SENTIMENT ANALIZER

Final Project

Group Members:

Albaro Blanquicett – C0927639

Gina Ferrera – C0933111

Luis Carlos Muñoz – C0932513

Marian Velazquez – C0937278

Lambton College

2025S-t3 BAM 3034 – Sentiment Analysis

Real-Time Sentiment Analyzer

Executive summary

We built a web application that reads any piece of text (a tweet, a comment, a review) and instantly tells you whether the overall **sentiment** is **Positive, Negative, or Neutral**. The app also shows how confident the system is, visualizes the result with a simple chart, and highlights the **words that most influenced the decision**, plus a word cloud.

Under the hood, the system uses a classic but reliable approach: it cleans the text, turns it into numerical features with **TF-IDF**, and feeds those features into a **Logistic Regression** model trained on the **IMDB movie reviews** dataset (50,000 labeled reviews). We wrapped the trained model in a lightweight **Flask** website with a clean interface and deployed it online so anyone can try it.

Why sentiment analysis?

Online spaces are crowded with opinions. Businesses, researchers, and even casual users want a quick sense of whether people sound positive or negative about a topic. A good sentiment tool can:

- Summarize thousands of comments into a clear signal.
- Track public mood over time.
- Offer early warning when reactions start to sour.

Our goal was to create a **practical, explainable** tool that demonstrates the full journey—from data and modeling to an accessible web experience.

The data

We used the **IMDB 50K Movie Reviews** dataset (balanced between positive and negative reviews). It's a clean, widely used academic resource that's ideal for teaching and benchmarking because it already includes labels ("positive" or "negative") for every review.

Why this matters:

- The dataset is **large enough** to train a solid model.

- Labels allow us to **learn** what distinguishes positive from negative language.
- Movies are a rich domain: reviews often contain emotion, nuance, and slang—useful for training.

What the model actually learns (in plain English)

Computers don't "read" words like we do. We first convert text into numbers. Our approach uses **TF-IDF**, which rewards words that are informative in a given text but not overly common across all texts. For example, "movie" appears everywhere and isn't very helpful, while words like "masterpiece" or "boring" carry stronger sentiment signals.

Once every review is mapped to numbers, a **Logistic Regression** model finds a mathematical boundary that best separates positive from negative examples. It also tells us, for each word, whether it "pushes" toward positivity or negativity and by how much. That transparency is valuable for teaching and trust.

From raw text to ready-to-predict: the cleaning pipeline

Real text is messy. Before the model ever sees it, we apply a simple, repeatable cleaning process:

1. **Lowercasing** (turn EVERYTHING into lowercase so "Great" and "great" are treated the same).
2. **Remove HTML breaks** (e.g.,
 that often appears in scraped reviews).
3. **Remove digits and punctuation** (they rarely express sentiment).
4. **Tokenization and stopword removal** using **spaCy** (we split text into words and drop ultra-common words like "the", "and", "is").
5. **Lemmatization** with **spaCy** (we reduce "loved", "loving", "loves" to the base form **love**, shrinking the vocabulary and improving generalization).

After cleaning, we transform the text with **TF-IDF** (we included single words and two-word phrases). Now the text is a compact numeric vector ready for the model.

```python
# Load spaCy English model
nlp = spacy.load("en_core_web_sm", disable=['ner', 'parser'])  # speed up preprocessing

def spacy_clean(text: str) -> str:
    """Clean + tokenize + lemmatize with spaCy.
    Steps: lowercase, remove digits/punct, remove stopwords/spaces, remove html breaks, lemmatize
    Returns a whitespace-joined string of lemmas.
    """
    text = text.lower()
    # Remove HTML line breaks and other tags
    text = re.sub(r'<br\s*/?>', ' ', text)  # handles <br>, <br/> and <br >
    text = re.sub(r'\d+', ' ', text)  # remove digits
    text = re.sub(f'[{re.escape(string.punctuation)}]', ' ', text)  # remove punctuation
    doc = nlp(text)
    lemmas = []
    for tok in doc:
        if tok.is_space or tok.is_punct or tok.is_stop:
            continue
        lemma = tok.lemma_.strip()
        if lemma:
            lemmas.append(lemma)
    return ' '.join(lemmas)

# Sample cleaning to estimate time
sample = df['review'].iloc[0]
print('Sample (raw)    :', sample[:120].replace('\n',' ') + '...')
print('Sample (cleaned):', spacy_clean(sample)[:120] + '...')

# Clean the whole dataset (this may take a few minutes depending on size)
df['clean'] = df['review'].astype(str).apply(spacy_clean)
df = df.dropna(subset=['clean','sentiment']).reset_index(drop=True)
```

Training and evaluation

We split the dataset into **training** and **test** sets so we can check how well the model performs on text it hasn't seen before. We also ran **cross-validation** (multiple randomized splits) to make sure the results are stable and not lucky.

Typical outcomes for this setup:

- **Accuracy** around **88–91%**
- **F1 score** (a balance of precision and recall) in a similar range

Why we chose Logistic Regression:

- It's **fast**, **robust**, and **interpretable**.
- It provides **probabilities**, which we use to compute a **confidence score**.
- It's an excellent baseline; more complex models (like BERT) can improve accuracy but are heavier to train and deploy.

We experimented with **SVM** as well; performance was comparable, but Logistic Regression's built-in probabilities made downstream UX simpler.

How the prediction is displayed

When you paste text into the app and click **Analyze**, the system:

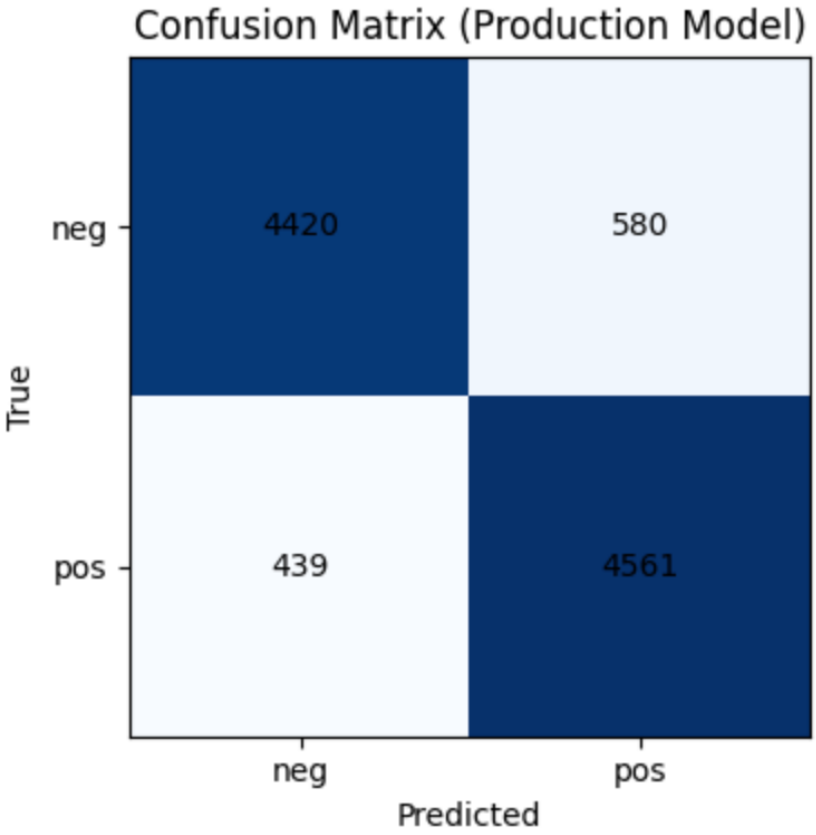
1. Cleans and lemmatizes your text (same steps used during training).
2. Converts it into TF-IDF features.
3. Gets the model's **probability that the text is positive** (call it *p*).
4. Computes a **confidence score** by seeing how far *p* is from 0.5 (pure uncertainty).
    - If the prediction isn't very confident (close to 0.5), we label it **Neutral**.
    - Otherwise, **Positive** if $p \geq 0.5$, **Negative** if $p < 0.5$.

You'll see:

- The **label** (Positive/Negative/Neutral)
- The **positive probability** (a number from 0 to 1)
- A **confidence** number (0 to 1; higher means more sure)
- A simple **doughnut chart** of Positive/Negative/Neutral
- A **word cloud** for your text
- Lists of **key influencing words** (terms in your text that pushed the result up or down based on the trained model)

This last piece—the influencing words—makes the model feel less like a "black box" and more like a partner showing its reasoning.

```
[LogisticRegression] Accuracy: 0.8981 | F1: 0.8995

              precision    recall  f1-score   support

    negative       0.91      0.88      0.90      5000
    positive       0.89      0.91      0.90      5000

    accuracy                           0.90     10000
   macro avg       0.90      0.90      0.90     10000
weighted avg       0.90      0.90      0.90     10000

[LinearSVC (calibrated)] Accuracy: 0.9082 | F1: 0.9087

              precision    recall  f1-score   support

    negative       0.91      0.90      0.91      5000
    positive       0.90      0.91      0.91      5000

    accuracy                           0.91     10000
   macro avg       0.91      0.91      0.91     10000
weighted avg       0.91      0.91      0.91     10000
```



The web app

We built a straightforward **Flask** application:

- **Home/Results** (one page): a text box, **Analyze** and **Clear** buttons, and a results panel that appears under the form.
- **Visuals**: one chart (using **Chart.js**), a word cloud, and two short lists of "positive signals" and "negative signals."
- **Design**: clean dark theme, good contrast, tidy layout, and mobile-friendly.

We intentionally avoided fancy interactions so the focus stays on clarity and learning.

Deployment (how it's available online)

We deployed the app to a cloud hosting platform so anyone can try it in a browser. The production server uses **Gunicorn** (a fast, production-grade Python web server). One note about free hosting tiers: they often "sleep" when idle, which causes a short "waking up" delay on the first visit after inactivity. That's expected and normal.

Strengths and limitations

Strengths

- End-to-end pipeline: from raw data to a polished app.
- Explainability: you can see which words mattered.
- Practical performance with simple, fast technology.
- Lightweight hosting requirements.

Limitations

- The model is trained on **movie reviews**. If you feed it technical financial reports or medical notes, accuracy may drop because the language is different.
- The "Neutral" rule is simple (based on confidence). A true three-class model trained on Neutral examples could do better.
- Sarcasm, humor, and subtle context can fool any sentiment model, not just ours.

Ethical use and privacy

This project is an academic demonstration. It does not store user inputs or identify people. Still, anyone applying sentiment analysis in the real world should:

- Be transparent about how predictions are made and used.
- Avoid high-stakes decisions without human oversight.
- Consider biases in training data (movie reviews might over-represent certain styles or topics).

What we learned

- **Simple methods go a long way.** With careful cleaning and TF-IDF features, a linear model can be both strong and transparent.
- **Reproducibility matters.** Saving the vectorizer and model as artifacts means we can reliably deploy what we trained.
- **UX matters.** A small visual (chart, word cloud) and clear language build trust and help non-technical users.
- **Deployment is part of the job.** Packaging the model and choosing the right hosting setup are as important as accuracy.

Future directions

If we extend the project, we'd consider:

- **Domain adaptation**: retrain or fine-tune on the specific domain of interest (e.g., product reviews, support tickets, finance).
- **Transformer models**: fine-tuned **BERT** or similar often improves accuracy, especially on nuanced text.
- **True three-class training**: include Neutral examples in training rather than deriving Neutral from confidence.
- **Live data connectors**: optional integration to fetch and analyze live tweets or news comments.

- **Analytics dashboard**: trends over time, filters by topic or source.

Conclusion

We set out to build a practical, explainable sentiment analyzer that regular people can use. By combining a trusted dataset, careful text processing, an interpretable model, and a simple web interface, we produced a system that is both **useful** and **teachable**. It shows how machine learning can move from a notebook to a real application that communicates clearly with its users.

If you're curious to explore more, try different types of text and watch how the words you use change the model's confidence and its choice of label. That interplay—between your language and the model's learned signals—is the heart of modern NLP.