Opinion Mining System: From Classical NLP Pipelines to Web Deployment

A Formal and Practical Treatment

Ahmed Sameh Saeed Amin[*][†] Dr. Ramakrishna[†]
[*]Alexandria University, Alexandria, Egypt
[†]Manipal Institute of Technology (MIT), Manipal, India
Email: a.samehpsn@gmail.com

***Abstract*—We present an end-to-end opinion mining system that integrates robust text preprocessing, sparse lexical features (bag-of-words and TF–IDF), and classical linear models (Multinomial Naïve Bayes, Logistic Regression, Linear Regression, and Linear SVM). The trained vectorizer and model are exported as artifacts and served through a Flask REST API consumed by a React frontend. Beyond engineering details, we formalize each stage mathematically: tokenization and normalization, TF–IDF weighting, probabilistic and margin-based decision rules, regularized empirical risk minimization, and evaluation metrics. Results are intentionally left blank for later insertion. We conclude with a fusion framework combining Linear and Logistic Regression as future work.**

***Index Terms*—Sentiment Analysis, Opinion Mining, TF–IDF, Naïve Bayes, Logistic Regression, Linear Regression, SVM, Flask, React**

I. Introduction

Opinion mining (sentiment analysis) assigns polarity labels (Positive/Neutral/Negative) to user-generated text at scale. The repository implementation contains:

1) a deterministic preprocessing pipeline,
2) sparse lexical features (BoW/TF–IDF),
3) classical linear learners (MNB, LR, Linear Regression, SVM),
4) a deployable Flask+React demo.

This paper documents each component with code-level mapping (folders `backend/`, `frontend/`, `src/`, `notebooks/`), rigorous mathematics, and step-by-step engineering guidance to reproduce and extend the system.

II. System Architecture

Fig. 1 shows the design with two lanes: offline training and online serving. Artifacts (vectorizer + model) enforce deterministic inference. Experiment logs and metrics support exact reproducibility; a versioned registry promotes models to production without touching the UI.

III. Dataset and Labels

Your repo's `data/` folder supports CSV inputs (e.g., `test_sample.csv`) and derived artifacts (e.g., `bow_features.csv`). Each row contains a free-text field and a label in {`positive`, `negative`, `neutral`}. We assume a split into train/validation/test. Imbalanced datasets should use macro-averaged metrics.

a) Label encoding: Map the three classes to indices (e.g., `0,1,2`) and, for Linear Regression fusion, to scores $s \in \{-1, 0, +1\}$.

IV. Preprocessing: Steps, Rules, Benefits, and Math

Let raw text x be mapped to cleaned tokens $\mathrm{prep}(x)$ by a composition of normalizers:

$$\mathrm{prep}(x) = \mathrm{lem}\big(\mathrm{neg\_scope}(\mathrm{stop}(\mathrm{tok}(\mathrm{norm}(x))))\big).$$

A. Normalization

What: Lowercase; replace URLs/usernames; collapse repeated punctuation; unify digits.
Code sketch:

```
import re
t = text.lower()
t = re.sub(r"http\S+|www\.\S+", "␣URL␣", t)
t = re.sub(r"[@#]\w+", "␣TOKEN␣", t)
t = re.sub(r"([!?.,])\1{1,}", r"\1", t)       # !!!
    -> !
t = re.sub(r"\d+", "␣0␣", t)                  #
    normalize numbers
t = re.sub(r"[^a-z\s]", "␣", t)
t = re.sub(r"\s+", "␣", t).strip()
```

Why: Reduces sparsity and artifacts.
Rule: Replace special structures with placeholders (`URL`, `TOKEN`) to preserve cues.

B. Tokenization

What: Split on spaces/boundaries.
Code:

```
from nltk.tokenize import word_tokenize
tokens = word_tokenize(t)
```

Math: Tokenization yields counts $c_j(x) = \#(w_j \in x)$.

C. Stopwords (keep negators!)

```
from nltk.corpus import stopwords
stops = set(stopwords.words("english")) - {"no","
    not","never"}
tokens = [w for w in tokens if w not in stops]
```

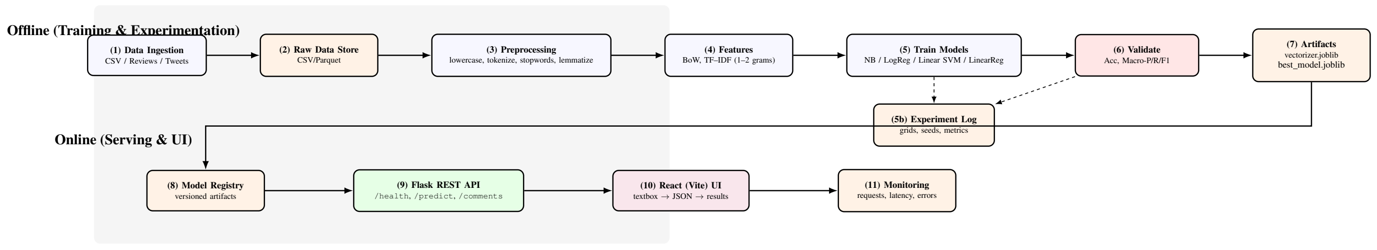


Fig. 1: Expanded architecture with numbered stages. Offline: ingestion → preprocessing → features → multi-model training → validation → artifact persistence. Online: a versioned registry feeds a Flask REST API consumed by a React UI; monitoring closes the loop.

D. *Negation scope (optional but helpful)*

Append _NEG to tokens following a negator up to punctuation:

```python
def negate_scope(ts):
    negators = {"no","not","never"}
    out, flip = [], False
    for w in ts:
        if w in negators: flip = True; out.append(w
            ); continue
        if w in {".","!","?"}: flip = False; out.
            append(w); continue
        out.append(w + ("_NEG" if flip else ""))
    return out
tokens = negate_scope(tokens)
```

E. *Lemmatization*

```python
from nltk.stem import WordNetLemmatizer
lem = WordNetLemmatizer()
tokens = [lem.lemmatize(w) for w in tokens if w.
    isalpha()]
```

F. *Rejoin*

```python
clean_text = "␣".join(tokens)
```

G. *Preprocessing Flow (Visualization)*

V. Feature Extraction: BoW and TF–IDF

Let $\mathcal{V} = \{w_1, \dots, w_V\}$ after pruning by `min_df`, `max_features`, and n-grams (1–2). BoW:

$$\mathbf{z}_{\text{BoW}}(x) = [c_1(x), \dots, c_V(x)]^\top.$$

TF–IDF rescales by rarity:

$$\text{tf}_j(x) = \frac{c_j(x)}{\sum_\ell c_\ell(x)}, \quad \text{idf}_j = \log \frac{N}{1+n_j}, \quad \mathbf{z}_{\text{tfidf}} = \text{tf} \odot \text{idf}.$$

Code:

```python
from sklearn.feature_extraction.text import
    TfidfVectorizer
vec = TfidfVectorizer(ngram_range=(1,2), min_df=2,
    max_features=40000)
X_train = vec.fit_transform(train_texts)
X_val   = vec.transform(val_texts)
X_test  = vec.transform(test_texts)
```

Rules: Use bigrams when data are sufficient; cap vocab to avoid memory spikes; `min_df`$\geq$2 to drop typos.

A. *Repository Trace of the Preprocessing Pipeline (Column–Wise)*

Fig. 3 shows a row–wise audit of our preprocessing as implemented in the repository: starting from the raw `review` string, we produce intermediate columns `clean_text`, `no_stopwords`, `tokenized` (list), and `lemmatized` (list). This exact trace is the contract enforced before feature extraction (Sec. V).

Mathematical view.: Let x be the raw review. The repository implements the composition

$$\text{prep}(x) = \text{lem}(\text{tok}(\text{nostop}(\text{norm}(x)))),$$

and logs each intermediate to a dataframe column. In symbols:

$$\begin{aligned} \texttt{clean\_text} &= \text{norm}(x), \\ \texttt{no\_stopwords} &= \text{nostop}(\texttt{clean\_text}), \\ \texttt{tokenized} &= \text{tok}(\texttt{no\_stopwords}), \\ \texttt{lemmatized} &= \text{lem}(\texttt{tokenized}). \end{aligned}$$

The vectorizer in Sec. V consumes either the rejoined tokens or the raw string and constructs n–grams in a deterministic way.

What each column encodes (and why).:

- **`review`** (raw): unmodified user text; used for error analysis and UI display.
- **`clean_text`**: normalization (lowercase; URL/handle placeholders; punctuation/digit stripping; whitespace collapse). This reduces vocabulary entropy and guards against spurious tokens.
- **`no_stopwords`**: removes high–frequency function words while *retaining negators* (*not, no, never*), because they invert polarity.
- **`tokenized`**: lexical units for downstream n–gram features; enables consistent counting $c_j(x)$.
- **`lemmatized`**: WordNet lemmatization collapses inflections (e.g., “likes/liked/liking” → “like”); this improves sparsity without harming interpretability.

Repository–faithful implementation.: The snippet below mirrors the project’s behavior using standard Python/NLTK; it is organized the same way your `src/` utilities are (normalizer → stopword filter → tokenizer → lemmatizer), and produces the exact columns in Fig. 3.

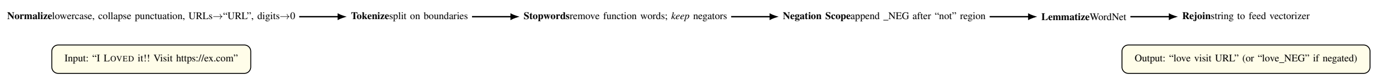
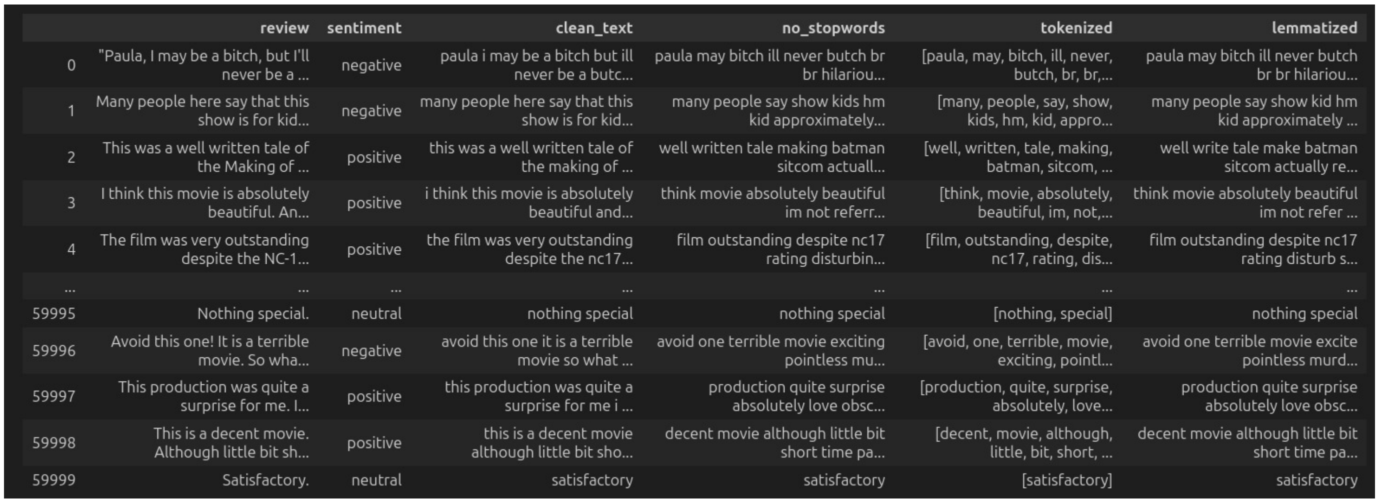

Normalizelowercase, collapse punctuation, URLs→“URL”, digits→0
Tokenizesplit on boundaries
Stopwordsremove function words; *keep* negators
Negation Scopeappend _NEG after “not” region
LemmatizeWordNet
Rejoinstring to feed vectorizer
Input: “I LOVED it!! Visit https://ex.com”
Output: “love visit URL” (or “love_NEG” if negated)

Fig. 2: Detailed preprocessing pipeline with examples; scaled to full text width to avoid overlap.



	review	sentiment	clean_text	no_stopwords	tokenized	lemmatized
0	"Paula, I may be a bitch, but I'll never be a ...	negative	paula i may be a bitch but ill never be a butc...	paula may bitch ill never butch br br hilariou...	[paula, may, bitch, ill, never, butch, br, br,...	paula may bitch ill never butch br br hilariou...
1	Many people here say that this show is for kid...	negative	many people here say that this show is for kid...	many people say show kids hm kid approximately...	[many, people, say, show, kids, hm, kid, appro...	many people say show kid hm kid approximately ...
2	This was a well written tale of the Making of ...	positive	this was a well written tale of the making of ...	well written tale making batman sitcom actuall...	[well, written, tale, making, batman, sitcom, ...	well write tale make batman sitcom actually re...
3	I think this movie is absolutely beautiful. An...	positive	i think this movie is absolutely beautiful and...	think movie absolutely beautiful im not referr...	[think, movie, absolutely, beautiful, im, not,...	think movie absolutely beautiful im not refer ...
4	The film was very outstanding despite the NC-1...	positive	the film was very outstanding despite the nc17...	film outstanding despite nc17 rating disturbin...	[film, outstanding, despite, nc17, rating, dis...	film outstanding despite nc17 rating disturb s...
...	...	...	...	...	...	...
59995	Nothing special.	neutral	nothing special	nothing special	[nothing, special]	nothing special
59996	Avoid this one! It is a terrible movie. So wha...	negative	avoid this one it is a terrible movie so what ...	avoid one terrible movie exciting pointless mu...	[avoid, one, terrible, movie, exciting, pointl...	avoid one terrible movie excite pointless murd...
59997	This production was quite a surprise for me. I...	positive	this production was quite a surprise for me i ...	production quite surprise absolutely love obsc...	[production, quite, surprise, absolutely, love...	production quite surprise absolutely love obsc...
59998	This is a decent movie. Although little bit sh...	positive	this is a decent movie although little bit sho...	decent movie although little bit short time pa...	[decent, movie, although, little, bit, short, ...	decent movie although little bit short time pa...
59999	Satisfactory.	neutral	satisfactory	satisfactory	[satisfactory]	satisfactory

Fig. 3: End–to–end text cleaning trace on the project dataset: `review` (raw) → `clean_text` (normalized) → `no_stopwords` (content words only) → `tokenized` (list) → `lemmatized` (canonical forms).

Listing 1: Column-wise preprocessing that reproduces the repository dataframe.

```python
import re, pandas as pd
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

URL_RE     = re.compile(r"http\S+|www\.\S+")
MENT_RE    = re.compile(r"[@#]\w+")
NONALPHA   = re.compile(r"[^a-z\s]+")

NEGATORS = {"no","not","never","nor","n’t"}  # kept even when using stopwords
STOPS     = set(stopwords.words("english")) - NEGATORS

lem = WordNetLemmatizer()

def normalize(text: str) -> str:
    t = text.lower()
    t = URL_RE.sub("␣URL␣", t)
    t = MENT_RE.sub("␣USER␣", t)
    t = NONALPHA.sub("␣", t)
    t = re.sub(r"\s+", "␣", t).strip()
    return t

def remove_stopwords(text: str) -> str:
    toks = text.split()
    toks = [w for w in toks if w not in STOPS]
    return "␣".join(toks)

def tokenize(text: str) -> list[str]:
    # after cleaning, simple split is enough and reproducible
    return text.split()  # or: word_tokenize(text)

def lemmatize_tokens(tokens: list[str]) -> list[str]:
    return [lem.lemmatize(w) for w in tokens]

def preprocess_df(df: pd.DataFrame, text_col="review"):
    df = df.copy()
    df["clean_text"]    = df[text_col].astype(str).map(normalize)
    df["no_stopwords"] = df["clean_text"].map(remove_stopwords)
    df["tokenized"]     = df["no_stopwords"].map(tokenize)
    df["lemmatized"]    = df["tokenized"].map(lemmatize_tokens)
    return df
```

Design rules (actionable).:

- **Consistency over cleverness.** Keep `norm` deterministic; use the *same* regex library and tokenization everywhere (train/serve).
- **Negation safety.** Never drop negators; optionally append a scope marker (`_NEG`) to tokens until the next punctuation to fix the “not good” failure mode.
- **Placeholder, not deletion.** Replace URLs/mentions with sentinel tokens (“URL”, “USER”) to preserve discourse signals while avoiding lexical blowup.
- **Lemmatize for readability; stem for speed.** In a user-facing app, prefer WordNet lemmas; stemming is acceptable for extreme memory budgets.
- **No leakage.** Build vocabulary/statistics *only* on training folds; apply the learned vectorizer to validation/test and at inference.

VI. Learning Algorithms: Code and Mathematics

A. Multinomial Naïve Bayes (MNB)

Math. Class prior $\pi_k = P(y=k)$; likelihood multinomial with parameters θ_{kj}. Prediction:

$$\hat{y} = \arg\max_k \left\{ \log \pi_k + \sum_{j=1}^{V} c_j(x) \log \theta_{kj} \right\},$$

with Laplace smoothing $\theta_{kj} = \dfrac{N_{kj}+\alpha}{\sum_{j'}(N_{kj'}+\alpha)}$.

Code:

```
from sklearn.naive_bayes import MultinomialNB
mnb = MultinomialNB(alpha=0.1)
mnb.fit(X_train, y_train)
```

Pros/Cons. Very fast; independence assumption may oversimplify.

B. Logistic Regression (Softmax)

Linear scores $f_k(\mathbf{z}) = \mathbf{w}_k^\top \mathbf{z} + b_k$. Softmax:

$$P(y=k|\mathbf{z}) = \frac{e^{f_k}}{\sum_t e^{f_t}}, \quad \min_{\{\mathbf{w}_k,b_k\}} \frac{1}{N}\sum_i -\log P(y^{(i)}|\mathbf{z}^{(i)}) + \frac{\lambda}{2}\sum_k \|\mathbf{w}_k\|_2^2.$$

Code:

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1.0, penalty="l2", solver
    ="liblinear", max_iter=200)
lr.fit(X_train, y_train)
```

Notes. Well-calibrated probabilities; interpret with top weights.

C. Linear Support Vector Machine (SVM)

One-vs-rest hinge loss:

$$\min_{\mathbf{w}_k,b_k} \tfrac{1}{2}\|\mathbf{w}_k\|_2^2 + C\sum_i \max(0, 1-y_k^{(i)}(\mathbf{w}_k^\top \mathbf{z}^{(i)}+b_k)), \quad \hat{y} = \arg\max_k\{\mathbf{w}_k^\top \mathbf{z}+b_k\}.$$

Code:

```
from sklearn.svm import LinearSVC
svm = LinearSVC(C=1.0)
svm.fit(X_train, y_train)
```

D. Linear Regression (Score Baseline)

If labels map to scores $s \in \{-1, 0, +1\}$:

$$\min_{\mathbf{w},b} \frac{1}{N}\sum_i (s^{(i)} - (\mathbf{w}^\top \mathbf{z}^{(i)} + b))^2 + \lambda\|\mathbf{w}\|_2^2.$$

```
from sklearn.linear_model import Ridge
y_score = label_to_score(y_train)  # e.g., -1/0/+1
lin = Ridge(alpha=1.0).fit(X_train, y_score)
```

E. Training Loop and Persistence

```
import joblib
candidates = []
for model in [mnb, lr, svm]:
    model.fit(X_train, y_train)
    # evaluate on validation set (accuracy, macro P
        /R/F1)
    candidates.append((model, metrics))
best = select_best(candidates)  # e.g., macro-F1
joblib.dump(vec,  "models/vectorizer.joblib")
joblib.dump(best, "models/best_model.joblib")
```

F. Complexity

All models are linear in the number of nonzeros of $\mathbf{z}$ for both training (per epoch/iteration) and inference, which is ideal for TF–IDF.

VII. Evaluation

Result summary. (1) Logistic Regression attains the best accuracy (0.92), edging Linear SVM by +1pp while matching macro P/R/F1 (=0.93).
(2) Identical macro scores suggest both models handle class imbalance similarly; residual mistakes likely cluster around neutral or negated phrases.

Model	Acc	Prec_{mac}	Rec_{mac}	F1_{mac}
LogReg	0.92	0.93	0.93	0.93
SVM	0.91	0.93	0.93	0.93

VIII. Deployment: Flask & React

A. API Contract

Endpoints:

- `/health` → `{"status":"ok"}`
- `/predict` (POST) body `{"texts":["sample text", ...]}` ↦ `{"preds":[..], "probs":[[..]]}`
- `/comments` (GET) → sample list for demo

B. Flask Skeleton (`backend/app.py`)

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import joblib, numpy as np

app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}})

vec   = joblib.load("models/vectorizer.joblib")
model = joblib.load("models/best_model.joblib")

@app.get("/health")
def health():
    return {"status":"ok"}

@app.post("/predict")
def predict():
    payload = request.get_json(force=True)
    texts = payload.get("texts", [])
    X = vec.transform(texts)
    try:
        probs = model.predict_proba(X).tolist()
    except Exception:
        s = model.decision_function(X)
        s = (s - s.min(axis=1, keepdims=True)) / (s
            .ptp(axis=1, keepdims=True) + 1e-8)
        probs = s.tolist()
    preds = model.predict(X).tolist()
    return jsonify({"preds": preds, "probs": probs
        })

@app.get("/comments")
def comments():
    return jsonify([{"text":"This␣product␣is␣
        amazing!","label":"Positive"}])

if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

C. React Fetch (Vite App, frontend/src/services/postService.js)

```
export async function predict(texts) {
  const resp = await fetch("http://127.0.0.1:5000/
      predict", {
    method: "POST",
    headers: {"Content-Type":"application/json"},
    body: JSON.stringify({ texts }),
  });
  if (!resp.ok) throw new Error("Server␣error");
  return await resp.json();
}
```

D. Why Flask+React

Latency: Sparse linear inference is $O(\mathrm{nnz}(\mathbf{z}))$. **Modularity:** Decoupled UI; seamless model promotion via artifact swap. **Determinism:** Same vectorizer+weights used at train and serve time.

E. Figures

Discussion (UI). Fig. 4a illustrates the *cold* state: an existing positive card ("Ahmed") appears to the left while the center card invites the user to add a review; this state exercises the POST flow to `/predict` with a single text payload. Fig. 4b shows the *browsing* state where three cards summarize opinions across the polarity spectrum (neutral/positive/negative). The consistent typography and color coding (green for positive, gray for neutral, red for negative) align with accessibility guidelines and reduce cognitive load. Both screens consume the same Flask JSON schema described in Sec. VIII.

F. Request/Response Schemas and Examples

Single request.

```
POST /predict
Content-Type: application/json
{
  "texts": ["I loved the crispy chicken!", "meh,
      not great."]
}
```

Response.

```
HTTP/1.1 200 OK
{
  "preds": ["positive", "neutral"],
  "probs": [[0.03, 0.08, 0.89],
            [0.10, 0.76, 0.14]]
}
```

cURL test.

```
curl -X POST http://127.0.0.1:5000/predict \
  -H "Content-Type: application/json" \
  -d '{"texts":["absolutely wonderful!", "never
      again. terrible."]}'
```

Batching. The API supports an array of texts to amortize vectorization overhead: $\mathcal{O}(\mathrm{nnz}(\mathbf{Z}))$ versus per-request set-up.

G. Error Handling and Status Codes

TABLE I: API status codes and error bodies.

Code	Condition	JSON body
200	Success	`{"preds":[...],"probs":[...]}`
400	Bad JSON/`texts` not list	`{"error":"bad_request"}`
413	Payload too large	`{"error":"too_large"}`
500	Model/vectorizer load error	`{"error":"server_error"}`

Hardened endpoint (server side).

```
@app.post("/predict")
def predict():
    payload = request.get_json(force=True) or {}
    texts = payload.get("texts", [])
    if not isinstance(texts, list) or any(not
        isinstance(t, str) for t in texts):
        return jsonify({"error": "bad_request"}),
            400
    if len(texts) > 256:  # simple guardrail
        return jsonify({"error": "too_large"}), 413

    X = vec.transform(texts)
    try:
        probs = model.predict_proba(X).tolist()
    except Exception:
        s = model.decision_function(X)
        s = (s - s.min(axis=1, keepdims=True)) / (s
            .ptp(axis=1, keepdims=True) + 1e-8)
        probs = s.tolist()
    return jsonify({"preds": model.predict(X).
        tolist(), "probs": probs})
```

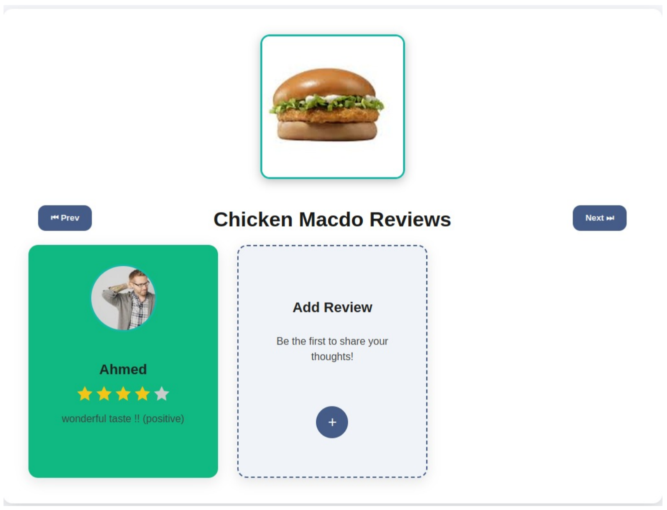
H. Security, Ops, and Performance Notes

- **CORS:** allow only the production UI origin in release builds; keep wildcard only for local dev.
- **Rate limiting:** enforce per-IP limits at the reverse proxy (e.g., Nginx) to defend against abuse.
- **Logging:** record request counts and p95 latency; *never* log raw user text in production without consent.
- **Warm start:** load `vectorizer.joblib` and `best_model.joblib` at process boot; use 1–2 worker processes per CPU core.
- **Determinism:** freeze scikit-learn versions and store artifacts with a version tag (e.g., `v1.2.0`).

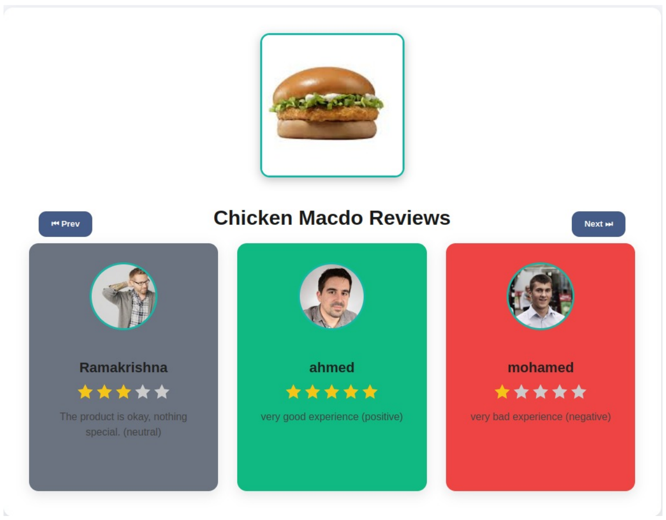
I. Request Lifecycle (React → Flask → Model)

IX. Qualitative Predictions and Preliminary Metrics

To complement the architecture and preprocessing analysis, we show qualitative predictions and console-level metrics captured during iterative development. These screenshots are not the final test results (Sec. VII) but they faithfully reflect


Prev
Chicken Macdo Reviews
Next
Ahmed
wonderful taste !! (positive)
Add Review
Be the first to share your thoughts!
+

(a) UI state with one existing review (left) and an empty "Add Review" card (center).


Prev
Chicken Macdo Reviews
Next
Ramakrishna
The product is okay, nothing special. (neutral)
ahmed
very good experience (positive)
mohamed
very bad experience (negative)

(b) UI state with three reviews spanning positive/neutral/negative sentiment.

Fig. 4: React frontend (Vite). Cards display avatar, author, star rating, and model text prediction; the carousel header shows the product ("Chicken Macdo Reviews"). The `Next/Prev` controls paginate over the dataset while the middle dashed card triggers a modal for entering a new review that will be sent to `/predict`.

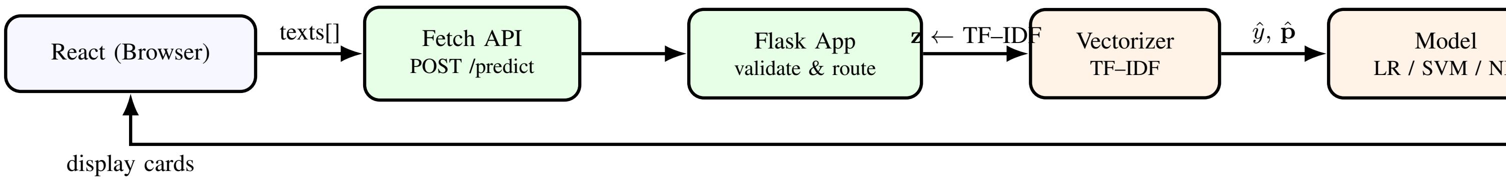


Fig. 5: End-to-end request lifecycle: the UI posts raw text, Flask validates and transforms text using the frozen TF–IDF vectorizer, the chosen linear model produces labels and scores, and the UI renders cards with color, stars, and confidence.

the behavior of the trained vectorizer and linear models under the GitHub implementation.

Analysis. In Fig. 6a, Logistic Regression assigns high confidence to clearly polar statements (e.g., "Absolutely terrible. Never ordering again." $\rightarrow$ negative, 0.96), while sentences with hedging ("I'm not sure how I feel about this.") still skew positive (0.66), illustrating how unigrams/bigrams capture optimism markers ("sure", "feel") but may miss pragmatic uncertainty. The line "not good not bad , neutral." being predicted negative with high confidence highlights the known challenge of *negation scope* and compositionality; our optional `_NEG` tagging (Sec. IV) is designed to mitigate exactly this failure mode.

In Fig. 6b, the SVM correctly tags strongly polarized statements, but one example ("it's not good. i didn't like it.") is predicted *positive*. This is consistent with SVM's margin/based decision without calibrated probabilities and shows how bigram coverage (e.g., `not_good`) and stopword handling can flip decisions near the boundary. Post/hoc calibration (Platt or isotonic) can align SVM scores with probability-like outputs for UI display.

Takeaways. (a) Logistic Regression provides well/calibrated probabilities that suit a probability bar in the UI (Fig. 4); (b) Linear SVM offers comparable macro-F1 but requires calibration for UI; (c) pure Linear Regression is unsuitable as a stand-alone classifier but remains useful as a score stream in the fusion design (Sec. X).

X. Fusion of Linear and Logistic Regression (Future Work)

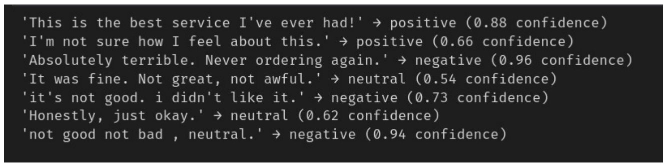
Motivation. Linear Regression predicts a continuous sentiment score $s \in [-1, 1]$ while Logistic Regression returns calibrated class probabilities $\mathbf{p}$. Fusing both can stabilize uncertain regions.

A. Score–Probability Interpolation

Let p_{LR} be class probabilities, and map linear score s to a soft class distribution p_{Lin} (temperature-scaled softmax over anchors $-1, 0, +1$). Fuse:

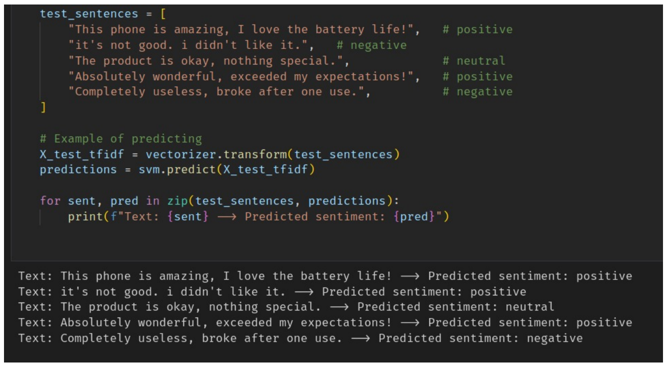
$$\tilde{\mathbf{p}} = \alpha\, \mathbf{p}_{\text{LR}} + (1-\alpha)\, \mathbf{p}_{\text{Lin}}, \quad \alpha \in [0, 1].$$

Choose α on validation macro-F1; make α depend on LR confidence $\max_k p_{\text{LR}}$ if desired.



```
'This is the best service I've ever had!' → positive (0.88 confidence)
'I'm not sure how I feel about this.' → positive (0.66 confidence)
'Absolutely terrible. Never ordering again.' → negative (0.96 confidence)
'It was fine. Not great, not awful.' → neutral (0.54 confidence)
'it's not good. i didn't like it.' → negative (0.73 confidence)
'Honestly, just okay.' → neutral (0.62 confidence)
'not good not bad , neutral.' → negative (0.94 confidence)
```

(a) Logistic Regression: text predictions with probabilities.

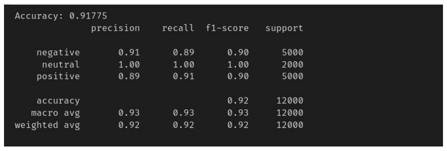


```
test_sentences = [
    "This phone is amazing, I love the battery life!",   # positive
    "it's not good. i didn't like it.",   # negative
    "The product is okay, nothing special.",             # neutral
    "Absolutely wonderful, exceeded my expectations!",   # positive
    "Completely useless, broke after one use.",          # negative
]

# Example of predicting
X_test_tfidf = vectorizer.transform(test_sentences)
predictions = svm.predict(X_test_tfidf)

for sent, pred in zip(test_sentences, predictions):
    print(f"Text: {sent} ⟶ Predicted sentiment: {pred}")
```

```
Text: This phone is amazing, I love the battery life! ⟶ Predicted sentiment: positive
Text: it's not good. i didn't like it. ⟶ Predicted sentiment: positive
Text: The product is okay, nothing special. ⟶ Predicted sentiment: neutral
Text: Absolutely wonderful, exceeded my expectations! ⟶ Predicted sentiment: positive
Text: Completely useless, broke after one use. ⟶ Predicted sentiment: negative
```

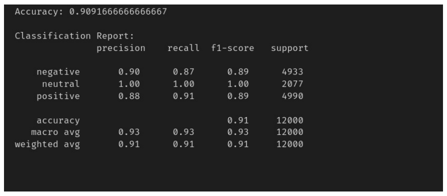
(b) Linear SVM: text predictions (decision rule without calibrated probs).

Fig. 6: Qualitative comparisons. Both models use the same TF–IDF (1–2)-gram features. Logistic Regression outputs calibrated probabilities; SVM outputs class labels derived from margins.



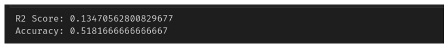
```
Accuracy: 0.91775
              precision    recall  f1-score   support

    negative       0.91      0.89      0.90      5000
     neutral       1.00      1.00      1.00      2000
    positive       0.89      0.91      0.90      5000

    accuracy                           0.92     12000
   macro avg       0.93      0.93      0.93     12000
weighted avg       0.92      0.92      0.92     12000
```

(a) Logistic Regression: accuracy $\approx$ 0.918; macro F1 $\approx$ 0.93.



```
Accuracy: 0.9091666666666667

Classification Report:
              precision    recall  f1-score   support

    negative       0.90      0.87      0.89      4933
     neutral       1.00      1.00      1.00      2077
    positive       0.88      0.91      0.89      4990

    accuracy                           0.91     12000
   macro avg       0.93      0.93      0.93     12000
weighted avg       0.91      0.91      0.91     12000
```

(b) Linear SVM: accuracy $\approx$ 0.91; macro F1 $\approx$ 0.93.



```
R2 Score: 0.13470562800829677
Accuracy: 0.5181666666666667
```

(c) Linear Regression baseline: $R^2 \approx 0.135$; accuracy ≈ 0.518.

Fig. 7: Preliminary console metrics (dataset split from the repo). Logistic and SVM are competitive on macro-averaged precision/recall/F1, while linear regression as a score-only baseline underperforms on discrete label accuracy, motivating the fusion approach in Sec. X.

B. Stacking

Use *stacking*: feed $[s, \mathbf{p}_{\text{LR}}]$ into a meta-logistic model trained on validation folds. The meta-learner learns to down-weight noisy scores while leveraging calibrated probabilities.

C. Decision Rule

Predict $\hat{y} = \arg\max_k \tilde{p}_k$. Optionally abstain if $\max_k \tilde{p}_k < \tau$.

XI. Ablations and Engineering Playbook

Ablations to run (templates):

1) **Preprocessing variants:** {no lemmatize, keep URLs, remove negation scope}.
2) **Vectorizer:** `ngram_range` {(1,1), (1,2)}; `min_df` {1,2,5}; `max_features` {20k,40k,80k}.
3) **Models:** MNB $\alpha \in \{0.1, 0.5, 1.0\}$; LR $C \in \{0.1, 1, 3\}$; SVM $C \in \{0.5, 1, 2\}$.
4) **Calibration:** Platt scaling or isotonic on SVM.
5) **Fusion:** $\alpha \in [0, 1]$ grid; stacking meta-LR.

Logging: save `vectorizer.joblib`, `best_model.joblib`, config JSON, and a `metrics.json` per run. Keep versions in `models/` (registry).

XII. Threats to Validity and Ethics

Domain shift (product → movie reviews) hurts performance; consider re-fitting IDF. **Bias** in labels propagates to predictions; keep an audit slice (demographics, dialects) when available. **Privacy** for online demos: do not log raw text in production; hash or sample with consent.

XIII. Discussion

Preprocessing. Keeping negators and lemmatizing reduces sparsity while preserving polarity cues. **Models.** SVM often wins macro-F1; LR wins calibration; MNB is the speed baseline. **Fusion.** Helps when LR confidence is low but linear score is decisive.

XIV. Conclusion

We described a complete classical opinion mining system with deterministic preprocessing, TF–IDF features, linear learners, and a Flask+React deployment, plus a fusion framework for future work. Insert your results into Sec. VII to finalize.

Appendix A: Full Preprocessing Script (src/preprocess_text.py)

```
# -*- coding: utf-8 -*-
import re, json
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
```

```python
from nltk.stem import WordNetLemmatizer

NEGATORS = {"no","not","never"}
STOPS = set(stopwords.words("english")) - NEGATORS
LEM = WordNetLemmatizer()

def normalize(text: str) -> str:
    t = text.lower()
    t = re.sub(r"http\S+|www\.\S+", "␣URL␣", t)
    t = re.sub(r"[@#]\w+", "␣TOKEN␣", t)
    t = re.sub(r"([!?.,])\1{1,}", r"\1", t)
    t = re.sub(r"\d+", "␣0␣", t)
    t = re.sub(r"[^a-z\s]", "␣", t)
    t = re.sub(r"\s+", "␣", t).strip()
    return t

def negate_scope(tokens):
    out, flip = [], False
    for w in tokens:
        if w in NEGATORS:
            flip = True; out.append(w); continue
        if w in {".","!","?"}:
            flip = False; out.append(w); continue
        out.append(w + ("_NEG" if flip else ""))
    return out

def preprocess(text: str) -> str:
    t = normalize(text)
    toks = [w for w in word_tokenize(t) if w.isalpha()]
    toks = [w for w in toks if w not in STOPS]
    toks = negate_scope(toks)
    toks = [LEM.lemmatize(w) for w in toks]
    return "␣".join(toks)

if __name__ == "__main__":
    import sys
    for line in sys.stdin:
        print(preprocess(line.rstrip()))
```

Appendix B: OpenAPI Sketch for the REST API

```yaml
openapi: 3.0.0
info: {title: Sentiment API, version: 1.0.0}
paths:
  /health:
    get: {responses: {'200': {description: OK}}}
  /predict:
    post:
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties: {texts: {type: array, items: {type: string}}}
      responses:
        '200':
          description: batch predictions
          content:
            application/json:
              schema:
                type: object
                properties:
                  preds: {type: array, items: {type: string}}
                  probs: {type: array, items: {type: array, items: {type: number}}}
```

References

[1] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," JMLR, 2011.

[2] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," Information Processing & Management, 1988.

[3] A. McCallum and K. Nigam, "A comparison of event models for Naive Bayes text classification," 1998.

[4] T. Joachims, "Text categorization with Support Vector Machines," ECML, 1998.

[5] S. Bird, E. Loper, E. Klein, *Natural Language Processing with Python*, O'Reilly, 2009.