

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.



D Please explain this project to me, I really don't understand.
All I have in my jupyter notebook 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:

1. 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.

```
D I have init and sigmoid functions defined. I'm not sure how to define fit(self, X, y) or
predict(self, X)

I especially have no clue what to do for fit, could you explain it instead of giving me the
definition?

def fit(self, X, y):
    # Train the model using gradient descent
    # X is training features, y is labels
    # Sets the weights to the correct shape and initializes them to 0
    # Applies batch gradient descent (entire dataset) in a loop for # iterations specified
    # Re-initialize weights and bias
    print("fit")

Also could you check how correct my predict function is

def predict(self, X):
    # Make predictions on test data
    # Computes W DOT x + b
    output = numpy.dot(self.weights, X) + self.bias
    output = self.sigmoid(output)
    return output
```

I'm glad you've made progress with the `__init__` and `sigmoid` functions. Let's focus on understanding the `fit` method and then review your `predict` method.

Explanation of the `fit` method:

The `fit` method is where the actual learning happens in your logistic regression model. Here's a step-by-step explanation of what it should do:

1. Initialize weights and bias:
 - The weights should be a 1D numpy array with length equal to the number of features in X.
 - The bias should be a single scalar value.
2. Gradient Descent Loop:
 - For the specified number of iterations: a. Calculate the predictions using the current weights and bias. b. Calculate the error (difference between predictions and actual labels). c. Calculate the gradients for weights and bias. d. Update the weights and bias using the gradients and learning rate.

The key idea is to iteratively adjust the weights and bias to minimize the difference between the predicted values and the actual labels.

Here's a conceptual outline of what the `fit` method might look like:

```
python Copy

def fit(self, X, y):
    # Initialize weights and bias
    num_features = X.shape[1]
    self.weights = np.zeros(num_features)
    self.bias = 0

    # Gradient descent loop
    for _ in range(self.num_iters):
        # Calculate predictions
        predictions = self.sigmoid(np.dot(X, self.weights) + self.bias)

        # Calculate error
        error = predictions - y

        # Calculate gradients
        dw = (1 / len(y)) * np.dot(X.T, error)
        db = (1 / len(y)) * np.sum(error)

        # Update weights and bias
        self.weights -= self.learning_rate * dw
        self.bias -= self.learning_rate * db
```

This is a simplified version, and you might want to add checks for convergence or print out the cost at each iteration to monitor progress.

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.

Lexicon-Based Sentiment Classification - This assignment will use a movie reviews dataset (`imdb_reviews.txt`). The feature set will be a list of boolean flags, one for each of the combined set of words found in the positive and negative lexicons from `positive-words.txt` and `negative-words.txt`. The 'positive' and 'negative' labels should be converted to 1 and 0 respectively, in order for the data to work in both classifiers, use as arrays of booleans and `numpy` labels. See below table format.

Review ID	Review Text	Sentiment
1	This movie is really good.	1
2	I hate this movie.	0
3	It was a waste of time.	0
4	Great acting and story.	1
5	Not worth the money.	0

D I also really do not understand this part, could you explain it please?
I have `imdb_reviews.txt` `negative-words.txt` and `positive-words.txt`
I know I need to read them and convert them into np arrays but I don't understand.

Certainly! I'd be happy to explain this part of the project. Let's break it down step-by-step:

- Lexicons:**
 - You have two files: `positive-words.txt` and `negative-words.txt`.
 - These files contain lists of words associated with positive and negative sentiments respectively.
- Movie Reviews:**
 - You have a file `imdb_reviews.txt` which contains movie reviews.
- Feature Creation:**
 - Combine all unique words from both positive and negative lexicons into one set. Let's call this the "lexicon set".
 - For each review in `imdb_reviews.txt`, you'll create a feature vector.
 - This vector will have one boolean value for each word in the lexicon set.
 - If a word from the lexicon set appears in the review, the corresponding feature will be True, otherwise False.

4. Label Creation:

- Each review should have a sentiment label (positive or negative).
- Convert positive sentiment to 1 and negative sentiment to 0.

Here's a conceptual approach to implement this:

1. Read lexicons:

```
python
with open('positive-words.txt', 'r') as f:
    positive_words = set(f.read().splitlines())
with open('negative-words.txt', 'r') as f:
    negative_words = set(f.read().splitlines())
lexicon_set = positive_words.union(negative_words)
```

2. Process reviews and create features:

```
python
features = []
labels = []
with open('imdb_reviews.txt', 'r') as f:
    for line in f:
        review, sentiment = line.strip().split('\t') # Assuming tab-separated format
        # Create feature vector
        feature_vector = [word in review.lower().split() for word in lexicon_set]
        features.append(feature_vector)
        # Create label (1 for positive, 0 for negative)
        labels.append(1 if sentiment == 'positive' else 0)
```

3. Convert to numpy arrays:

```
python
X = np.array(features, dtype=bool)
y = np.array(labels, dtype=int)
```

This process creates:

- `X`: A 2D numpy array where each row represents a review, and each column represents a word from the lexicon set. The values are booleans (True if the word is in the review, False otherwise).
- `y`: A 1D numpy array of labels (1 for positive reviews, 0 for negative reviews).

The resulting `X` and `y` arrays correspond to the table in the image you provided. Each row in `X` represents a review, with True/False values indicating the presence/absence of lexicon words. The `y` array contains the corresponding sentiment labels.

This approach turns text data into a numerical format that can be used by machine learning algorithms, including your custom logistic regression classifier.

Does this explanation help clarify the process? Let me know if you need any further clarification or have questions about implementing this part of 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.

D 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:

1. `for _ in range(self.num_iters):`
 - 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, I discovered that my predictions were 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.

D Then I need to adjust my sigmoid function to be able to handle the array of sample Z values

```
def sigmoid(self, z):
    print("apply sigmoid to z")
    output = 1/(1+math.pow((math.e), -z))
    print("Sigmoid:", output)
    return output
```

You're absolutely right. Your current sigmoid function is designed to work with a single scalar value, but we need it to handle an array of values. Let's modify it to use NumPy operations, which will allow it to work efficiently with arrays:

```
python Copy

import numpy as np

def sigmoid(self, z):
    print("Applying sigmoid to z")
    output = 1 / (1 + np.exp(-z))
    print(f"Sigmoid output shape: {output.shape}")
    return output
```

Here's what changed and why:

1. We replaced `math.pow((math.e), -z)` with `np.exp(-z)`.
 - `np.exp()` is the NumPy exponential function.
 - It can operate element-wise on arrays, which is exactly what we need.

2. We're using NumPy's built-in operations (`1 / (...)`) which also work element-wise on arrays.

3. 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.

I didn't realize until now that `imdb_reviews` includes the word "positive" or "negative" at the end. 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`? This is what I have currently

```
for _ in range(5):
    # Shuffle input data
    # 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)
    mylr = CustomLogisticRegression(learning_rate=0.1, num_iters=500)
    predictionValues = skllr.predict(X[-2,:]) #X[array-like, sparse matrix] of shape
(n_samples, n_features)
    print(predictionValues)
    print(skllr.score(X, y))

    print(skllr.metrics.confusion_matrix(trueValues, predictionValues))
    print(skllr.metrics.classification_report(trueValues, predictionValues,
target_names=sentimentWords))

    print(mylr.confusion_matrix(trueValues, predictionValues))
    print(mylr.classification_report(trueValues, predictionValues,
target_names=sentimentWords))

    # Calculate average confusion matrix
    #xTrain, xTest, yTrain, yTest = train_test_split(x, y, random_state=1)
    mylr.fit(X, y)
```

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:

```
python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report
import numpy as np

# Assuming X and y are your features and labels respectively
# and trueValues is your array of actual labels

# Initialize variables to store average confusion matrices
avg_confusion_matrix_skllr = np.zeros((2, 2))
avg_confusion_matrix_mylr = np.zeros((2, 2))

for trial in range(5):
    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

    # Sklearn LogisticRegression
    skllr = LogisticRegression(solver='sag', C=0.001, max_iter=500)
    skllr.fit(X_train, y_train)
    skllr_predictions = skllr.predict(X_test)

    # Your custom LogisticRegression
    mylr = CustomLogisticRegression(learning_rate=0.1, num_iters=500)
    mylr.fit(X_train, y_train)
    mylr_predictions = np.array([mylr.predict(x) for x in X_test])

    # Evaluate sklearn model
    print(f"Trial {trial + 1} - Sklearn LogisticRegression:")
    print(confusion_matrix(y_test, skllr_predictions))
    print(classification_report(y_test, skllr_predictions))

    # Evaluate custom model
    print(f"Trial {trial + 1} - Custom LogisticRegression:")
    print(confusion_matrix(y_test, mylr_predictions))
    print(classification_report(y_test, mylr_predictions))

    # Update average confusion matrices
    avg_confusion_matrix_skllr += confusion_matrix(y_test, skllr_predictions)
    avg_confusion_matrix_mylr += confusion_matrix(y_test, mylr_predictions)
```

```
print("Average Confusion Matrix - Sklearn LogisticRegression:")
print(avg_confusion_matrix_skllr)

print("Average Confusion Matrix - Custom LogisticRegression:")
print(avg_confusion_matrix_mylr)
```

Key changes and explanations:

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 first tried to reduce the time complexity myself but wasn’t getting any progress since I haven’t used sets in Python before, although they are similar to dictionaries.

```
# If a sentiment word is in the review, mark it as True in the X feature table
for review in range(len(reviews)): # 25,000
    for word in range(len(sentimentWords)): # * 6,786
        if sentimentWords[word] in reviews[review]: # = 169,650,000 loops
            X[review, word] = True # Takes 2-5 minutes

# cursor-small improved my time complexity from  $O(n*m*k)$  to  $O(n*k)$ 
# Convert sentimentWords list to a set for faster membership testing
sentimentWords_set = set(sentimentWords)
for review_index, review in enumerate(reviews):
    words = review.split()
    for word in words:
        if word in sentimentWords_set: # This check is now  $O(1)$  by using hashable set
            X[review_index, sentimentWords.index(word)] = True # Reduces time to <15 seconds
```

```
# If a sentiment word is in the review, mark it as True in the X feature table
for review in range(len(reviews)):
    for word in range(len(sentimentWords)):
        if sentimentWords[word] in reviews[review]:
            X[review, word] = True

# for review in range(len(reviews)):
#     for word in range(len(review)):
#         if review[word] in sentimentWords:
#             print(word)
#             X[review, word] = True

for review_index, review in enumerate(reviews):
    words = review.split()
    for word_index, word in enumerate(words):
        if word in sentimentWords:
            X[review_index, sentimentWords.index(word)] = True
```

which of these would be the fastest time complexity

cursor-small @ Mention chat ctrl+@ codebase

To analyze the time complexity of the three code snippets provided, let's break them down:

1. First Snippet (Commented Out):

```
for review in range(len(reviews)): # 25,000
    for word in range(len(sentimentWords)): # * 6,786
        if sentimentWords[word] in reviews[review]: # = 1
            X[review, word] = True
```

- Time Complexity: This snippet has a time complexity of $O(n \cdot m \cdot k)$, where:

- n is the number of reviews (25,000),
- m is the number of sentiment words (6,786),
- k is the average length of the reviews (which is not explicitly stated but can be assumed to be a constant).

2. Second Snippet (Commented Out):

```
for review in range(len(reviews)):
    for word in range(len(review)):
        if review[word] in sentimentWords:
            print(word)
            X[review, word] = True
```

- Time Complexity: This snippet also has a time complexity of $O(n \cdot k \cdot m)$, where:

- n is the number of reviews,
- k is the average length of the reviews,
- m is the number of sentiment words.

3. Third Snippet (Active Code):

```
for review_index, review in enumerate(reviews):
    words = review.split()
    for word_index, word in enumerate(words):
        if word in sentimentWords:
            X[review_index, sentimentWords.index(word)] =
```

- Time Complexity: This snippet has a time complexity of $O(n \cdot k \cdot m)$, where:

- n is the number of reviews,
- k is the average number of words in each review,
- m is the number of sentiment words.

The inner `if` statement checks if a word is in the sentiment words list, which can take $O(m)$ time, and the `sentimentWords.index(word)` call also takes $O(m)$ time in the worst case, leading to a total complexity of $O(n \cdot k \cdot m)$.

Conclusion:

All three snippets have a similar time complexity of $O(n \cdot k \cdot m)$. However, the first two snippets are less efficient due to the nested loops iterating over both reviews and sentiment words, while the third snippet uses `enumerate` and `split`, which is more Pythonic but still has the same complexity.

To improve performance, consider using a set for `sentimentWords` to reduce the membership test from $O(m)$ to $O(1)$. This would significantly speed up the checks in all snippets.

proj3.ipynbCurrent File

Could you provide an example of what a set does and how to use a set?

Certainly! A set in Python is a built-in data type that represents an unordered collection of unique elements. Sets are useful for membership testing, removing duplicates from a sequence, and performing mathematical operations like unions and intersections.

Key Characteristics of Sets:

- **Unordered:** The items in a set do not have a defined order.
- **Unique:** A set cannot contain duplicate elements.
- **Mutable:** You can add or remove items from a set.

Basic Operations with Sets

Here's how to create and use a set in Python:

1. Creating a Set

You can create a set using curly braces `{}` or the `set()` constructor.

```
# Using curly braces
my_set = {1, 2, 3, 4, 5}

# Using the set() constructor
another_set = set([1, 2, 3, 4, 5])
```

Run as cell

2. Adding Elements

You can add elements to a set using the `add()` method.

```
my_set.add(6) # my_set is now {1, 2, 3, 4, 5, 6}
```

Run as cell

3. Removing Elements

You can remove elements using the `remove()` or `discard()` methods. The difference is that `remove()` will raise an error if the element is not found, while `discard()` will not.

```
my_set.remove(3) # my_set is now {1, 2, 4, 5, 6}
my_set.discard(10) # No error, even though 10 is not in the
```

Run as cell

4. Membership Testing

You can check if an element is in a set using the `in` keyword, which is very efficient (average time complexity of $O(1)$).

```
if 2 in my_set:
    print("2 is in the set") # This will print
```

Run as cell

5. Set Operations

You can perform various set operations like union, intersection, and difference.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union
union_set = set_a | set_b # {1, 2, 3, 4, 5}

# Intersection
intersection_set = set_a & set_b # {3}

# Difference
difference_set = set_a - set_b # {1, 2}
```

Run as cell

Example Usage in Your Context

In the context of your sentiment analysis code, you can use a set to store the sentiment words for faster membership testing. Here's how you can modify your code:

```
# Convert sentimentWords List to a set for faster membership
sentimentWords_set = set(sentimentWords)

for review_index, review in enumerate(reviews):
    words = review.split()
    for word in words:
        if word in sentimentWords_set: # This check is now
            x[review_index, sentimentWords.index(word)] = Tr
```

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 $O(m)$ to $O(1)$.