Drew Lickman

CSCI 4820-001

Project #3

Due: 10/9/24

A.I. Disclaimer: Work for this assignment was completed with the aid of artificial intelligence tools and comprehensive documentation of the names of, input provided to, and output obtained from, these tools is included as part of my assignment submission.

Overall, I was confused with this project due to vagueness of instructions and having to figure out how to use the sklearn library and more Python functions and data structures. I was able to use Claude 3.5 Sonnet and Cursor to augment my understanding of the project and make my code more efficient.

I started by using Claude 3.5 Sonnet to help explain the project to me, which helped a little.



Please explain this project to me, I really don't understand.
 All I have in my jupyternotebook right now is the CustomLogisticRegression class with empty functions

I understand you're feeling a bit overwhelmed with this project. Let's break it down step by step to make it more manageable. I'm glad you've already started with the CustomLogisticRegression class. We'll build on that.

Here's an overview of what the project is asking you to do:

- 1. Implement a custom logistic regression classifier
- 2. Use a sentiment lexicon to classify movie reviews as positive or negative
- 3. Compare your custom classifier with scikit-learn's LogisticRegression classifier

### Let's go through the main components:

- CustomLogisticRegression class: You've already created this class. Now you need to implement these methods:
  - \_\_init\_\_: Initialize learning rate, number of iterations, weights, and bias
  - sigmoid: Implement the sigmoid function
  - · fit: Train the model using gradient descent
  - · predict : Make predictions on new data

#### 2. Lexicon-based sentiment classification:

- · You'll be given positive and negative word lexicons
- · Each movie review will be converted into a feature vector based on these lexicons
- Each feature will be a boolean (True/False) indicating if a word from the lexicon
  is present in the review

# 3. Classification process:

- Load and preprocess the movie review data
- · Split the data into training and test sets
- · Train and test both your custom classifier and scikit-learn's LogisticRegression
- Run this process 5 times and average the results

### 4. Output:

- For each trial and each classifier, print the confusion matrix and classification report
- Calculate and print the average confusion matrix across all trials for each classifier

#### 5. Experimentation (for testing):

· Try different learning rates and numbers of iterations for your custom classifier

### 6. Analysis (for CSCI 5820 students):

- · Discuss the imbalance between positive and negative words in the lexicon
- · Compare the performance of your custom classifier with scikit-learn's
- · Analyze the impact of different learning rates and iteration numbers

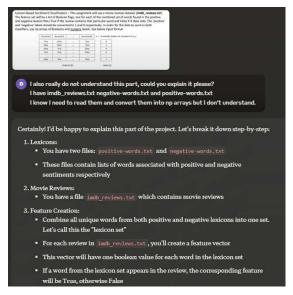
To get started, I suggest focusing on implementing the methods in your CustomLogisticRegression class. Once that's done, we can move on to data preprocessing and the classification process.

Would you like me to help you implement one of the methods in your CustomLogisticRegression class to get you started? I started by writing my custom Logistic Regression class, but got stuck with the fit() and predict() functions, as I was just really confused about what they should do and how. It confirmed that my predict function was correct, and helped significantly with the fit() function.

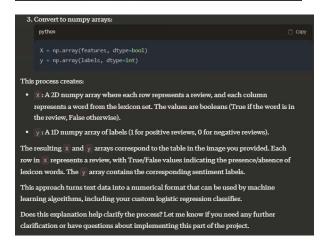




I was also confused with the X features table and Y labels list. After having it explained step-by-step I started feeling a bit better about understanding the project.







I came back to the gradient descent part of the code that Claude 3.5 Sonnet provided so that it could explain it so I could understand it better, which allowed me to adjust it to work a bit better.

```
Can you explain this part of the code?
    # Gradient descent
        # This for loop was written by Claude 3.5 Sonnet
        for _ in range(self.num_iters):
          predictions = self.sigmoid(self.linearTransform(X))
          error = predictions - y
          dw = (1 / len(y)) * np.dot(X.T, error) #X.T is transposed
          db = (1 / len(y)) * np.sum(error)
          self.weights -= self.learning_rate * dw
          self.bias -= self.learning_rate * db
          # Apply batch gradient descent on entire dataset
        # Sets the weights to the correct shape and initializes them to 0
        # Re-initialize weights and bias
Certainly! Let's break down this gradient descent loop step by step:

    This loop runs for a specified number of iterations (self.num iters).

    The underscore __ is used as a throwaway variable since we don't need the loop

         index.
```

```
2. predictions = self.sigmoid(self.linearTransform(X))

    This calculates the predictions for all input features X.

    self.linearTransform(X) computes X * weights + bias.

    The sigmoid function is then applied to convert these values to probabilities.

3. error = predictions - y
    · This calculates the difference between predicted values and actual labels.
4. dw = (1 / len(y)) * np.dot(X.T, error)
    · This computes the gradient for the weights.

    X.T is the transpose of X.

    np.dot(X.T, error) performs matrix multiplication.

    · The result is divided by the number of samples for averaging.
5. db = (1 / len(y)) * np.sum(error)
    · This computes the gradient for the bias.
    · It's the average of all errors.
6. self.weights -= self.learning_rate * dw

    This updates the weights using the computed gradient.

    self.learning_rate determines the step size of each update.

7. self.bias -= self.learning_rate * db
    · This updates the bias in a similar manner.
```

As I was testing my code and using debug tools, <u>I discovered that my predictions were</u> using 1d arrays, rather than 2d arrays after the first iteration. I did some bug tracking and learned that math.pow(math.e), -z)) was only working on single elements, but as suggested by Claude and using np.exp(-z)) it allowed the code to work on entire arrays at once.



We're using NumPy's built-in operations (1 / (...)) which also work element-wise on arrays.
 We've added a shape print statement to help you debug and understand the dimensions of the output.
 This modified sigmoid function will now work correctly with both single values and arrays, making it compatible with batch processing in your logistic regression implementation.
 When you use this function, it will:

 If z is a scalar, return a scalar.
 If z is a 1D array (which it will be in your case), return a 1D array of the same shape.
 If z is a 2D array, it will return a 2D array of the same shape.

 This flexibility ensures that your sigmoid function will work correctly in all parts of your logistic regression algorithm, whether you're processing a single sample or a batch of samples.

Since we used the sklearn library for the logistic regression classifier, I figured it would have a built-in function for randomizing training / test data. I asked Claude about the sklearn functions and discovered model\_selection.train\_test\_split(). I already had most of the trial loop code as skeleton code, I wasn't sure how to use the library functions to run the sklearn

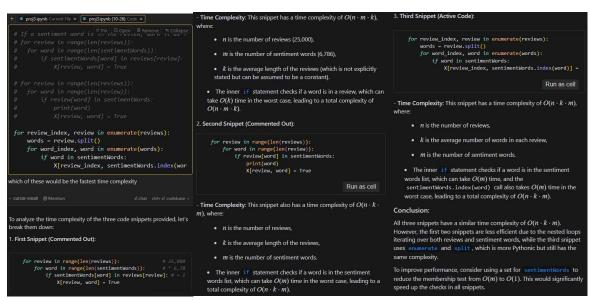
model

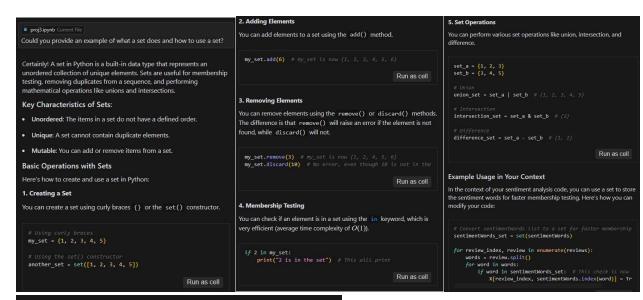
```
D I didn't realize until now that imdb_reviews includes the word "positive" or "negative" at the
      I am now saving each of those into an array trueValues
      Now, how do I use sklearn.model_selection.train_test_split and confusion_matrix and
      classification report?
      for _ in range(5):
         #Split into 80% 20% split of training and test sets
         sklearn.model_selection.train_test_split
         skllr = sklearn.linear_model.LogisticRegression(solver='sag', C=0.001,
       max_iter=50).fit(X, y)
          nylr = CustomLogisticRegression(learning_rate=0.1, num_iters=50)
         predictionValues = skllr.predict(X[:2, :]) #X{array-like, sparse matrix} of shape
         print(predictionValues)
        print(skllr.metrics.confusion_matrix(trueValues, predictionValues))
            nt(skllr.metrics.classification_report(trueValues, predictionValues
       target names=sentimentWords))
         print(mylr.confusion_matrix(trueValues, predictionValue
         print(mylr.classification_report(trueValues, predictionValues,
         # Calculate average confusion matrix
         #xTrain, xTest, yTrain, yTest = train_test_split(x, y, random_state=1)
        mvlr.fit(X, v)
  Great progress on identifying the true labels! Now, let's adjust your code to properly use
  the sklearn functions for splitting the data and evaluating the models. Here's how you can
  modify your code:
```



```
t("Average Confusion Matrix - Sklearn LogisticRegression:"
 print("Average Confusion Matrix - Custom LogisticRegression:")
 print(avg confusion matrix mvlr)
 1. We use train_test_split to split the data into training and test sets for each trial.
 2. We fit both models on the training data and make predictions on the test data.
 3. We use confusion_matrix and classification_report from sklearn.metrics to
    evaluate both models.
 4. We calculate the average confusion matrix across all trials for both models.
 5. The predict method of your custom model should be able to handle multiple
    samples (an entire array) at once. If it can't, we use a list comprehension to predict
    one sample at a time
 6. We use y_test instead of trueValues because y_test contains the true labels for
    the test set we're evaluating on.
  7. We don't need to use sentimentWords as target_names in classification_report.
    Instead, you can use ['negative', 'positive'] if you want to specify names.
Make sure your CustomLogisticRegression class, particularly the predict method, can
handle multiple samples at once. If it can't, you might need to modify it or use the list
comprehension approach shown above.
This setup will give you the required output for each trial and the average confusion
matrices across all trials. You can then experiment with different learning rates and
numbers of iterations for your custom classifier by changing these parameters and
running the trials again.
```

I noticed my feature table assignment loop was taking longer than I expected, usually about 2 to 5 minutes, so I asked my IDE AI "cursor-small" to assist with the time complexity. It explained to me what Python sets are and significantly reduced the time it takes to run this loop to below 15 seconds. I <u>first tried to reduce the time complexity myself</u> but wasn't getting any progress since I haven't used sets in Python before, although they are similar to dictionaries.





# Summary

Using a set can significantly improve the performance of membership tests in your code, especially when dealing with large datasets, as it reduces the time complexity from O(n) to O(1).