Sentiment Analysis from Scratch Using Logistic Regression

Name: Harshkumar Rana
Roll Number: 22BCP091

Project Overview

This project implements a complete sentiment analysis pipeline from scratch using Python. The objective is to classify Amazon product reviews as *positive* or *negative* based on their textual content.

The workflow includes:

- Loading and preprocessing the Amazon review dataset
- Constructing a vocabulary and generating TF-IDF-based text representations
- Implementing Logistic Regression training using batch gradient descent with L2 regularization
- Evaluating model performance using metrics such as accuracy, precision, recall, and F1-score
- Displaying a confusion matrix and identifying the most influential words contributing to predictions

All computations, including model training and evaluation, are implemented manually without using external machine learning or visualization libraries like Matplotlib. The results are displayed directly in text format for clarity and compliance with project requirements.

Dataset

The dataset contains Amazon product reviews with numerical ratings. Reviews rated **4 or 5** are considered *positive*, while those rated **1 or 2** are considered *negative*. Neutral reviews (rating 3) are excluded to maintain a balanced binary classification. A subset of data is used to ensure efficient training on available hardware.

Model and Training

A custom Logistic Regression model was implemented entirely from first principles. The model includes manual implementations of the sigmoid activation function, dot product computation, binary cross-entropy loss, and parameter updates using batch gradient descent.
Regularization is applied to prevent overfitting, and the model is trained over multiple epochs for stable convergence.

Goals

- Develop a clear understanding of text feature extraction techniques such as **TF-IDF**
- Gain hands-on experience implementing **machine learning algorithms from scratch**
- Learn to evaluate classification performance using standard metrics
- Focus on algorithmic understanding rather than pre-built libraries or external visualizations

---

In [4]:
```python
import csv
import math
import random
import collections
from typing import List, Tuple, Dict

# ---------- USER CONFIG ----------
DATAPATH = "../data/Reviews.csv"  # change if your CSV is elsewhere
MAXSAMPLESPERCLASS = 50000  # per class (reduce if memory/time is tight)
VOCABSIZE = 5000  # top-k words to keep
TESTRATIO = 0.2  # fraction for test set
RANDOMSEED = 42
LEARNINGRATE = 0.5
EPOCHS = 30
BATCHSIZE = 128
L2REG = 1e-4
# ---------------------------------
random.seed(RANDOMSEED)
```

In [5]:
```python
def load_reviews(path: str):
    data = []
```

```
    with open(path, newline="", encoding="utf-8") as f:
        reader = csv.DictReader(f)
        for i, row in enumerate(reader):
            if "Text" not in row or "Score" not in row:
                continue
            text = row["Text"].strip()
            try:
                score = float(row["Score"])
            except:
                continue
            if score >= 4:
                data.append((text, 1))
            elif score <= 2:
                data.append((text, 0))
    return data


print("Loading dataset (this may take a few seconds)...")
all_data = load_reviews(DATAPATH)
print("Total binary-labeled samples in file:", len(all_data))
```

```
Loading dataset (this may take a few seconds)...
Total binary-labeled samples in file: 525814
```

In [6]:
```
def sample_balanced(data: List[Tuple[str, int]], max_per_class: int):
    pos = [t for t in data if t[1] == 1]
    neg = [t for t in data if t[1] == 0]
    random.shuffle(pos)
    random.shuffle(neg)
    pos = pos[:max_per_class]
    neg = neg[:max_per_class]
    sampled = pos + neg
    random.shuffle(sampled)
    return sampled

data = sample_balanced(all_data, MAXSAMPLESPERCLASS)
print(f"After sampling balanced subset: {len(data)} - positives: {sum(1 for _, l in data if l == 1)}, negatives: {sur
```

```
After sampling balanced subset: 100000 - positives: 50000, negatives: 50000
```

In [7]:
```
STOPWORDS = set(
    [
        "the",
```

```
        "and",
        "a",
        "an",
        "is",
        "it",
        "this",
        "that",
        "to",
        "for",
        "of",
        "i",
        "you",
        "was",
        "with",
        "on",
        "in",
        "my",
        "we",
        "they",
        "he",
        "she",
        "but",
        "not",
        "are",
        "as",
        "have",
        "had",
        "be",
        "at",
        "from",
        "so",
        "if",
        "or",
        "its",
    ]
)


def preprocess_text(text: str) -> list:
    # 1. Convert to lowercase
    text = text.lower()
```

```
    # 2. Remove punctuation (anything not a-z, A-Z, 0-9, or whitespace)
    allowed_chars = "abcdefghijklmnopqrstuvwxyz0123456789 "
    cleaned = []
    for ch in text:
        if ch in allowed_chars:
            cleaned.append(ch)
        else:
            cleaned.append(" ")
    cleaned_text = "".join(cleaned)

    # 3. Replace multiple spaces with one space (manual multi-space removal)
    final_text = []
    prev_char = None
    for ch in cleaned_text:
        if ch == " " and prev_char == " ":
            continue
        final_text.append(ch)
        prev_char = ch
    final_text_str = "".join(final_text).strip()

    # 4. Split into tokens
    tokens = final_text_str.split(" ")

    # 5. Filter out stopwords, tokens of length <= 1, and tokens that are all digits
    filtered_tokens = []
    for token in tokens:
        if len(token) > 1 and token not in STOPWORDS and not token.isdigit():
            filtered_tokens.append(token)

    return filtered_tokens


# Quick test
print(
    preprocess_text("I loved this product! It's amazing, would buy again. 1010")
)
```

```
['loved', 'product', 'amazing', 'would', 'buy', 'again']
```

In [8]:

```
def build_vocab(doc_tokens: List[List[str]], vocab_size: int):
    df = collections.Counter()
    for tokens in doc_tokens:
```

```
        unique = set(tokens)
        for t in unique:
            df[t] += 1
    most_common = df.most_common(vocab_size)
    vocab = {w: idx for idx, (w, _) in enumerate(most_common)}
    doc_freqs = [count for _, count in most_common]
    return vocab, doc_freqs


doc_tokens = [preprocess_text(text) for text, _ in data]
vocab, doc_freqs = build_vocab(doc_tokens, VOCABSIZE)
print("Vocab size:", len(vocab))
print("Sample vocab items:", list(vocab.items())[:10])
```

```
Vocab size: 5000
Sample vocab items: [('like', 0), ('br', 1), ('these', 2), ('good', 3), ('taste', 4), ('just', 5), ('one', 6), ('prod
uct', 7), ('very', 8), ('all', 9)]
```

In [9]:
```
# Visualize class distribution (positive vs negative samples)
def print_class_distribution(data):
    pos = 0
    neg = 0
    for _, label in data:
        if label == 1:
            pos += 1
        else:
            neg += 1

    total = pos + neg
    print("Class Distribution:")
    print("Positive: ", pos, "({:.2f}%)".format((pos / total) * 100))
    print("Negative: ", neg, "({:.2f}%)".format((neg / total) * 100))


# Simple text bar chart for token frequencies
def print_simple_bar_chart_no_lib(values, labels):
    max_val = 0
    for v in values:
        if v > max_val:
            max_val = v

    max_width = 50
```

```
    for i in range(len(values)):
        val = values[i]
        label = labels[i]
        bar_length = (val * max_width) // max_val
        label_str = label + (" " * (15 - len(label)))
        bar_str = ""
        for _ in range(bar_length):
            bar_str += "#"
        print(label_str + "| " + bar_str + " (" + str(val) + ")")


# Example usage to show top 10 tokens frequency in your vocab
def visualize_top_tokens_frequency(vocab, doc_tokens):
    token_counts = {}
    for tokens in doc_tokens:
        for t in tokens:
            token_counts[t] = token_counts.get(t, 0) + 1

    # Sort tokens by frequency
    sorted_tokens = sorted(token_counts.items(), key=lambda x: x[1], reverse=True)
    top_tokens = sorted_tokens[:10]

    labels = [t[0] for t in top_tokens]
    values = [t[1] for t in top_tokens]

    print("\nTop 10 Tokens Frequency:")
    print_simple_bar_chart_no_lib(values, labels)


# Call functions with your data variables
print_class_distribution(data)
visualize_top_tokens_frequency(vocab, doc_tokens)
```

```
Class Distribution:
Positive:  50000 (50.00%)
Negative:  50000 (50.00%)

Top 10 Tokens Frequency:
br             | ################################################## (117619)
like           | ##################### (47937)
these          | ################# (39679)
them           | ############### (35896)
product        | ############## (34377)
taste          | ############# (32932)
one            | ############# (32289)
just           | ############# (31539)
good           | ############# (31119)
can            | ############ (29914)
```

In [10]:

```python
def compute_idf(doc_freqs: List[int], num_docs: int):
    idf = []
    for df in doc_freqs:
        idf.append(math.log(1 + num_docs / (1 + df)))
    return idf


IDF = compute_idf(doc_freqs, len(doc_tokens))


def vectorize_tfidf(tokens: List[str], vocab: Dict[str, int], idf: List[float]):
    tf = [0.0] * len(vocab)
    for t in tokens:
        if t in vocab:
            tf[vocab[t]] += 1.0
    total = sum(tf)
    if total == 0:
        return tf
    for i in range(len(tf)):
        tf[i] = tf[i] / total * idf[i]
    return tf


X = [vectorize_tfidf(tokens, vocab, IDF) for tokens in doc_tokens]
y = [label for _, label in data]
print("Built feature matrix N =", len(X), "features per sample =", len(X[0]))
```

```
Built feature matrix N = 100000 features per sample = 5000
```

In [11]:
```python
def train_test_split(X, y, test_ratio=TESTRATIO):
    idxs = list(range(len(X)))
    random.shuffle(idxs)
    split = int(len(idxs) * (1 - test_ratio))
    train_idx = idxs[:split]
    test_idx = idxs[split:]
    X_train = [X[i] for i in train_idx]
    y_train = [y[i] for i in train_idx]
    X_test = [X[i] for i in test_idx]
    y_test = [y[i] for i in test_idx]
    return X_train, X_test, y_train, y_test


X_train, X_test, y_train, y_test = train_test_split(X, y)
print("Train =", len(X_train), "Test =", len(X_test))
```

```
Train = 80000 Test = 20000
```

In [12]:
```python
def dot(a, b):
    return sum(ai * bi for ai, bi in zip(a, b))


def sigmoid(z):
    if z >= 0:
        ez = math.exp(-z)
        return 1.0 / (1.0 + ez)
    else:
        ez = math.exp(z)
        return ez / (1.0 + ez)


class LogisticRegressionScratch:
    def __init__(self, nfeatures, lr=0.1, l2=0.0):
        self.w = [random.uniform(-0.01, 0.01) for _ in range(nfeatures)]
        self.b = 0.0
        self.lr = lr
        self.l2 = l2

    def predict_proba_single(self, x):
        z = dot(self.w, x) + self.b
```

```
        return sigmoid(z)

    def predict_single(self, x, threshold=0.5):
        return 1 if self.predict_proba_single(x) >= threshold else 0

    def compute_loss(self, X, y):
        total = 0.0
        for xi, yi in zip(X, y):
            p = max(1e-12, min(1 - 1e-12, self.predict_proba_single(xi)))
            total += -yi * math.log(p) - (1 - yi) * math.log(1 - p)
        avg = total / len(X)
        l2_term = 0.5 * self.l2 * sum(w * w for w in self.w)
        return avg + l2_term

    def fit(self, X, y, epochs=10, batch_size=32, verbose=True):
        n = len(X)
        loss_history = []
        for epoch in range(epochs):
            idxs = list(range(n))
            random.shuffle(idxs)
            for start in range(0, n, batch_size):
                end = min(start + batch_size, n)
                batch_idxs = idxs[start:end]
                grad_w = [0.0] * len(self.w)
                grad_b = 0.0
                for i in batch_idxs:
                    xi = X[i]
                    yi = y[i]
                    pi = self.predict_proba_single(xi)
                    err = pi - yi
                    for j in range(len(self.w)):
                        grad_w[j] += err * xi[j]
                    grad_b += err
                batch_len = len(batch_idxs)
                for j in range(len(self.w)):
                    grad = grad_w[j] / batch_len + self.l2 * self.w[j]
                    self.w[j] -= self.lr * grad
                self.b -= self.lr * grad_b / batch_len
            loss = self.compute_loss(X, y)
            loss_history.append(loss)
            if verbose:
                print(f"Epoch {epoch + 1}/{epochs} - loss {loss:.4f}")
```

```
        return loss_history


model = LogisticRegressionScratch(nfeatures=len(vocab), lr=LEARNINGRATE, l2=L2REG)
losshist = model.fit(X_train, y_train, epochs=EPOCHS, batch_size=BATCHSIZE)
```

```
Epoch 1/30 - loss 0.6431
Epoch 2/30 - loss 0.6062
Epoch 3/30 - loss 0.5788
Epoch 4/30 - loss 0.5579
Epoch 5/30 - loss 0.5418
Epoch 6/30 - loss 0.5294
Epoch 7/30 - loss 0.5190
Epoch 8/30 - loss 0.5110
Epoch 9/30 - loss 0.5045
Epoch 10/30 - loss 0.4989
Epoch 11/30 - loss 0.4944
Epoch 12/30 - loss 0.4907
Epoch 13/30 - loss 0.4873
Epoch 14/30 - loss 0.4846
Epoch 15/30 - loss 0.4824
Epoch 16/30 - loss 0.4810
Epoch 17/30 - loss 0.4794
Epoch 18/30 - loss 0.4774
Epoch 19/30 - loss 0.4762
Epoch 20/30 - loss 0.4751
Epoch 21/30 - loss 0.4744
Epoch 22/30 - loss 0.4743
Epoch 23/30 - loss 0.4728
Epoch 24/30 - loss 0.4723
Epoch 25/30 - loss 0.4717
Epoch 26/30 - loss 0.4712
Epoch 27/30 - loss 0.4708
Epoch 28/30 - loss 0.4704
Epoch 29/30 - loss 0.4701
Epoch 30/30 - loss 0.4698
```

In [13]:

```
def ascii_loss_curve(loss_history):
    max_len = 40  # Maximum width of the plotted bar
    if not loss_history or len(loss_history) == 0:
        print("No loss history available to plot.")
        return
```

```
    max_loss = max(loss_history)
    print("Loss Curve Over Epochs:")
    for i, loss in enumerate(loss_history):
        bar_len = int((loss / max_loss) * max_len) if max_loss > 0 else 0
        bar = "" * bar_len
        print(f"Epoch {i + 1:3}: {bar:<40} {loss:.4f}")


ascii_loss_curve(losshist)
```

```
Loss Curve Over Epochs:
Epoch   1:  0.6431
Epoch   2:  0.6062
Epoch   3:  0.5788
Epoch   4:  0.5579
Epoch   5:  0.5418
Epoch   6:  0.5294
Epoch   7:  0.5190
Epoch   8:  0.5110
Epoch   9:  0.5045
Epoch  10:  0.4989
Epoch  11:  0.4944
Epoch  12:  0.4907
Epoch  13:  0.4873
Epoch  14:  0.4846
Epoch  15:  0.4824
Epoch  16:  0.4810
Epoch  17:  0.4794
Epoch  18:  0.4774
Epoch  19:  0.4762
Epoch  20:  0.4751
Epoch  21:  0.4744
Epoch  22:  0.4743
Epoch  23:  0.4728
Epoch  24:  0.4723
Epoch  25:  0.4717
Epoch  26:  0.4712
Epoch  27:  0.4708
Epoch  28:  0.4704
Epoch  29:  0.4701
Epoch  30:  0.4698
```

In [19]:
```python
def printmetrics(metrics, datasetname):
    print(f"\n---- {datasetname} Metrics ----")
    print(f"Accuracy:  {metrics['accuracy']:.4f}")
    print(f"Precision: {metrics['precision']:.4f}")
    print(f"Recall:    {metrics['recall']:.4f}")
    print(f"F1-score:  {metrics['f1']:.4f}")
    print(
        f"TP: {metrics['TP']}, FP: {metrics['FP']}, FN: {metrics['FN']}, TN: {metrics['TN']}"
    )
    print("---------------------------\n")
```

In [20]:
```python
def calculate_metrics(y_true, y_pred):
    TP = sum((yt == 1 and yp == 1) for yt, yp in zip(y_true, y_pred))
    TN = sum((yt == 0 and yp == 0) for yt, yp in zip(y_true, y_pred))
    FP = sum((yt == 0 and yp == 1) for yt, yp in zip(y_true, y_pred))
    FN = sum((yt == 1 and yp == 0) for yt, yp in zip(y_true, y_pred))

    accuracy = (TP + TN) / len(y_true)
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0.0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0.0
    f1_score = (
        (2 * precision * recall) / (precision + recall)
        if (precision + recall) > 0
        else 0.0
    )

    return {
        "accuracy": accuracy,
        "precision": precision,
        "recall": recall,
        "f1": f1_score,
        "TP": TP,
        "FP": FP,
        "FN": FN,
        "TN": TN,
    }


# Use your model instance to predict each sample's label:
train_predictions = [model.predict_single(x) for x in X_train]
test_predictions = [model.predict_single(x) for x in X_test]
```

```
# Calculate metrics for train and test
trainmetrics = calculate_metrics(y_train, train_predictions)
testmetrics = calculate_metrics(y_test, test_predictions)

# Now use your existing printmetrics(metrics, datasetname) function to display:
printmetrics(trainmetrics, "Train Dataset")
printmetrics(testmetrics, "Test Dataset")
```

```
---- Train Dataset Metrics ----
Accuracy:  0.8725
Precision: 0.8831
Recall:    0.8590
F1-score:  0.8709
TP: 34403, FP: 4554, FN: 5645, TN: 35398
----------------------------


---- Test Dataset Metrics ----
Accuracy:  0.8647
Precision: 0.8712
Recall:    0.8543
F1-score:  0.8627
TP: 8502, FP: 1257, FN: 1450, TN: 8791
----------------------------
```

In [23]:
```
def top_features_by_weight(w: List[float], vocab: Dict[str, int], topk=25):
    inv = {idx: word for word, idx in vocab.items()}
    pairs = [(inv[i], wi) for i, wi in enumerate(w) if i in inv]
    pairs.sort(key=lambda x: x[1], reverse=True)
    top_pos = pairs[:topk]
    top_neg = pairs[-topk:]
    return top_pos, top_neg

top_pos, top_neg = top_features_by_weight(model.w, vocab, topk=25)
print("Top positive-weight tokens (indicative of positive class):")
for tok, wt in top_pos:
    print(f"{tok: <12} {wt:.4f}")
print("Top negative-weight tokens (indicative of negative class):")
for tok, wt in top_neg:
    print(f"{tok: <12} {wt:.4f}")
```

```
Top positive-weight tokens (indicative of positive class):
great        8.4334
best         5.6148
love         5.4053
delicious    4.8593
good         4.3095
perfect      4.2156
loves        4.0710
excellent    3.4111
favorite     3.2869
nice         3.2408
easy         3.1342
wonderful    3.0620
highly       3.0052
find         2.6271
well         2.4817
tasty        2.3941
use          2.3780
happy        2.3135
always       2.3013
snack        2.2816
little       2.2775
stores       2.2549
day          2.2130
smooth       2.1348
amazing      2.1345
Top negative-weight tokens (indicative of negative class):
off          -2.0142
china        -2.0708
should       -2.1208
even         -2.1355
disappointing -2.1536
maybe        -2.2702
away         -2.3625
unfortunately -2.4255
didn         -2.4535
weak         -2.4718
stale        -2.4748
return       -2.5254
taste        -2.6373
waste        -2.6821
would        -2.7492
```

```
terrible     -2.7567
awful        -2.7631
horrible     -2.8257
did          -2.9202
worst        -2.9332
were         -3.0050
thought      -3.1963
money        -3.2110
bad          -3.5079
disappointed -4.4780
```

In [22]:
```python
def save_model_weights(path: str, model, vocab, idf):
    with open(path, "w", encoding="utf-8") as f:
        f.write("BINARYLOGREGMODELV1\n")
        f.write(f"strlenmodel.w {len(model.w)}\n")
        f.write(f"bias {model.b}\n")
        f.write("weights\n")
        for w in model.w:
            f.write(f"{w}\n")
        f.write("vocab\n")
        for word, idx in vocab.items():
            f.write(f"{word} {idx}\n")
        f.write("idf\n")
        for v in idf:
            f.write(f"{v}\n")


save_model_weights("logreg_model_v1.txt", model, vocab, IDF)
print("Model saved to logreg_model_v1.txt")
```

```
Model saved to logreg_model_v1.txt
```

Conclusion:

In this project, I developed a custom sentiment analysis model from scratch using Logistic Regression to classify Amazon product reviews as positive or negative. The implementation covered every stage of the machine learning pipeline from text preprocessing, tokenization, and feature vectorization to gradient-based optimization and evaluation. Without relying on pre-built ML libraries, the model was able to learn meaningful patterns from review data and achieve consistent accuracy across the dataset. Through this process, I gained a deeper understanding of how Logistic Regression works internally, including the impact of parameters like

learning rate, regularization, and loss convergence. The visualization of token frequencies and class distribution helped interpret dataset balance and common word patterns. Overall, this project strengthened my understanding of core machine learning concepts and demonstrated how custom-built models can effectively solve real-world text classification problems.