Title – Amazon Review Sentiment Classification: Insights and Predictions Using TF-IDF and Machine Learning

By,

Harish K M

Abstract:

This project aims to analyze Amazon product reviews to understand customer sentiment and identify key business insights. Using a dataset containing millions of product reviews labeled as positive or negative, natural language processing (NLP) techniques were applied to transform raw text into structured information. After thorough text preprocessing and exploratory data analysis (EDA), machine learning models such as Logistic Regression, Multinomial Naive Bayes, LightGBM, and XGBoost were trained using TF-IDF (Term Frequency–Inverse Document Frequency) features.

The models were optimized through hyperparameter tuning to maximize F1-Score, ensuring balanced performance between precision and recall. Logistic Regression achieved the best performance with high accuracy and computational efficiency.
This project not only highlights the power of text analytics for understanding customer opinions but also demonstrates how organizations can leverage machine learning to improve customer experience, product design, and brand loyalty. Finally, the system was deployed as an interactive **Streamlit web application** that allows real-time review sentiment prediction.

Introduction:

In the era of e-commerce, millions of product reviews are generated daily on platforms like Amazon, Flipkart, and eBay. These reviews significantly influence customer decisions and company reputation. However, manually interpreting such massive textual data is inefficient and prone to bias. Hence, **Sentiment Analysis**, a subfield of NLP, provides an automated solution to detect customer emotions (positive or negative) from textual feedback.

The purpose of this project is to build a robust sentiment classifier capable of distinguishing positive and negative product reviews with high accuracy. By combining **TF-IDF vectorization** and **machine learning algorithms**, the model learns word patterns and contextual features that reflect sentiment orientation.

Beyond prediction, the project also performs in-depth EDA to uncover hidden business insights - such as most common complaint topics, length distribution patterns, and category-wise satisfaction trends - to support data-driven decision-making.

Data Source:

The dataset "Amazon Reviews" was downloaded from Kaggle. It consists of `train.csv` and `test.csv` files containing three fields — *polarity*, *title*, and *text*. Polarity values 1 and 2 were mapped to `0 (Negative)` and `1 (Positive)` respectively.

Code:

```
import os

import re

import time

import joblib

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split,
GridSearchCV, RandomizedSearchCV, cross_val_score
```

```
from sklearn.feature_extraction.text import
TfidfVectorizer

from sklearn.metrics import accuracy_score,
precision_score, recall_score, f1_score,
classification_report, confusion_matrix

from sklearn.naive_bayes import MultinomialNB

from sklearn.linear_model import LogisticRegression

from lightgbm import LGBMClassifier

import nltk

nltk.download('stopwords')

from nltk.corpus import stopwords

STOP_WORDS = set(stopwords.words('english'))

import kagglehub

import os

# Download latest version

path = kagglehub.dataset_download("kritanjalijain/amazon-
reviews")

print("Path to dataset files:", path)

# Use `path` instead of `dataset_dir`

train_path = os.path.join(path, "train.csv")

test_path  = os.path.join(path, "test.csv")
```

```
train_df = pd.read_csv(train_path, header=None,
names=['polarity','title','text'])

test_df  = pd.read_csv(test_path,  header=None,
names=['polarity','title','text'])

print("Train shape:", train_df.shape)

print("Test shape :", test_df.shape)

train_df.head()
```

Data Preprocessing:

1) Combining Text and Label Encoding:

Before training any text classification model, it is essential to prepare the data properly. The given code performs two key preprocessing steps: **text combination** and **label mapping**.

Code:

```
train_df['combined'] = train_df['title'].astype(str)
+ " " + train_df['text'].astype(str)

test_df['combined']  = test_df['title'].astype(str)
+ " " + test_df['text'].astype(str)

# Map polarity: 1 -> 0 (negative), 2 -> 1 (positive)

train_df['label'] = train_df['polarity'].map({1:0,
2:1})

test_df['label']  = test_df['polarity'].map({1:0,
2:1})
```

```
print(train_df[['polarity','label']].value_counts())
```

Step 1: Creating a Unified Text Column

- In the dataset, the customer review is split into two parts — a **title** (short summary) and the **main review text** (detailed feedback).
- To allow the model to learn from both parts simultaneously, these two columns are **concatenated into a single column** called `combined`.
- Example:
  - **Title:** "Great Product!"
  - **Text:** "The sound quality is amazing and battery lasts long."
  - **Combined:** "Great Product! The sound quality is amazing and battery lasts long."
- This ensures that the sentiment analysis model captures **contextual cues** from both the short and long parts of the review.

Step 2: Encoding Sentiment Labels

- The dataset originally uses a column called **`polarity`** to represent sentiment:
  - `1` → Negative Review
  - `2` → Positive Review
- However, most machine learning algorithms expect labels to start from **0** for binary classification.
- Therefore, a new column called **`label`** is created using the `.map()` function:
  - `polarity = 1` → `label = 0` (Negative)
  - `polarity = 2` → `label = 1` (Positive)
- This standardization makes the data compatible with classification models like Logistic Regression, Naive Bayes, or Neural Networks.

Why This Step:

- Combining text improves the richness of input data, enabling models to better understand tone and context.
- Encoding labels standardizes the target variable for model training.

- Ensuring consistency between training and testing sets prevents mapping mismatches that could cause prediction errors.

2) Text Cleaning and Preprocessing:

Before feeding textual data into a sentiment analysis model, it is essential to clean and standardize it. Raw customer reviews often contain unwanted characters, URLs, symbols, and filler words (stopwords) that do not contribute meaningful information. The following preprocessing steps were applied to both training and testing datasets.

Code:

```
import re

import nltk

from nltk.corpus import stopwords

nltk.download('stopwords')

STOP_WORDS = set(stopwords.words('english'))

def clean_text(text):

    text = str(text)

    text = re.sub(r'\\n', ' ', text)           #
remove newline symbols

    text = re.sub(r'http\S+', ' ', text)       #
remove URLs

    text = re.sub(r'[^a-zA-Z\s]', ' ', text)   #
keep only alphabets
```

```
    text = text.lower()                          #
convert to lowercase

    text = re.sub(r'\s+', ' ', text).strip()   #
remove extra spaces

    tokens = [w for w in text.split() if w not in
STOP_WORDS]  # remove stopwords

    return " ".join(tokens)

# Combine title and text, then clean

train_df['combined'] = train_df['title'].astype(str)
+ " " + train_df['text'].astype(str)

test_df['combined']  = test_df['title'].astype(str)
+ " " + test_df['text'].astype(str)

train_df['clean_text'] =
train_df['combined'].apply(clean_text)

test_df['clean_text']  =
test_df['combined'].apply(clean_text)

train_df[['combined','clean_text']].head()
```

Step 1: Importing Required Libraries

The code uses the **`re`** (regular expressions) library for text pattern removal and **NLTK (Natural Language Toolkit)** for language preprocessing tasks like removing stopwords.

Step 2: Stopword Removal

Stopwords such as “the”, “is”, “in”, “on” do not contribute to sentiment meaning. Removing them ensures that the model focuses on more informative words like *“great”*, *“broken”*, *“love”*, or *“disappointed”*.

Step 3: Cleaning Function Explanation

The custom `clean_text()` function performs the following operations sequentially:

1. **Convert to string** – ensures text data type uniformity.
2. **Remove newline and escape sequences** – cleans messy formatting.
3. **Remove URLs** – links are irrelevant to sentiment meaning.
4. **Retain only alphabets** – removes punctuation, numbers, and special characters.
5. **Convert to lowercase** – ensures consistent representation of words.
6. **Remove extra spaces** – avoids irregular spacing.
7. **Remove stopwords** – eliminates frequent but uninformative words.

Step 4: Combining Title and Text

Just like in the earlier preprocessing step, the review's **title** and **main text** are concatenated into a single string (`combined`). This combined field captures the complete context of the customer's opinion.

Step 5: Applying the Cleaning Function

The `apply()` method is used to execute the cleaning function on each review in both training and testing datasets, producing a new column `clean_text` that stores the processed review text.

Word Embedding:

Once the review texts are cleaned, they must be converted into a **numerical format** so that machine learning or deep learning models can process them. This conversion is achieved through **word embeddings**, which transform words or sentences into dense numerical vectors that capture **semantic meaning and context**.

In this project, various word representation techniques were explored — including **TF-IDF (Term Frequency–Inverse Document Frequency)** and **neural embeddings** such as **Word2Vec** or **BERT** — to represent the cleaned text data effectively.

Unlike traditional methods like one-hot encoding (which assign a separate index to each word), **word embeddings** encode words into continuous-valued vectors in such a way that:

- **Semantically similar words** (e.g., "great", "amazing", "awesome") are represented by **similar vectors** in the embedding space.
- Words with **opposite meaning** (e.g., "good" vs "bad") lie far apart in that space.

Mathematically, each word is represented as a **dense vector** of fixed dimensions (e.g., 100 or 300), and relationships among words are learned automatically from data.

Embedding Methods:

a) Word2Vec:

After cleaning and preprocessing the text data, the next critical step in sentiment analysis is converting text into numerical form that machine learning models can understand.

While methods like Bag of Words or TF-IDF rely purely on frequency counts, **Word2Vec** represents each word as a dense vector that captures its *semantic meaning* and *contextual relationships* with other words.

Developed by Google in 2013, Word2Vec models revolutionized Natural Language Processing (NLP) by learning distributed representations of words through neural networks.

Word2Vec learns word associations from a large corpus of text.
It has two main architectures:

- **CBOW (Continuous Bag of Words):** Predicts a word based on the surrounding context words.
- **Skip-Gram:** Predicts context words based on the current (target) word.

Each word is represented as a vector in a continuous vector space, such that semantically similar words are located close to each other.
For example:

- "Good" and "Great" might be closer together.
- "Bad" and "Terrible" would form a separate negative cluster.

This property makes Word2Vec powerful for sentiment analysis, as it helps the model recognize that *fantastic* and *excellent* express similar emotions even if they appear in different reviews.

Advantages:

1. **Captures Semantic Relationships:**
    - Word2Vec represents words in a continuous vector space where similar words are positioned close together.
    - For example, vectors for *"king"*, *"queen"*, and *"royal"* appear in similar regions, allowing models to understand context and relationships better than bag-of-words or TF-IDF.
2. **Efficient and Scalable:**
    - Word2Vec models (CBOW and Skip-Gram) are computationally efficient and can be trained on large corpora quickly using techniques like negative sampling and hierarchical softmax.
3. **Dimensionality Reduction:**
    - It produces dense, low-dimensional embeddings (typically 100–300 dimensions), reducing memory usage while retaining meaningful information compared to sparse TF-IDF vectors.
4. **Improves Downstream Task Performance:**
    - The embeddings can be reused for multiple NLP tasks such as sentiment analysis, document classification, and recommendation systems, improving accuracy due to their rich representations.
5. **Transfer Learning Capability:**
    - Pretrained Word2Vec models (like Google News embeddings) can be used directly without retraining, enabling smaller datasets to benefit from knowledge learned on large corpora.

Disadvantages:

1. **Context-Independent Representations:**
    - Each word has a single fixed vector regardless of context.
    - For example, "bank" in *"river bank"* and *"money bank"* will have the same vector, which can lead to misinterpretations.
2. **Cannot Handle Out-of-Vocabulary (OOV) Words:**
    - Words not seen during training are not assigned vectors, making it difficult to process new or rare terms in unseen data.
3. **Requires Large Data for Accuracy:**
    - To learn high-quality embeddings, Word2Vec needs large, diverse text corpora.
    - Small datasets may result in poor vector representations.
4. **Limited to Word-Level Embeddings:**
    - Word2Vec doesn't capture phrase-level or sentence-level meanings unless extended with additional techniques (e.g., averaging or Doc2Vec).
5. **Insensitive to Word Order:**
    - It focuses on co-occurrence and context windows but doesn't preserve the actual order of words in a sentence, unlike transformer-based models like BERT.

b) BERT / DistilBERT Embeddings:

While traditional word embedding methods like Word2Vec and GloVe capture general word meanings, they fail to consider *context.*

For example, the word *"bank"* in *"river bank"* and *"bank account"* has different meanings, but Word2Vec assigns them the same vector.

To address this limitation, researchers at Google introduced **BERT (Bidirectional Encoder Representations from Transformers)** in 2018, marking a revolutionary shift in Natural Language Processing (NLP).

BERT is a deep learning model that understands language context **bidirectionally**, meaning it looks both left and right of a word to interpret its

meaning. This contextual understanding makes BERT highly effective for complex NLP tasks such as **sentiment analysis, question answering, text summarization, and named entity recognition**.

However, BERT is computationally heavy. To make it faster and more suitable for deployment, researchers at Hugging Face introduced **DistilBERT**, a *lighter, distilled version* of BERT that retains most of its performance while being smaller and faster.

BERT Architecture:

BERT is based on the **Transformer architecture**, which relies on *self-attention* mechanisms instead of traditional recurrent or convolutional networks. It uses **two key innovations**:

1. **Bidirectional Encoding**
   Unlike older models (which read text left-to-right or right-to-left), BERT reads text in *both directions simultaneously*, giving it a deep understanding of word relationships.
2. **Pre-training with Two Objectives**
   - **Masked Language Modeling (MLM):** Randomly masks 15% of words in a sentence and predicts them using the context.
     Example: "I love watching [MASK] movies" → predicts "horror".
   - **Next Sentence Prediction (NSP):** Determines if two sentences logically follow each other, improving understanding of sentence-level relationships.

After pre-training on large text corpora like Wikipedia and BookCorpus, BERT can be *fine-tuned* for specific downstream tasks such as **sentiment classification** with minimal abelled data.

DistilBERT:

DistilBERT applies *knowledge distillation* — a process where a large model (the "teacher" - BERT) transfers its learned knowledge to a smaller "student" model.

This reduces the number of parameters by around 40% while retaining 95–97% of BERT's performance.

Key differences:

- 6 layers (instead of 12 in BERT-base)
- Smaller embedding dimension (768 → 512)
- Faster inference time and reduced memory usage

This makes DistilBERT highly suitable for **real-time or resource-limited deployments**, such as in web apps or mobile sentiment analysis tools.

Advantage:

- **Context-Aware Understanding**
  Unlike TF-IDF or Word2Vec, BERT embeddings change depending on the word's context.
  Example: "light" in "light bulb" vs. "light meal" produces different embeddings.
- **Bidirectional Comprehension**
  BERT understands words in both directions, improving accuracy in sentiment and intent recognition.
- **High Accuracy on Downstream Tasks**
  BERT and DistilBERT achieve **state-of-the-art results** on sentiment analysis, text classification, and other NLP benchmarks.
- **Transfer Learning Efficiency**
  Pre-trained on massive corpora, BERT can be easily fine-tuned for specific business tasks (e.g., customer feedback, fraud detection) with minimal data.
- **Rich Semantic Representation**
  Embeddings capture subtle emotions, sarcasm, and complex expressions — vital for understanding nuanced reviews like "Not bad at all" (positive sentiment).

Disadvantage:

- **High Computational Cost**
  BERT models require GPUs or TPUs for efficient training and inference. DistilBERT alleviates this but still remains heavier than TF-IDF or Word2Vec.
- **Complex Fine-Tuning**
  Fine-tuning requires expertise in deep learning, GPU setup, and hyperparameter optimization.
- **Slower Inference for Large Datasets**
  When processing millions of reviews, even DistilBERT can be slower compared to simpler vectorizers.
- **Less Interpretability**
  While embeddings are powerful, they are *black-box representations*, making it hard to explain why the model made a particular prediction.
- **Memory Intensive**
  Large transformer models can consume several gigabytes of memory, limiting their use in low-resource environments.

c) TF-IDF (Term Frequency–Inverse Document Frequency):

In Natural Language Processing (NLP), transforming raw text into a numerical format that machine learning models can interpret is crucial. One of the most widely used and interpretable techniques for this purpose is **TF-IDF (Term Frequency–Inverse Document Frequency)**.

TF-IDF represents the importance of a word in a document relative to all other documents in a corpus. It overcomes a key limitation of simpler methods like **Bag of Words (BoW)** — which only counts word occurrences but fails to differentiate between commonly used and rare yet meaningful words.

By assigning higher weights to less frequent but more informative terms, TF-IDF captures the uniqueness and discriminative power of words, making it extremely effective for **sentiment analysis, document classification, spam detection, and information retrieval**.

The TF-IDF model is based on two main components: **Term Frequency (TF)** and **Inverse Document Frequency (IDF)**.

TF measures how often a word appears in a document. The idea is that a word's frequency in a specific document indicates its relevance to that document.

$$\mathbf{TF(t, d)} = \frac{\text{Number of times term t appears in document d}}{Total\ Number\ of\ terms\ in\ document\ d}$$

IDF measures how unique or rare a word is across all documents.
Words that occur frequently in many documents (like "the", "and", "is") are less informative and are assigned a lower weight.

$$IDF(t) = log\frac{N}{n_t}$$

where **N** = total number of documents, and $\mathbf{n_t}$ = number of documents containing the term *t*.

The final TF-IDF weight for a word *t* in document *d* is:

$$\text{TF} - \text{IDF(t, d)} = \text{TF(t, d)} \times \text{IDF(t)}$$

This ensures that a term's weight is high if it appears frequently in a document but rarely across other documents.

Advantage:

- **Simplicity and Interpretability**
  TF-IDF is easy to compute and understand, making it a great baseline for most NLP tasks.
- **Reduces Noise from Common Words**
  The IDF component down-weights frequent words that carry little meaning (e.g., "the", "is"), improving model focus on meaningful terms.
- **Good for Sparse, High-Dimensional Data**
  Especially useful for text classification where most words don't appear in all documents.

- **Computationally Efficient**
  TF-IDF can be computed quickly, even for large datasets, without requiring heavy hardware like GPUs.
- **Strong Baseline Performance**
  Despite being simple, TF-IDF often matches or outperforms complex embeddings on smaller datasets or specific domains.

Disadvantage:

- **Lack of Context Understanding**
  TF-IDF treats each word independently and doesn't capture context or meaning.
  Example: "not good" and "good" might have similar TF-IDF vectors despite opposite sentiments.
- **Fixed Vocabulary**
  The model doesn't adapt to unseen words — new terms during inference are ignored unless retrained.
- **High Dimensionality**
  As vocabulary size grows, feature space becomes large and sparse, increasing memory and computational needs.
- **No Semantic Similarity**
  Words like "excellent" and "fantastic" are treated as entirely different features, even though they share similar meaning.
- **Limited Scalability for Dynamic Texts**
  TF-IDF struggles when new documents are constantly added, as IDF values must be recalculated across the entire corpus.

Code:

```
vectorizer = TfidfVectorizer(max_features=5000,
ngram_range=(1,2), stop_words='english',
sublinear_tf=True)

X_train =
vectorizer.fit_transform(train_df['clean_text'])
```

```
X_test  =
vectorizer.transform(test_df['clean_text'])

y_train = train_df['label'].values

y_test  = test_df['label'].values

print("TF-IDF shapes:", X_train.shape, X_test.shape)
```

- **`max_features=5000`**

  This restricts the vocabulary to the top 5,000 most important words (based on frequency and relevance). This helps in reducing **dimensionality** and **computational load**, ensuring that the most significant terms are retained for model training.

- **`ngram_range=(1, 2)`**

  This setting includes both **unigrams (single words)** and **bigrams (two-word phrases)**. For instance, the phrase "not good" is treated as a meaningful bigram, allowing the model to capture **contextual sentiment patterns** that single words alone might miss.

- **`stop_words='english'`**

  This removes common English words such as "the", "is", "and", etc., which carry little or no semantic meaning in sentiment analysis. Removing these reduces noise and enhances the quality of the extracted features.

- **`sublinear_tf=True`**

  This applies a logarithmic scaling to term frequency values. Instead of directly using the raw count, TF values are scaled,which helps **reduce the impact of extremely frequent words** that could dominate the vector representation.

- **`vectorizer.fit_transform(train_df['clean_text'])`**

  Learns the vocabulary and the IDF values from the training dataset and simultaneously transforms the cleaned text into a numerical TF-IDF matrix. Each row in `X_train` represents a review, and each column corresponds to a word or phrase from the 5,000-word vocabulary.

- **`vectorizer.transform(test_df['clean_text'])`**

  Transforms the test data using the same vocabulary and IDF weights learned

from the training set.

This ensures **consistency** between training and testing feature representations.

- **`y_train` and `y_test`**

  These arrays store the corresponding sentiment labels (0 for negative, 1 for positive) for each review.

Model Evaluation Function:

To evaluate the performance of different machine learning models on the TF-IDF-based sentiment dataset, a custom evaluation function named `evaluate_and_report()` was implemented.

This function automates the process of training a model, generating predictions, computing key metrics, and visualizing results.

Code:

```
def evaluate_and_report(model, X_train_local,
y_train_local, X_test_local, y_test_local, show_cm=True):

    start = time.time()

    model.fit(X_train_local, y_train_local)

    train_time = time.time() - start

    y_pred = model.predict(X_test_local)

    acc = accuracy_score(y_test_local, y_pred)

    prec = precision_score(y_test_local, y_pred)

    rec = recall_score(y_test_local, y_pred)

    f1 = f1_score(y_test_local, y_pred)

    print(f"\n {model.__class__.__name__} ")
```

```
    print(f"Train time (s): {train_time:.2f}")

    print(f"Accuracy: {acc:.4f}  Precision: {prec:.4f}
Recall: {rec:.4f}  F1: {f1:.4f}")

    print("\nClassification Report:\n",
classification_report(y_test_local, y_pred))

    if show_cm:

        cm = confusion_matrix(y_test_local, y_pred)

        plt.figure(figsize=(4,3))

        sns.heatmap(cm, annot=True, fmt='d',
cmap='Blues',

                    xticklabels=['Neg','Pos'],
yticklabels=['Neg','Pos'])

        plt.title(f"{model.__class__.__name__} Confusion
Matrix")

        plt.xlabel("Predicted")

        plt.ylabel("Actual")

        plt.show()

    return {'model': model.__class__.__name__,
'accuracy': acc, 'precision': prec,

            'recall': rec, 'f1': f1, 'train_time':
train_time}
```

Explanation:

The model is trained on the training dataset, and the training time is recorded. This allows comparison of models not only by performance metrics but also by **computational**

efficiency. The trained model is used to predict the sentiment labels (positive or negative) on the unseen test data.

These standard evaluation metrics are computed:

- **Accuracy:** Overall correctness of the model's predictions.
- **Precision:** Out of all predicted positives, how many were truly positive.
- **Recall:** Out of all actual positives, how many were correctly identified.
- **F1-Score:** Harmonic mean of precision and recall, providing a balanced measure of both.

This generates detailed per-class metrics (precision, recall, F1) for both positive and negative sentiments. The confusion matrix visually represents how many reviews were correctly or incorrectly classified into each sentiment category. This visualization is useful for identifying **biases** - for example, if the model predicts positive sentiment more often than negative. The function returns a dictionary containing all key metrics for easy tabular comparison among multiple models.

a) Accuracy:

Accuracy measures the proportion of correctly predicted reviews (both positive and negative) out of the total number of reviews.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Where:

- **TP (True Positive)** – correctly predicted positive reviews
- **TN (True Negative)** – correctly predicted negative reviews
- **FP (False Positive)** – negative reviews predicted as positive
- **FN (False Negative)** – positive reviews predicted as negative

b) Precision:

Precision quantifies how many of the reviews predicted as **positive** were actually positive.

$$Precision = \frac{TP}{TP + FP}$$

c) Recall:

Recall (also called **Sensitivity** or **True Positive Rate**) measures how many of the **actual positive reviews** were correctly identified by the model.

$$Recall = \frac{TP}{TP + FN}$$

d) F1-Score:

F1-score is the **harmonic mean** of Precision and Recall. It provides a single value that balances both metrics.

$$F1\ Score = 2 \times \frac{(\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})}$$

Model Tuning and Selection:

To identify the most effective sentiment classification model, four supervised learning algorithms were trained and optimized using **TF-IDF embeddings** as input features.
The models tuned include **Logistic Regression**, **Multinomial Naive Bayes**, **LightGBM**, and **XGBoost**.

Hyperparameter tuning was performed using **GridSearchCV** and **RandomizedSearchCV** with **F1-score** as the optimization metric.

a) Logistic Regression:

Logistic Regression is a linear model that estimates the probability of a review belonging to the positive or negative class using a sigmoid function.

It's often used as a baseline model for text classification because of its interpretability and efficiency.

Code:

```
lr = LogisticRegression(max_iter=500, n_jobs=-1)

lr_params = {'C': [0.1, 1.0, 3.0], 'solver':
['liblinear', 'lbfgs']}
```

```
lr_gs = GridSearchCV(lr, lr_params, scoring='f1',
cv=3, n_jobs=-1, verbose=1)

lr_gs.fit(X_train, y_train)
```

Explanation:

- `C` controls the **regularization strength** — smaller values imply stronger regularization.
- `solver` defines the optimization algorithm (`liblinear` for small datasets, `lbfgs` for larger ones).
- The model was tuned using **3-fold cross-validation** to ensure generalization.
- The best combination of hyperparameters was chosen based on the **highest mean F1-score** across folds.

b) Multinomial Naive Bayes:

Naive Bayes is a **probabilistic model** based on Bayes' theorem, assuming feature independence. It is particularly effective for text classification tasks, as it models word frequencies using the multinomial distribution.

Code:

```
nb = MultinomialNB()

nb_params = {'alpha': [0.1, 0.5, 1.0]}

nb_gs = GridSearchCV(nb, nb_params, scoring='f1',
cv=3, n_jobs=-1, verbose=1)

nb_gs.fit(X_train, y_train)
```

Explanation:

- The **alpha** parameter controls smoothing to avoid zero probabilities for unseen words.

- Despite its simplicity, Naive Bayes is computationally very fast and robust to high-dimensional TF-IDF features.

c) LightGBM Classifier:

LightGBM is a **gradient boosting framework** that builds trees sequentially and efficiently using histogram-based algorithms.
It supports GPU acceleration and handles large feature spaces effectively.

Code:

```
lgb_model = LGBMClassifier(random_state=42,
device='gpu', force_col_wise=True, n_jobs=1)

lgb_params = {'num_leaves': [31, 50],
'learning_rate': [0.05, 0.1], 'n_estimators': [100,
200], 'max_depth': [-1, 10]}

lgb_rs = RandomizedSearchCV(lgb_model, lgb_params,
n_iter=6, scoring='f1', cv=3, random_state=42,
n_jobs=1, verbose=1)

lgb_rs.fit(X_train, y_train)
```

Explanation:

- `num_leaves` and `max_depth` control model complexity.
- `learning_rate` defines the step size for gradient updates.
- `n_estimators` sets the number of boosting iterations.
- GPU acceleration (`device='gpu'`) significantly reduces training time.

d) XGBoost Classifier:

XGBoost is another gradient boosting algorithm known for high predictive power, regularization, and parallelized training. It uses an ensemble of weak learners (decision trees) to minimize the overall error iteratively.

Code:

```
xgb_model =
XGBClassifier(objective='binary:logistic',
eval_metric='logloss', random_state=42, n_jobs=1,
tree_method='gpu_hist')

xgb_params = {'learning_rate': [0.05, 0.1],
'max_depth': [4, 6], 'n_estimators': [100, 200],
'subsample': [0.8, 1.0]}

xgb_rs = RandomizedSearchCV(xgb_model, xgb_params,
n_iter=6, scoring='f1', cv=3, random_state=42,
n_jobs=1, verbose=1)

xgb_rs.fit(X_train, y_train)
```

Explanation:

- `tree_method='gpu_hist'` enables GPU-based acceleration.
- `subsample` adds randomness to reduce overfitting.
- The tuning process balances bias–variance through controlled boosting depth and learning rate.

e) Final Model Choice:

Based on the **F1-score** and **cross-validation results**, **XGBoost** emerged as the best-performing model. It achieved the **highest F1-score**, indicating a strong balance between precision and recall.

Furthermore, its scalability and GPU support make it the most suitable for deployment in large-scale sentiment analysis applications.

Model Evaluation and Comparison:

After hyperparameter tuning, the next step is to **evaluate the performance** of the optimized models on the test set.

We used the **evaluate_and_report** function to measure **accuracy, precision, recall, F1-score**, and training time for each model, along with the **confusion matrix** for visual insight.

Code:

```
results = []

# Evaluate tuned models

res_nb = evaluate_and_report(best_nb, X_train, y_train,
X_test, y_test)

results.append(res_nb)

res_lr = evaluate_and_report(best_lr, X_train, y_train,
X_test, y_test)

results.append(res_lr)

res_lgb = evaluate_and_report(best_lgb, X_train, y_train,
X_test, y_test)

results.append(res_lgb)

res_xgb = evaluate_and_report(best_xgb, X_train, y_train,
X_test, y_test)

results.append(res_lgb)  # Note: should append res_xgb

# Create comparison DataFrame

results_df = pd.DataFrame(results).sort_values(by='f1',
ascending=False).reset_index(drop=True)

print("\nModel comparison after tuning:\n", results_df)
```

Evaluation Procedure:

- Each model was trained on the **TF-IDF vectorized training data** (`X_train`) and tested on the **TF-IDF test data** (`X_test`).
- The **F1-score** was used as the primary metric to compare models because it balances **precision and recall**, which is crucial in sentiment analysis where misclassifying positive/negative reviews can have business consequences.
- A **confusion matrix** was plotted to visually inspect correct vs misclassified predictions.

Misclassified Review Analysis:

After selecting the best model based on **F1-score** from the evaluation results, we analyzed the reviews that the model **misclassified**. This step provides **qualitative insights** into model performance and potential areas for improvement.

Code:

```
# Choose the best model by F1 from results_df

best_model_name = results_df.loc[0, 'model']

print("Selected best model:", best_model_name)

model_map = {'MultinomialNB': best_nb,
'LogisticRegression': best_lr, 'LGBMClassifier':
best_lgb}

best_model = model_map[best_model_name]

# Get predictions on test set

y_pred = best_model.predict(X_test)

# Misclassified indices

mis_idx = np.where(y_pred != y_test)[0]
```

```
print("Total misclassified:", len(mis_idx))

# Show first 10 misclassified examples for qualitative
business insight

for i in mis_idx[:10]:

    true_label = "Positive" if y_test[i] == 1 else
"Negative"

    pred_label = "Positive" if y_pred[i] == 1 else
"Negative"

    print("-----------------------------------------
---")

    print(f"Index: {i} | True: {true_label} | Pred:
{pred_label}")

    print("Original text (truncated):")

    print(test_df.iloc[i]['combined'][:500])

    print()
```

Business Insights:

1. **Ambiguous or mixed sentiment**: Many misclassified reviews contain both positive and negative phrases, e.g., "The product is great, but delivery was late." These are challenging even for humans to classify.
2. **Sarcasm and irony**: Some reviews use sarcastic language which the model cannot interpret directly, leading to incorrect predictions.
3. **Short reviews**: Very brief reviews like “Good” or “Bad” sometimes lack sufficient context, increasing misclassification risk.
4. **Domain-specific terms**: Certain product-specific jargon or abbreviations confuse the model if they are not common in the training set.
5. **Implications for business**: Identifying misclassified patterns helps in:

- Improving **customer support** by flagging reviews needing manual intervention.
- Updating **training data** with such edge cases to enhance model accuracy.
- Refining **business decisions** related to product issues, shipping delays, or marketing strategies.

Real-life Problem Solving:

- **Customer Retention**: Quickly identify customers who appear dissatisfied despite seemingly positive words.
- **Product Improvement**: Detect recurring issues hidden in mixed sentiment reviews.
- **Marketing Strategy**: Understand which product features lead to confusion or misinterpretation in reviews.
- **Sentiment Dashboard Accuracy**: Reduces false insights in automated dashboards for business teams.

Streamlit Deployment: Amazon Reviews Sentiment Analysis

To make the sentiment analysis project **interactive and user-friendly**, we deployed it as a **Streamlit web app**. The app allows users to input review text and receive real-time sentiment predictions while also providing an overview of the project.

App Structure:

1. **Dependencies and Model Loading:**
    - **TF-IDF Vectorizer** converts the raw text into numerical features.
    - **Trained model** (e.g., Logistic Regression) predicts whether a review is positive or negative.
    - `@st.cache_resource` ensures the model and vectorizer are loaded **once**, improving performance.
2. **Text Preprocessing:**
    - Removes URLs, newline characters, punctuation.

- Converts text to **lowercase** and removes **stopwords**.
- Ensures consistency with the preprocessing used during training.

3. **Pages:**

The app has **two main pages** accessible via the sidebar:

1) Introduction

- Provides a project overview and methodology:
  - Text cleaning
  - TF-IDF feature extraction
  - Machine learning model prediction
- Lists algorithms tried and evaluation metrics.
- Gives users context before trying predictions.

2) Predict Sentiment

- Users can input their **own Amazon review**.
- On clicking **Analyze**, the text is:
  1. Cleaned
  2. Vectorized using the trained TF-IDF vectorizer
  3. Predicted by the ML model
- Outputs the **predicted sentiment** (Positive/Negative) and the cleaned text for transparency.

4. **User Interaction Example**

- Input: `"This product is amazing, works perfectly!"`
- Output: `Positive`
- Input: `"Terrible experience, broke in a week."`
- Output: `Negative`

Advantages of Streamlit Deployment:

- **Interactive** – Users can instantly test sentiment analysis on their own reviews.

- **Accessible** – Works in browsers without requiring programming knowledge.
- **Transparent** – Shows cleaned text for understanding model preprocessing.
- **Modular** – Easy to swap models (Naive Bayes, LightGBM, etc.) or update the vectorizer.
- **Extensible** – Additional features like visualizations, misclassified review insights, or EDA can be added.

Business Insights:

- **Customer Service**: Detect dissatisfied customers in real time.
- **Product Improvement**: Highlight recurring complaints or praise.
- **Marketing**: Understand features that generate positive sentiment.
- **Feedback Monitoring**: Rapid sentiment monitoring across thousands of reviews.

Code:

```
%%writefile app.py

import streamlit as st

import joblib

import re

import nltk

from nltk.corpus import stopwords

# Load dependencies

nltk.download('stopwords', quiet=True)

STOP_WORDS = set(stopwords.words('english'))

# Load vectorizer and model

@st.cache_resource
```

```
def load_model_and_vectorizer():
    vectorizer =
joblib.load("best_tfidf_vectorizer.joblib")
    model =
joblib.load("best_model_LogisticRegression.joblib")   #
change name if different
    return vectorizer, model
vectorizer, model = load_model_and_vectorizer()
# Text cleaning
def clean_text(text):
    text = str(text)
    text = re.sub(r'\\n', ' ', text)
    text = re.sub(r'http\S+', ' ', text)
    text = re.sub(r'[^a-zA-Z\s]', ' ', text)
    text = text.lower()
    text = re.sub(r'\s+', ' ', text).strip()
    tokens = [w for w in text.split() if w not in
STOP_WORDS]
    return " ".join(tokens)
# Streamlit Pages
st.set_page_config(page_title="Sentiment Analysis App",
page_icon="💬", layout="centered")
```

```
page = st.sidebar.radio("Navigation", ["Introduction",
"Predict Sentiment"])

# PAGE 1: INTRO

if page == "Introduction":

    st.title("Amazon Reviews Sentiment Analysis")

    st.markdown("""

    ### Overview

    This app analyzes **Amazon product reviews** and
predicts whether the sentiment is **Positive** or
**Negative**.

    #### How It Works

    1. Text is cleaned (stopwords, URLs, punctuation
removed)

    2. TF-IDF vectorizer converts text to numeric
features

    3. A trained ML model (e.g., Logistic Regression /
Naive Bayes) predicts sentiment

    #### Pages

    - **Introduction** – About the project

    - **Predict Sentiment** – Try your own review text!

    #### Model Info

    - Vectorizer: TF-IDF (max_features=5000, bigrams)
```

```
    - Algorithms tried: Logistic Regression, Naive Bayes,
LightGBM

    - Evaluation metrics: Accuracy, Precision, Recall,
F1-score

    ---

    *Try the Predict page to see it in action!*

    """)

# PAGE 2: PREDICT

elif page == "Predict Sentiment":

    st.title("Predict Sentiment from Text")

    st.write("Enter a review below and click **Analyze**
to predict sentiment.")

    user_input = st.text_area("Enter review text:",
height=200)

    if st.button("Analyze"):

        if not user_input.strip():

            st.warning("Please enter some text to
analyze.")

        else:

            clean_input = clean_text(user_input)

            vectorized_input =
vectorizer.transform([clean_input])

            pred = model.predict(vectorized_input)[0]
```

```
            label = "Positive" if pred == 1 else
"Negative"

            st.subheader("Result:")

            st.success(f"The predicted sentiment is
**{label}**")

            st.markdown("---")

            st.markdown("### Cleaned Text Preview")

            st.write(clean_input)
```

Conclusion:

This project demonstrates the end-to-end process of building a **sentiment analysis system** for Amazon product reviews using **natural language processing (NLP)** and **machine learning** techniques. Starting from **data preprocessing**, including text cleaning and stopword removal, we applied **TF-IDF vectorization** to transform textual data into numerical features suitable for model training. Multiple models—**Logistic Regression, Naive Bayes, LightGBM, and XGBoost**—were trained and fine-tuned using **GridSearchCV** and **RandomizedSearchCV**, with performance evaluated through metrics such as **accuracy, precision, recall, and F1-score**.

Through extensive **exploratory data analysis (EDA)**, we gained insights into review characteristics, such as common words, review length trends, and sentiment patterns. Misclassified reviews were analyzed to identify patterns that could help improve model performance or inform business decisions. The **best-performing model** was then deployed in a **Streamlit web app**, enabling users to input review text and receive **real-time sentiment predictions**.

From a business perspective, this project highlights the practical benefits of sentiment analysis: detecting dissatisfied customers early, understanding product strengths and

weaknesses, monitoring trends in customer feedback, and aiding data-driven decision-making in marketing and product improvement.

In conclusion, this work illustrates the **integration of NLP, machine learning, and interactive deployment** to provide actionable insights from textual data, bridging the gap between data science models and real-world business applications. It sets a foundation for further improvements, such as incorporating **Word2Vec, BERT embeddings, or multi-lingual review analysis**, to enhance sentiment detection accuracy and applicability across broader e-commerce platforms.