AUTHOR
Jamie Hackney

PUBLISHED
April 1, 2024

Logistic Regression Algorithm Source Code: logistic.py

Abstract

This blog post contains code implementations for the Logistic Regression algorithm, as well as visualizations and discusssions on the algorithm. The goal of this blog post was to explore the Logistic Regression algorithm as a linear classifier to understand how it functions as well as its advantages and limitations. Specifically, Logistic Regression with momentum gradient descent was implemented. The algorithm was run and analyzed on synthetically generated data to see the effect of momentum on convergence speed as well as susceptibility to overfitting. While most code is my own, credit goes to Professor Phil Chodrow for providing some additional code for creating some of the visualizations and generating data - acknowledgements are given at specific code blocks.

```
%load_ext autoreload
%autoreload 2
from logistic import LogisticRegression, GradientDescentOptimizer
import torch
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

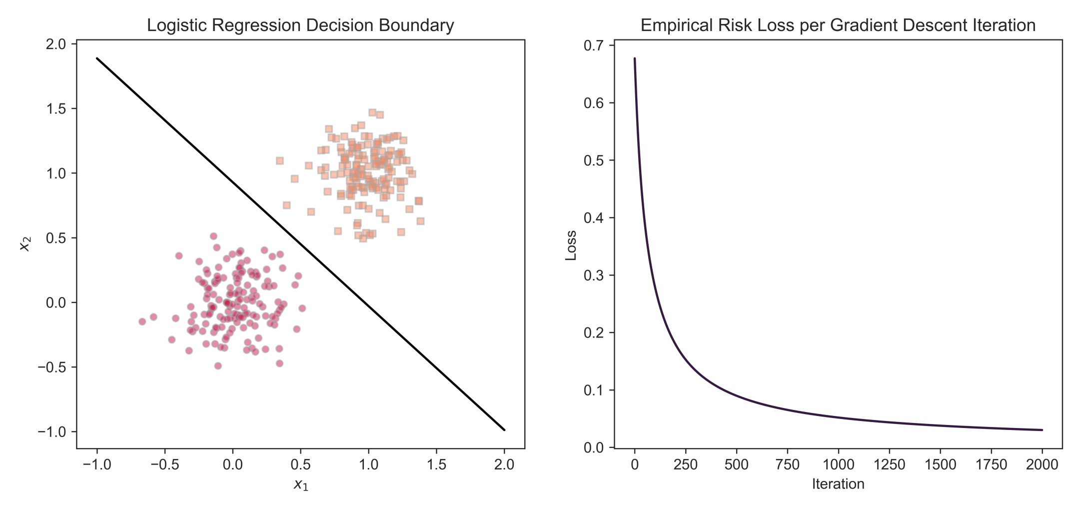
This code block contains all the functions used in this post.

```
sns.set_style("ticks")  # Set plot style
sns.set_palette("rocket")     # Set color palette

def classification_data(n_points = 300, noise = 0.2, p_dims = 2, seed=None):

    if seed:
        torch.manual_seed(seed)

    y = torch.arange(n_points) >= int(n_points/2)
    y = 1.0*y
    X = y[:, None] + torch.normal(0.0, noise, size = (n_points,p_dims))
    X = torch.cat((X, torch.ones((X.shape[0], 1))), 1)
    return X, y


# plot the loss over each iteration
def plot_loss(loss_vec, ax):
    sns.lineplot(data=loss_vec, ax=ax)


# plot the accuracy over each iteration
def plot_accuracies(accuracies, ax):
    sns.lineplot(data=accuracies, ax=ax, hue='data')


# code modified from Prof. Chodrow - plot regression data
def plot_regression_data(X, y, ax):
    assert X.shape[1] == 3, "This function only works for data created with p_dims == 2"
    targets = [0, 1]
    markers = ["o" , ","]
    for i in range(2):
        ix = y == targets[i]
        ax.scatter(X[ix,0], X[ix,1], s = 20,  c = y[ix], facecolors = "none", edgecolors


# code supplied by Prof. Chodrow - draw a classifying line
def draw_line(w, x_min, x_max, ax, **kwargs):
    w_ = w.flatten()
    x = torch.linspace(x_min, x_max, 101)
    y = -(w_[0]*x + w_[2])/w_[1]
    l = ax.plot(x, y, **kwargs)


# train an LR model and return vector of losses over iteration
def gradient_descent(opt):
    loss_vec = []
    for _ in range(2000):
        loss = opt.step(X, y)
        loss_vec.append(float(loss))

    return loss_vec


# plot the classifying line found by the LR model next to the loss of the model over eac
def plot_descent(loss_vec, LR, X, y):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5), dpi=600)

    plot_regression_data(X, y, axes[0])
    draw_line(LR.w, -1, 2, axes[0], color = "black")
    axes[0].set(xlabel = r"$x_1$", ylabel = r"$x_2$", title="Logistic Regression Decisio

    plot_loss(loss_vec, axes[1])
    axes[1].set(xlabel = "Iteration", ylabel = "Loss", title="Empirical Risk Loss per Gr


# plot two losses on the same graph
def compare_losses(loss_vec_1, loss_vec_2):
    _, ax = plt.subplots(figsize=(12, 5), dpi=600)

    ax.plot(loss_vec_1, label='Vanilla Gradient Descent')
    ax.plot(loss_vec_2, label='Gradient Descent with Momentum')

    ax.set_xscale('log')

    ax.set(xlabel="Iteration", ylabel="Loss", title="Vanilla Gradient Descent vs Gradien
    ax.legend()
```

Vanilla Gradient Descent

First, to check the implementation, we try vanilla gradient descent. To do so, we first create some random, synthetic classification data. Here, we generate 300 noisy observations belonging to two classes; each observation has 2 features. We instantiate a LogisticRegression model with a learning rate of $\alpha = 0.1$ and a momentum of $\beta = 0$. Since $\beta = 0$, this logistic regression model is the same as one implemented with vanilla gradient descent. We train the model and then plot its decision boundary and the evolution of loss over time. We can see that visually, the model seems to have learned a weight that corresponds to a correct classifying line. This is supported by seeing that the loss of the model decreases over time.

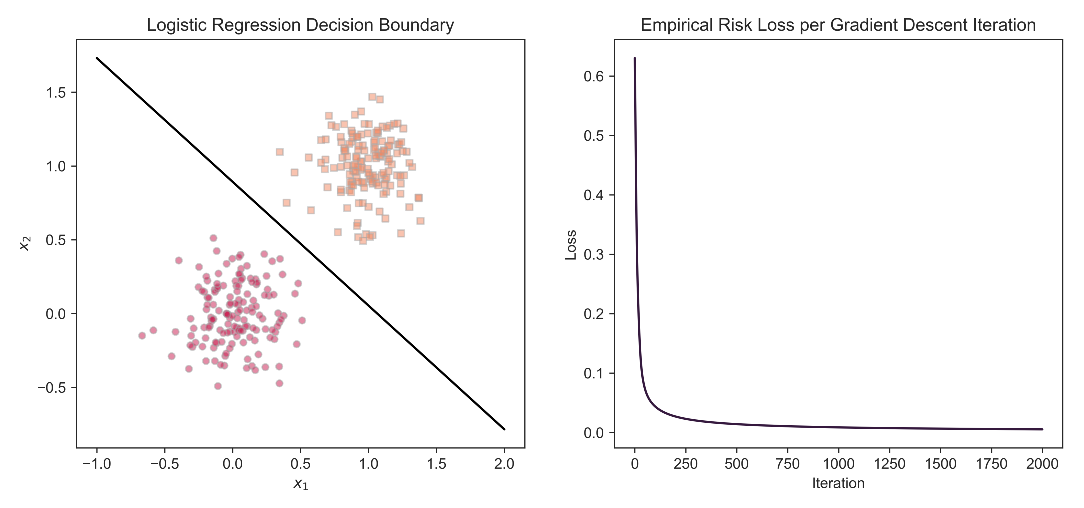
```
# make some data
X, y = classification_data(n_points=300, noise=0.2, p_dims=2, seed=1)

# build a model
LR = LogisticRegression()
opt = GradientDescentOptimizer(LR, learning_rate=0.1, momentum=0)

vanilla_loss = gradient_descent(opt)
plot_descent(vanilla_loss, LR, X, y)
```



Benefits of Momentum

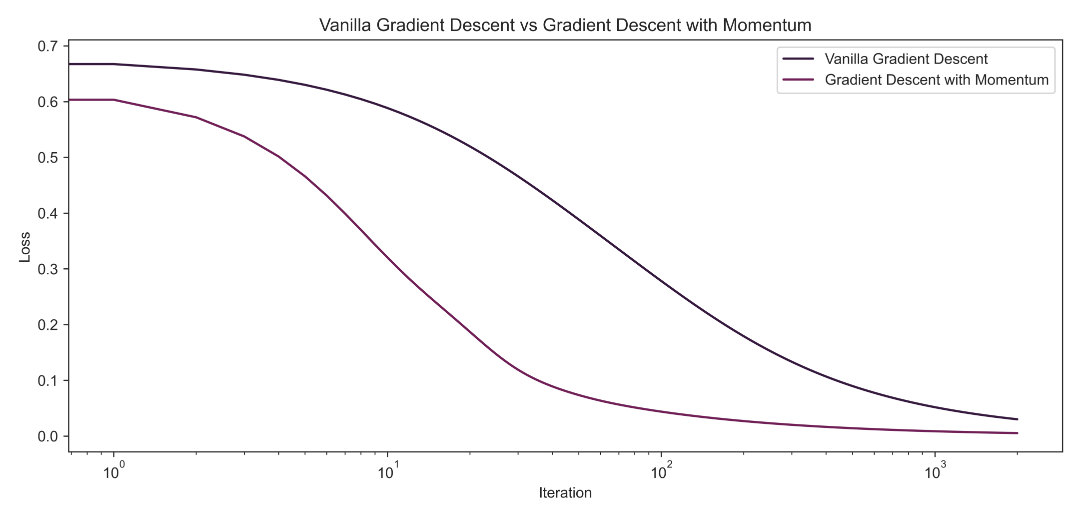
Now, to see the benefits of momentum in gradient descent, we train a logistic regression model again on the same data, but this time with $\beta = 0.9$. We can see visually that the model has found a weight vector that corresponds to a correct classifying line.

```
# build a model
LR = LogisticRegression()
opt = GradientDescentOptimizer(LR, learning_rate=0.1, momentum=0.9)

momentum_loss = gradient_descent(opt)
plot_descent(momentum_loss, LR, X, y)
```



As the plot below shows, the loss decreases much faster with Gradient Descent with Momentum than it does with Vanilla Gradient Descent, showing that this method of gradient descent is able to find the correct weight vector in fewer iterations.

```
compare_losses(vanilla_loss, momentum_loss)
```



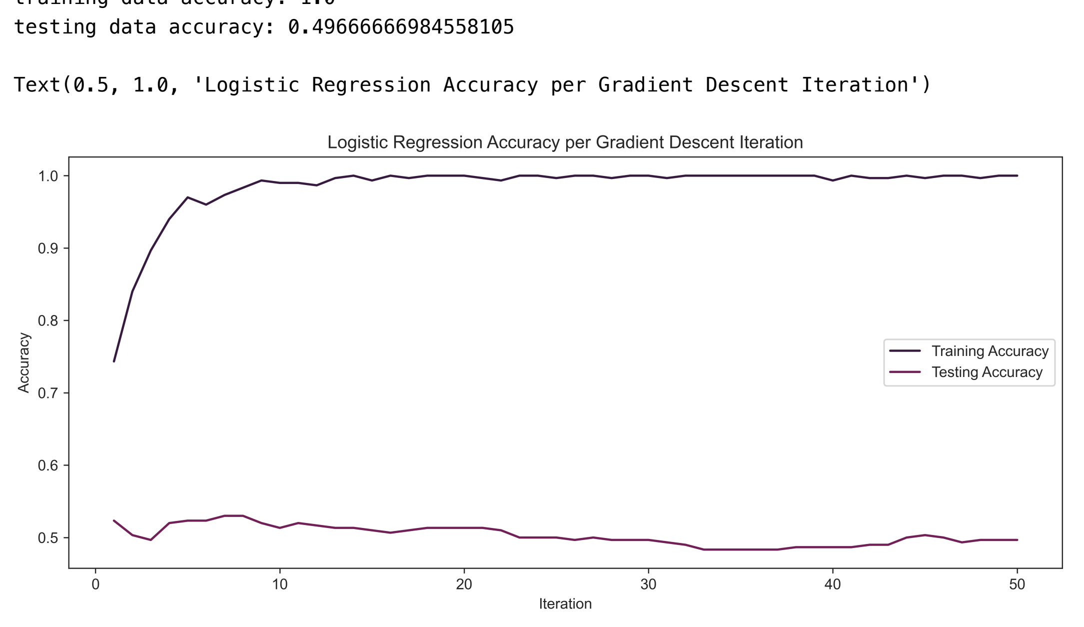
Overfitting

Finally, to see how susceptible the model is to overfitting, we generate some more random synthetic data. This time, we generate a training and testing set, which are both made of 300 observations and 400 features. We train a model on the training data, achieving an accuracy of 100% a little bit after the 30th iteration. Then, we evaluate the model on the testing data and see that it only achieves an accuracy of 51%. This shows that under certain circumstances, particularly when the number of features is larger than the number of observations, the model can learn to overfit the training data.

```
X_train, y_train = classification_data(n_points=300, noise=100, p_dims=400, seed=1)
X_test, y_test = classification_data(n_points=300, noise=100, p_dims=400, seed=2)

LR = LogisticRegression()
opt = GradientDescentOptimizer(LR, learning_rate=0.1, momentum=0.9)

# train the model
train_accuracies = []
test_accuracies = []
for _ in range(50):

    # complete a gradient descent iteration
    loss = opt.step(X_train, y_train)

    # calculate model training accuracy
    train_preds = LR.predict(X_train)
    train_accuracy = (1.0 * (train_preds == y_train)).mean()
    train_accuracies.append(float(train_accuracy))

     # calculate model testing accuracy
    test_preds = LR.predict(X_test)
    test_accuracy = (1.0 * (test_preds == y_test)).mean()
    test_accuracies.append(float(test_accuracy))

# evaluate model accuracy on training data
train_preds = LR.predict(X_train)
train_accuracy = (1.0 * (train_preds == y_train)).mean()

# run prediction using the trained model
test_preds = LR.predict(X_test)
test_accuracy = (1.0 * (test_preds == y_test)).mean()

print(f"training data accuracy: {train_accuracy}")
print(f"testing data accuracy: {test_accuracy}")

fig, ax = plt.subplots(1, 1, figsize=(12, 5), dpi=600)

iterations = range(1, len(train_accuracies) + 1)
sns.lineplot(x=iterations, y=train_accuracies, label='Training Accuracy', ax=ax)
sns.lineplot(x=iterations, y=test_accuracies, label='Testing Accuracy', ax=ax)

plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Logistic Regression Accuracy per Gradient Descent Iteration')
```

```
training data accuracy: 1.0
testing data accuracy: 0.49666666984558105
```

```
Text(0.5, 1.0, 'Logistic Regression Accuracy per Gradient Descent Iteration')
```



Discussion

Throughout this implementation we have seen different implementations of the Logistic Regression linear classifier. First, we verified that the vanilla gradient descent implementation was working by plotting the decision line against the data as well as the loss over time. Then, we implemented momentume gradient descent and showed that method is able to find the weight vector corresponding to the correct classifying line faster than vanilla gradient descent. Finally, we also showed that the model is susceptible to overfitting in some circumstances. In particular, when the number of features in the dataset exceeds the number of observations, the model finds a weight vector that achieves 100% accuracy on the training data, but only 51% accuracy on the testing data.