Logistic_Regression_Implementation

Summary

The code defines a function `get_logistic_regression(custom_implementation, data)` where `custom_implementation` is a binary that allows you to toggle between a custom implementation and one using sklearn's libraries.

The `custom_logistic_regression` function is designed for training a simple logistic regression model with various hyperparameters. The model is trained iteratively using gradient descent until convergence or reaching the maximum number of iterations. Throughout training, the algorithm updates the model parameters, evaluates the cost function, and monitors convergence criteria. Finally, the trained model is evaluated on the test set, and both accuracy and the area under the ROC curve (AUC) are calculated to assess its performance.

The function employs several helper functions, including those for logistic regression operations (performing predictions and updating parameters), computing the cost function, generating polynomial features, and checking for data leakage. Additionally, it handles certain debugging and visualization tasks, such as plotting the cost over iterations. Despite encountering challenges with continuous variables and issues related to data leakage, it demonstrates a comprehensive approach to implementing logistic regression. Though I still ran into issues with an ever-increasing cost, the overall accuracy was quite high. Whether this is due to correct configuration or simply a stroke of luck is beyond the scope of this project at this time, but is something I would like to explore in the future.

I tried to use sklearn's logistic regression model after implementing my own version. However, I was running into issues with train_test_split returning the wrong labels and would either get extremely low scores, or overfit so much I had 100% accuracy. For example:

```
Beginning sklearn implementation...
sklearn model accuracy: 0.14285714285714285 auc: 0.5
```

Implementation Overview

Imports

```
# Imports
import random
import warnings

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LogisticRegression, Lasso, Ridge, ElasticNet
from sklearn.metrics import
accuracy_score,classification_report,confusion_matrix,ConfusionMatrixDisplay
from sklearn.metrics import roc_auc_score
```

Helpers

These are functions to do small things or help me organize my thoughts as I built it out. For the most part, these all apply to the custom implementation only, as sklearn has all of these built in.

List of Helpers:

- load_data()
- shuffle_and_split()
- get_features_and_labels()
- generate_polynomial_features()
- compute_cost()
- perform_logistic_regression()
- update_theta()
- plot_cost()
- get_logistic_regression()
- run()

load_data()

can have view=True to print some X and y values for visual inspection. Otherwise, it just loads the dataset, concatenates X, y and then returns them. We reshape y to avoid a ValueError.

```
# Load
def load_data(view=False):
  X, y = load_breast_cancer(return_X_y = True)
  # Concatenate
  y = y.reshape(-1, 1)
  data = np.hstack((X, y))
  if view:
    print(f"X is of type {type(X)}, y is {type(y)}.")
    print(f"Some X examples:")
    print(X[:10])
    print(f"Some y examples:")
    print(y[:10])

  return data
```

shuffle_and_split()

can have view=True to print out a temporary slice of data and then shuffle and reprint it. This is useful when ensuring only rows are shuffled.

```
# Shuffle and Create Split Training/Dev/Test 60/20/20%
def shuffle_and_split(data, view=False):
  np.random.shuffle(data)
  total_sample = len(data)
  train = data[:int(total_sample*0.6)]
  dev = data[int(total_sample*0.6):int(total_sample*0.8)]
  test = data[int(total_sample*0.8):]

  if view:
    print("Verify shuffling is ok:")
    temp = data[:5]
    print("Initial Data:")
    print(temp)
    print()
    print("Shuffled:")
    np.random.shuffle(temp)
    print(temp)

  return total_sample, train, dev, test
```

get_features_and_labels()

does exactly that, it takes that data and extracts the features and labels, reshaping y along the way.

```
# Get Features and Labels
def get_features_and_labels(data):
  features = data[:, :-1]
  labels = data[:, -1]

  # Reshape labels
  labels = labels.reshape(-1, 1)

  return features, labels
```

generate_polynomial_features()

generates polynomial features up to a specified degree for each feature in the input array X.

```
# Generate Polynomial Features
def generate_polynomial_features(X, degree):
    num_samples, num_features = X.shape
    X_poly = np.zeros((num_samples, num_features * degree))
    for i in range(num_samples):
        for j in range(num_features):
            for d in range(1, degree + 1):

                # Replace value at location with poly
                X_poly[i, j * degree + d - 1] = X[i, j] ** d
    return X_poly
```

compute_cost()

computes the cost for the sigmoid / logistic regression implementation. *Note* that I added epsilon, a very small value that will prevent log(0) errors.

```
# Cost Function
def compute_cost(Y_pred, Y_true, length):
  m = length
  epsilon = 1e-15  # small value to prevent log(0)
  J = -1 / m * np.sum(Y_true * np.log(Y_pred + epsilon) + (1 - Y_true) * np.log(1 -
Y_pred + epsilon))

  return J
```

perform_logistic_regression()

A simple regression model with a sigmoid function where $0 \leq$ prediction ≤ 1.

```
def perform_logistic_regression(theta, input):
  z = np.dot(input, theta)
  prediction = 1 / (1 + np.exp(-z))

  # DEBUGGING
  # print(f'SHAPE OF INPUT: {input.shape}')
  # print(f'SHAPE OF THETA: {theta.shape}')
  # print(f'SHAPE OF PREDICTIONS: {prediction.shape}')

  return prediction
```

update_theta()

Calculates gradient descent and updates theta accordingly. This can take in different penalties. I had to do a weird workaround when updating the coefficients as I was getting multiple ValueErrors when very similar code was successfully used in polynomial regression with regularization. This is something I plan to explore.

```
# Gradient Descent
def update_theta(theta, X, Y_true, Y_pred, learning_rate, length, penalty='l1',
alpha=0.1):
  m = length
  number_features = X.shape[1]

  # compute regular gradient
  error = Y_pred - Y_true
  gradient = (np.dot(X.T, error) / m).T

  # update based on penalty
  if penalty == 'l1':
    regularized_gradient = gradient + alpha * np.sign(theta)
  elif penalty == 'l2':
    regularized_gradient = gradient + alpha * theta
  elif penalty == 'l12':
    l1_gradient = alpha * np.sign(theta)
    l2_gradient = alpha * theta
    regularized_gradient = gradient + l1_gradient + l2_gradient

  else:
    regularized_gradient = gradient
  updated_theta = np.zeros_like(theta)

  # ValueError: setting an array element with a sequence ??
  regularized_gradient = regularized_gradient.flatten()

  # Iterate over each coefficient and update based on error
  for j in range(len(theta)):

    # DEBUGGING
    # print(f'type of learning rate: {type(learning_rate)}')
    # print(f'type of theta: {type(theta)}')
    # print(f'type of updated_theta: {type(updated_theta)}')
    # print(f'tpye of gradient: {type(regularized_gradient)}')
    # print(f'shape of gradient: {regularized_gradient.shape}')



    updated_theta[j] = theta[j] - learning_rate * regularized_gradient[j]
  return updated_theta
```

plot_cost()

Plots the custom implementation's cost over iterations. I had this in because I realized the cost was always increasing (see this image). The code does not currently make any adjustments to the hyperparameters in the case of increasing cost, but it can be easily implemented with the way the code has been structured.

```
# Plot Cost
def plot_cost(k, iterations, cost_history, name):
  plt.plot(iterations, cost_history)
  plt.title(f'{name} set cost Over {k} Iterations')
  plt.xlabel('Iteration')
  plt.ylabel('Cost')
```

get_logistic_regression()

This basically just exists to adjust parameters during development. Some things, like reg_params for loops/sklearn, were not used for this simple implementation but I had added during the planning phase.

```
def get_logistic_regression(custom_implementation, data):
  # Parameters
  iterations = 1000
  penalties = [None, 'L1', 'L2', 'L12']
  reg_params = [random.uniform(0, 2) for _ in range(5)]
  # polynomial degrees greater than 4 caused me to run out of RAM and crash colab
  polynomial_degrees = 3



  if custom_implementation:
    print('Beginning custom regression implementation...')
    # Why was I sending reg_params? For gradient descent? In any case, not used currently
    custom_logistic_regression(data, iterations, penalties, reg_params,
polynomial_degrees)

  else: # use sklearn
    print('Beginning sklearn implementation...')
    sklearn_logistic_regression(data, iterations, penalties, reg_params,
polynomial_degrees)
```

run()

This cell makes everything happen. Decided when you want custom_implementation or not by setting True or False. Having everything wrapped like this made it easy to catch errors.

```
def run():
  try:
    # set True/False for custom_implementation
    get_logistic_regression(True, data)
  except ValueError as e:
    print(f'An error occurred: {e}')
run()
```

Running Custom Implementation

If `get_logistic_regression()` is called with the first argument True, the custom regression is performed on the data. This process starts by getting and splitting the data. Each of the steps is explained in the helper section above.

```
def custom_logistic_regression(data, iterations, penalties, reg_params,
polynomial_degree):
```

```python
# Get splits and initialize tracking stuff
total_sample, training_set, dev_set, test_set = shuffle_and_split(data)
k, cost_change, cost, cost_history, iteration_count = 0, 0, float('inf'), [], []

learning_rate = 0.0001
converged = False

# Check Data
data_leaking_check(training_set, dev_set)

# Get features/labels
X_train, y_train = get_features_and_labels(training_set)
print(f'X_train {X_train}')
print(f'y_train {y_train}')
X_dev, y_dev = get_features_and_labels(dev_set)
X_test, y_test = get_features_and_labels(test_set)

# Scale features prior to polynomial feature generation
scaler = StandardScaler()
x_train = scaler.fit_transform(X_train)
x_test = scaler.fit_transform(X_test)

# Generate polynomial features
x_train_poly = generate_polynomial_features(x_train, polynomial_degree)
x_test_poly = generate_polynomial_features(x_test, polynomial_degree)

# Initialize theta
theta = np.zeros(x_train_poly.shape[1])

# Train the model
print('Training custom model...')
while not converged and k < iterations:
  prediction = perform_logistic_regression(theta, x_train_poly)
  previous_cost = cost
  cost = compute_cost(prediction, y_train, len(y_train))

  if k == 0:
    lowest_cost = cost
  cost_change = cost - previous_cost

  # print(f'cost change: {cost_change}')
  if cost_change > 0:
    pass
```

```
      # can change params here to get cost to go down
      # but I'm out of time to write the code
      # print('Cost going up...')
      # print(f'{previous_cost} changed to {cost}')
    else:
      # print('Cost going down...')
      # print(f'{previous_cost} changed to {cost}')
      if abs(cost_change) < 0.001 and cost_change != 0:
        print('Convergence (hopefully)! Getting results...')
        converged = True
        break

    theta = update_theta(theta, x_train_poly, y_train, prediction, learning_rate,
len(y_train))
    best_theta = theta if k == 0 else best_theta

    # print(f'UPDATED THETA SHAPE: {theta.shape}')

    # Track everything
    cost_history.append(cost)
    if cost < lowest_cost:
      lowest_cost = cost
      # print(f'theta  prior to swap: {theta}')
      best_theta = theta
      # print(f'best_theta : {best_theta}')
    k += 1
    iteration_count.append(k)

  print(f'Lowest Cost of {lowest_cost} was achieved for training.')

  # DEBUGGING
  # plot_cost(k, iteration_count, cost_history, 'Training')
  # print('Before test set, what are these values?')
  # print(f'best theta shape: {best_theta.shape}')
  # print(f'best theta: {best_theta}')
  # print()
  # print(f'x_test_poly shape: {x_test_poly.shape}')

  print('Running test set...')
  # print(f'best theta shape: {best_theta.shape}, type {type(best_theta)}, len
{len(best_theta)}')
  # print(f'x_test shape: {x_test_poly.shape}, type {type(x_test_poly)}, len
{len(x_test_poly)}')
```

```
    test_predictions = perform_logistic_regression(best_theta, x_test_poly).reshape(-1, 1)


    # DEBUGGING
    # print('test_predictions:')
    # print(test_predictions)
    # print()
    # print('labels:')
    # print(y_test)
    # print()
    # print(f'shape of test_pred {test_predictions.shape}, labels {y_test.shape}')

    # continuous variables gave me SO MANY ISSUES
    binary_predictions = (test_predictions >= 0.5).astype(int)
    accuracy = np.mean(binary_predictions == y_test)
    auc = roc_auc_score(y_test, test_predictions)
    print(f'Custom model accuracy: {accuracy}\nCustom model AUC: {auc}')
```

The metrics calculated at the end could be moved into a helper function.

Output of Custom Implementation

The output of the custom implementation hints at a surprisingly accurate model. This could be a result of overfitting, the fact the data is so thoroughly cleaned prior to my obtaining it, or simply a stroke of luck. *Note* sometimes there is a RuntimeWarning because of overflow in prediction = 1 / (1 + np.exp(-z)). I never got a chance to look into solving this issue.

```
Beginning custom regression implementation...
No Data Leaking.
Training custom model...
Lowest Cost of 236.36318857094074 was achieved for training.
Running test set...
Custom model accuracy: 0.9035087719298246
Custom model AUC: 0.985803324099723

<ipython-input-8-ffaac9b20e54>:3: RuntimeWarning: overflow encountered in exp
  prediction = 1 / (1 + np.exp(-z))
```

Running sklearn Implementation

Here we simply make a series of calls to various sklearn modules/libraries. Start by calling `get_logistic_regression` with the first argument = False. I ran into a lot of issues with train_test_split() not returning the expected y values. Additionally, I was running into errors with

StandardScaler() when scaling the values from train_test_split() even though my custom implementation worked fine. I was unable to figure out what was causing this error before needing to complete the assignment, and so switched back to focusing on the custom implementation, as at least that had (seemingly) correct outputs.

```
def sklearn_logistic_regression(data, iterations, penalties, reg_params,
polynomial_degrees):
  X, y = data[0], data[1]

  # print(f'shape X {X.shape}')
  # print(f'shape y {y.shape}')

  # Get splits a standardize
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
train_size=0.6, random_state=42)

  # print(f'X_train: {X_train}')
  # print(f'y_train: {y_train}')
  # print(f'X_test: {X_test}')
  # print(f'y_test: {y_test}')

  # I'm not sure why this needed to happen here, but for some reason
  # it was throwing an error for it being 1D arrays rather than 2D
  X_train = X_train[:, np.newaxis] if len(X_train.shape) == 1 else X_train
  X_test = X_test[:, np.newaxis] if len(X_test.shape) == 1 else X_test

  scaler = StandardScaler()
  x_train = scaler.fit_transform(X_train)
  x_test = scaler.fit_transform(X_test)

  # No continuous variables ? why is this different from custom?
  threshold = 0.5
  y_train_binary = (y_train > threshold).astype(int)
  y_test_binary = (y_test > threshold).astype(int)

  # DEBUGGING
  # print(f'type y_train: {type(y_train)}')
  # print(f'np.unique(y_train): {np.unique(y_train)}')
  # print(f'type y_test: {type(y_test)}')
  # print(f'np.unique(y_test): {np.unique(y_test)}')

  # Get model
  model = LogisticRegression()
```

```
# Fit model
model.fit(x_train, y_train_binary)

# Predict test values
predictions = model.predict(x_test)
binary_predictions = (predictions > threshold).astype(int)

# Evaluate
accuracy = accuracy_score(y_test_binary, binary_predictions)
auc = roc_auc_score(y_test_binary, binary_predictions)

print(f'sklearn model accuracy: {accuracy}\nauc: {auc}')
```

Note running the sklearn implementation on the same data as the output from custom implementation above yields the following results:

```
Beginning sklearn implementation... sklearn model accuracy: 0.14285714285714285 auc: 0.5
```

Again, this is because the model is incorrectly using values from train_test_split. The y values should all be 0 or 1, but are continuous for some reason. I had planned to try using my own shuffle_and_split() and then passing those values to sklearn's model but did not have time.