.1:
        print("Gradient is close to 0, reached convergence.")
        print("Final Loglikelihood:")
        print(log_likelihood(features, classIDs, params))
        break

    # Print log-likelihoods
    if step % 10000 == 0:
        print(log_likelihood(features, classIDs, params))
```

The output from running this code is as follows:

```
-1633.26968675
-691.28043225
-689.915019929
-689.760198017
Gradient is close to 0, reached convergence.
Final Loglikelihood:
-689.742275396
```

2.2 Testing

First the data is formatted for the testing. This is done the same way as previously, however train is set to false, which means it will take data points not yet used in the training set. The train/test split is 80/20. The test data is taken as the last 20% of the data, and since this data was randomly generated, this method is acceptable (ie. it does not need to be shuffled before hand because it was already generated randomly). This is shown in the following code.

```
test_features, test_classIDs = format_data(mat, train = False)
```

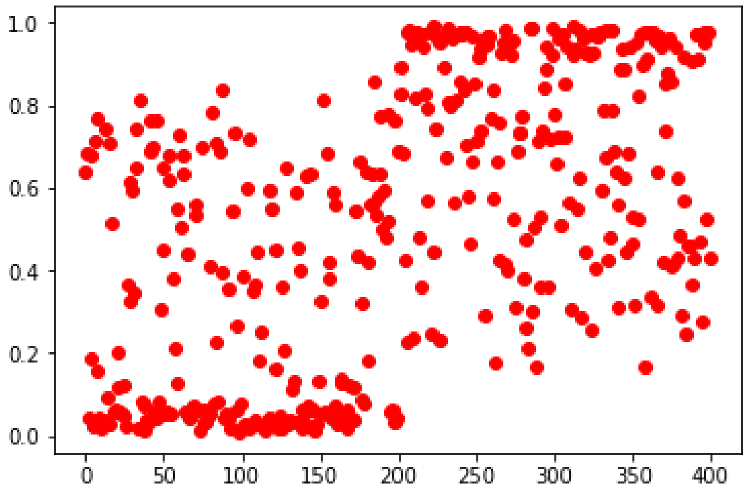
The results from one dataset is shown below. The final class predictions are obtained from the sigmoid. The cut-off threshold to associate class 0 or 1 with a given point is set at 0.5, and this is accomplished by the *round* function. The data is also plotted to visualize the success of the classification. The left half of the graph should be class 0 and the right half of the graph should be class 1. The accuracy is calculated as the number of correctly classified points after the rounding.

```
test_scores = np.dot(test_features, params)
preds = sigmoid(test_scores)
acc = sum(np.round(preds) == test_classIDs) / len(preds)

plt.plot(preds, 'ro')
plt.show()

print("Accuracy:")
print(acc)
```

The following graph shows the results of the predictions for the data. The left half of the graph should be 0 and the right half of the graph should be 1. Values are approximated from these predictions by rounding to either 0 or 1.



The final accuracy calculated from comparing to the given labels is:

```
Accuracy:
0.7175
```

2.3 Comparison to skLearn

We compared to the accuracy of sklearn to validate the success of our implementation of this algorithm. It is clear that the accuracies from our implementation is similar to the accuracy from sklearn. Sklearn logistic regression was implemented as shown in the following code. Note, the first column of the data is excluded because this was the added column necessary for the intercept, but the sklearn function has a built in option to do this. Also, note that sklearn has default regularization built in, so the regularization inverse constant C was set to $1e15$ to make the regularization negligible to make it a direct comparison to our implementation.

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(fit_intercept=True, C = 1e15)
clf.fit(features[:,1:], classIDs)

acc_sklearn = clf.score(test_features[:,1:], test_classIDs)

print('Accuracy from sk-learn: ')
print(acc_sklearn)
```

It is clear from the resulting accuracy that the sklearn implementation performed very similar to our implementation. This is the same on all datasets as shown in the results section.

```
Accuracy from sk-learn:
0.72
```

Also, using sklearn, we tested many different parameter settings including different regularizers, different learning methods and learning rates, however there was no significant improvement found in the results accuracy.

3 Results

From the following table, it is clear the custom implementation of logistic regression is performing the same as the library function from sklearn. Further analysis of results will be discussed in the following section.

Table 1: Logistic Regression Results.

File	Accuracy	sklearn Accuracy
classify_d3_k2_saved1.mat	0.655	0.655
classify_d3_k2_saved2.mat	0.64	0.64
classify_d3_k2_saved3.mat	0.6825	0.6825
classify_d4_k3_saved1.mat	0.7325	0.7325
classify_d4_k3_saved2.mat	0.7275	0.7275
classify_d5_k3_saved1.mat	0.7175	0.7175
classify_d5_k3_saved2.mat	0.7175	0.72
classify_d99_k50_saved1.mat	1.0	1.0
classify_d99_k50_saved2.mat	1.0	1.0
classify_d99_k60_saved1.mat	1.0	1.0
classify_d99_k60_saved2.mat	1.0	1.0

3.1 Observations

Logistic regression (LR) struggles with the datasets with three dimensional feature vectors versus those with 99 dimensional features. At first this seems counter intuitive, but when we analyze the dataset's principle components (see Figure 2) we see a pattern that would be difficult to fit with a linear model as we have trained in our logistic regression implementation.

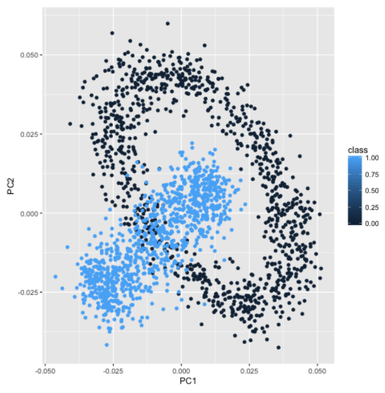


Figure 2: A visualization of the principle components of $classify\_d3\_k2\_saved1$. This class separation would be a challenge for a linear model. The data has been rotated in three dimensions to demonstrate a plausible separation between classes.

Looking at Figure 2 we would expect poor performance from a linear model. Compare this with an analysis of the higher dimensional dataset, we can see a pattern that lends itself to linear separation, as reflected in the reported accuracy of 100% in our LR model during testing.

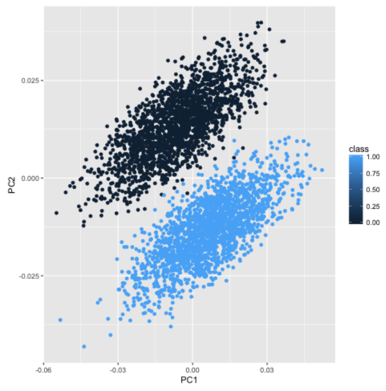


Figure 3: A visualization of the 99-dimensional data present in the $classify\_d99\_k60\_saved2$ dataset. We observe a clear separation between classes after performing a manual rotation in three dimensions

4 Conclusions

With no a priori information it is very difficult to select an appropriate classification scheme for a given dataset. Logistic regression is chosen as a baseline classifier and has shown variable performance across various datasets. We performed a PCA analysis and identified properties of the datasets that explained the results.