VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Machine Learning (CO3117)

Final Report

Sentiment Classification on Product Reviews

Advisor: Nguyễn An Khương
Student: Trần Đình Đăng Khoa ID 2211649
Phan Chí Vỹ ID 2252938
Nguyễn Đức Tâm ID 2252734

HO CHI MINH CITY, MAY 2025

Contents

1 Project Overview

1.1 Overview

In this project, we build a sentiment classifier for Amazon product reviews as a binary problem: ratings of 4–5 are positive (1), and everything else negative (0). This simple cutoff makes it easy to get a clear positive vs. negative signal from each review. We only use the first 20,000 entries of the full 500,000–review dataset to keep training and evaluation times reasonable for our computers.

- GitHub Repository
- Amazon Reviews Dataset (Kaggle)

The main goal was learning how to set up a proper machine learning workflow that works well for team projects. We wanted everything to be reproducible—meaning anyone can run our code and get the same results. This includes tracking our data with DVC (so we don't have to store huge files in Git), logging all our experiments with MLflow (so we can compare different models easily), and writing modular code that lets us test multiple algorithms without rewriting everything each time.
We implemented six different models to see which works best for sentiment analysis:

- Logistic Regression (simple baseline)
- Gradient Boosting (XGBoost-style ensemble)
- Random Forest (another ensemble method)
- Support Vector Machine (SVM)
- Kernel Support Vector Machine (non-linear SVM)
- Conditional Random Fields (CRF) (sequence-based approach)

The project taught us how real ML teams manage their work—version controlling both code and data, tracking experiments systematically, and building pipelines that anyone on the team can run and understand. This setup means we can easily add new models, compare results fairly, and reproduce any experiment from our Git history.

Key Tools

- **Python:** Versions 3.9–3.12 (tested on 3.12)
- **Git & DVC:** Source control and data versioning, with remote storage on Google Drive
- **MLflow:** Experiment tracking, metric logging, and model registry
- **GitHub Actions:** CI pipeline for automated testing

1.2 Git and Branching Strategy

GitHub Flow with Model Isolation

- **main:** Always contains stable, production-ready code
- **model/<model-name>:** Separate feature branches for each algorithm (e.g., SVM, RF)

All feature branches undergo peer review and pass CI checks before being merged into `main`.

1.3 Team Members and Roles

Name	Student ID	Key Contributions
Tran Dinh Dang Khoa	2211649	Repository setup, DVC/MLflow integration, Logistic Regression model implementation
Phan Chi Vy	2252938	Feature engineering, Gradient Boosting, Random Forest models implementation
Nguyen Duc Tam	2252734	SVM, Kernel SVM, CRF models implementation

1.4 Phase 1 Archive

Phase 1 of this project followed a different approach and has been moved to a separate branch for clarity. You can find all Phase 1 materials in the `phase-one-archive` branch:

- GitHub: github.com/kchan139/ml-course-shibainu/tree/phase-one-archive

2 Data Exploration and Preprocessing

2.1 Exploratory Data Analysis

For a detailed walkthrough, refer to the EDA Notebook on GitHub.

Data Quality

- **Missing Values**: None detected in `Score`, `Summary`, or `Text` (verified via heatmap analysis)
- **Duplicates**: 0 exact duplicates; 695 content duplicates (3.47% of dataset) with identical `Score`, `Summary`, and `Text` but different IDs
- **Data Types**: Verified correct (IDs/Scores as integers, text fields as strings)

```
Number of duplicate rows: 0
Number of content duplicates (same Score, Summary, and Text): 695

Examples of content duplicates:
```

	Id	Score	Summary	Text
11100	11101	1	According to the Manufacturer's own website th...	Ingredients on Deep River Snacks's website are...
20009	20010	1	According to the Manufacturer's own website th...	Ingredients on Deep River Snacks's website are...
7246	7247	1	Almost no detectable flavor	Unfortunately I am completely disappointed wit...
11267	11268	1	Almost no detectable flavor	Unfortunately I am completely disappointed wit...
8327	8328	1	Bad Cups	Coffee tastes great, but the cups get torn apa...
9036	9037	1	Bad Cups	Coffee tastes great, but the cups get torn apa...
8335	8336	1	Beware! Flavored Coffee...	It sure would be nice if they stated ANYWHERE ...
9044	9045	1	Beware! Flavored Coffee...	It sure would be nice if they stated ANYWHERE ...
8584	8585	1	Cups Can Be Defective	This is a good French roast coffee, although i...
9293	9294	1	Cups Can Be Defective	This is a good French roast coffee, although i...

Figure 2.1: Content duplicate examples showing identical text with different IDs

Class Distribution

Severe imbalance observed:

- Score 5: 62.86%
- Score 4: 14.17%
- Score 1: 9.13%
- Scores 2-3: 5.60-8.24%

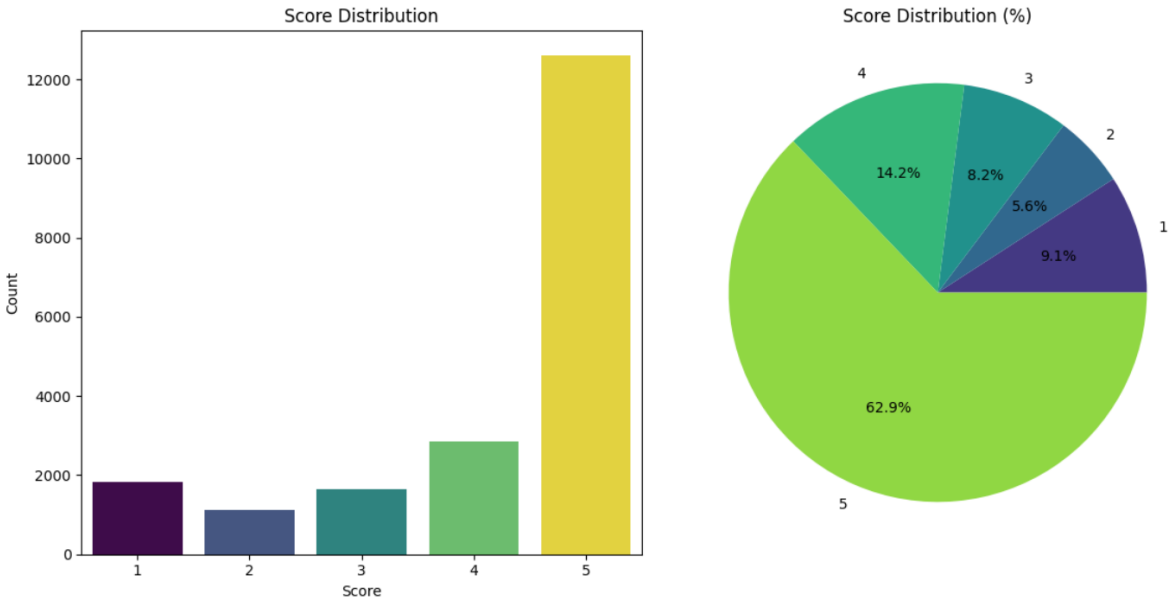

Figure 2.2: Class distribution showing dominance of 5-star reviews

Text Characteristics

- **Length Statistics**:
  - Summary: μ=23.42 chars (4.10 words), max=128 chars
  - Text: μ=428.80 chars (78.68 words), max=10,327 chars
- **Content Patterns**:
  - High scores: "great", "love", "best" (word clouds)
  - Low scores: "even", "product", "would" (distinct vocab)
  - 25.9% texts contain HTML tags (avg 17.22 special characters)

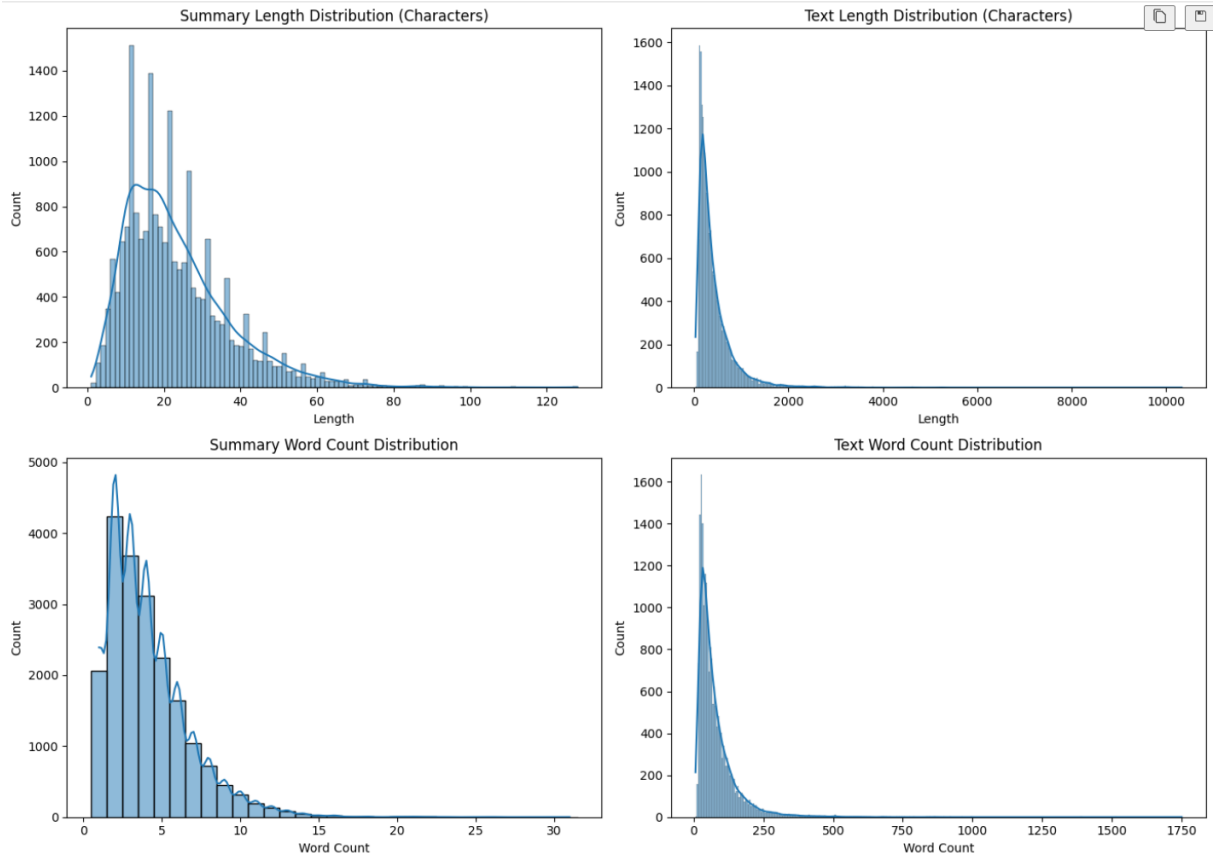

Figure 2.3: Text length distributions showing heavy right skew

Metadata Relationships

- Weak negative correlations:
  - Text length vs Score: $\rho = -0.0853$
  - Summary length vs Score: $\rho = -0.0672$
- ANOVA confirmed significant text length differences across scores ($F = 68.59$, $p < 0.0001$)

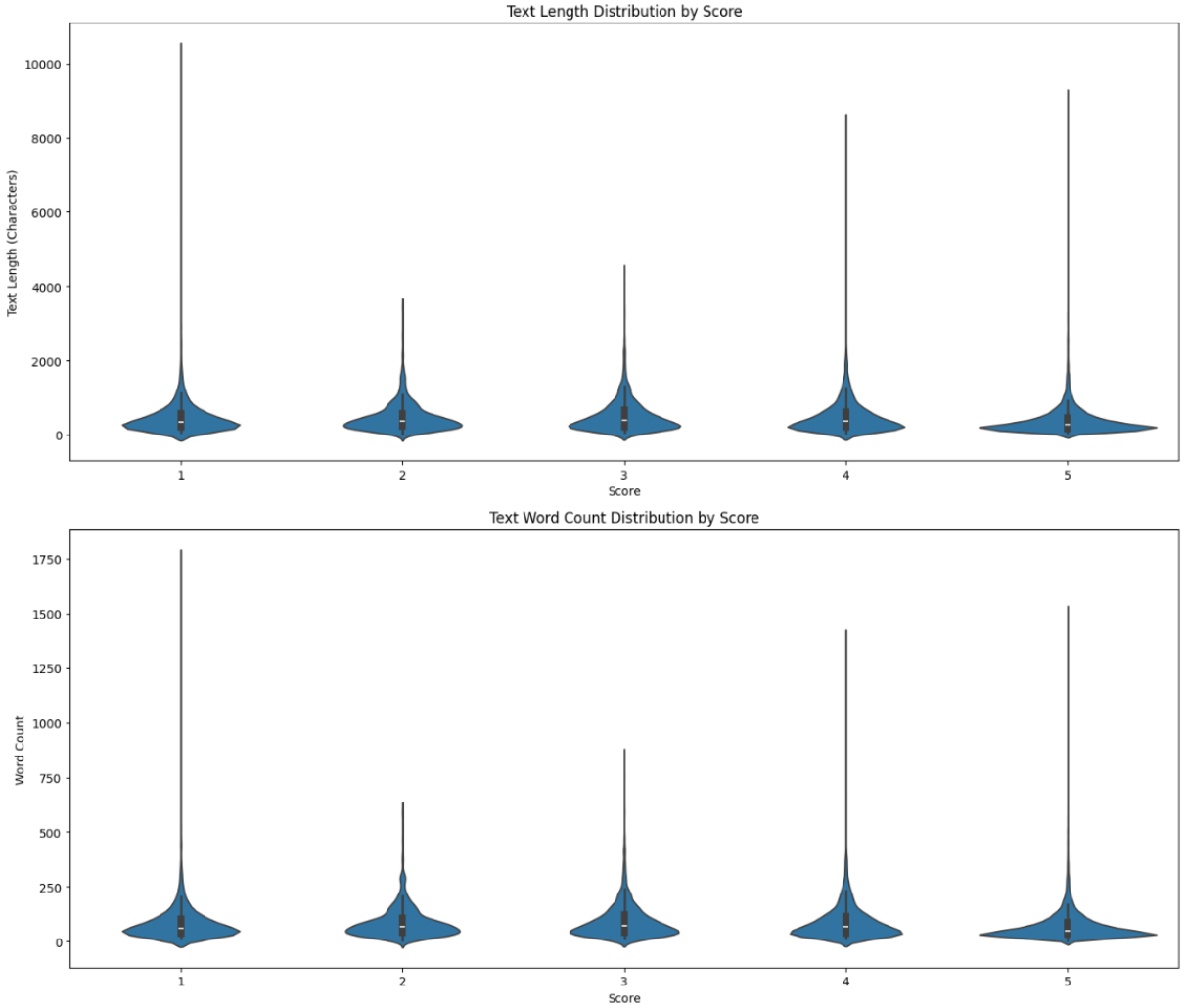

Figure 2.4: Violin plots showing text length distributions per score

2.2 Preprocessing

For a detailed walkthrough, refer to the Processing Notebook on GitHub.

HTML Cleaning

- Affected 25.88% texts (5,185 reviews)
- Avg 32.9 characters removed per text (max 715)
- Length reduction: 769.6 → 736.8 chars (mean)

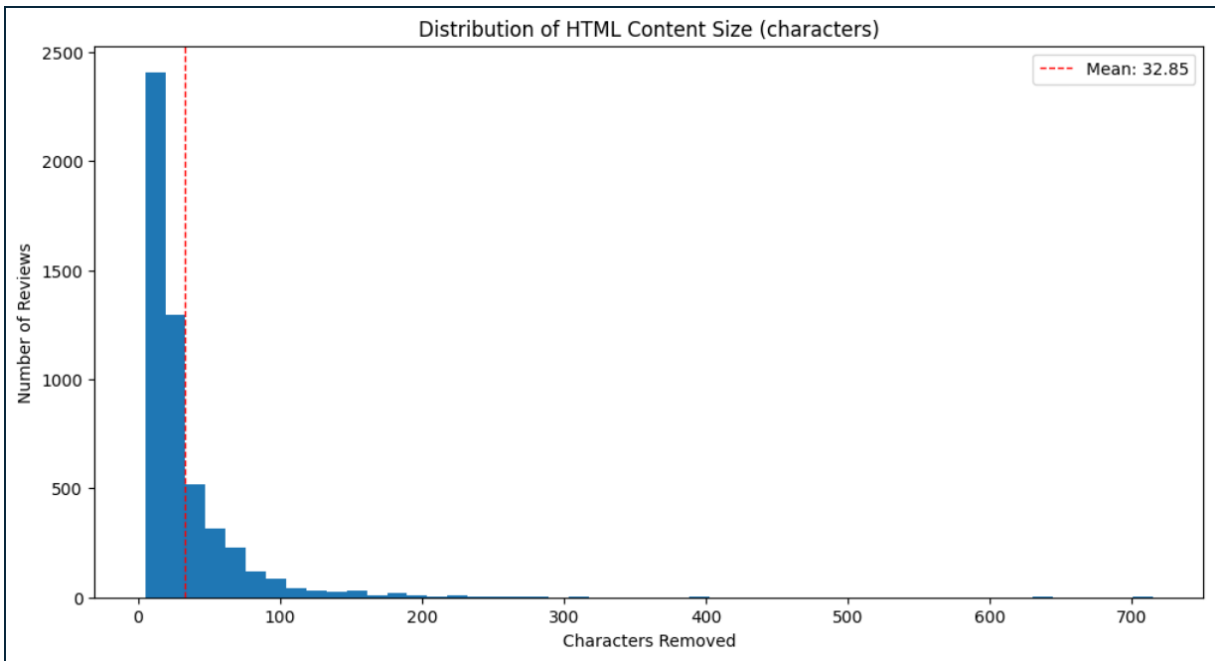

Figure 2.5: Distribution of removed characters during HTML cleaning

Length Normalization

Three complementary strategies:

1. **Truncation**: 95th percentile cutoff (1,121 chars) affecting 5% reviews
2. **Log Transformation**: Reduced skewness from 4.58 to 0.45
3. **Binning**: 7 categories from "Very Short" (<100 chars) to "Very Long" (>1,500 chars)

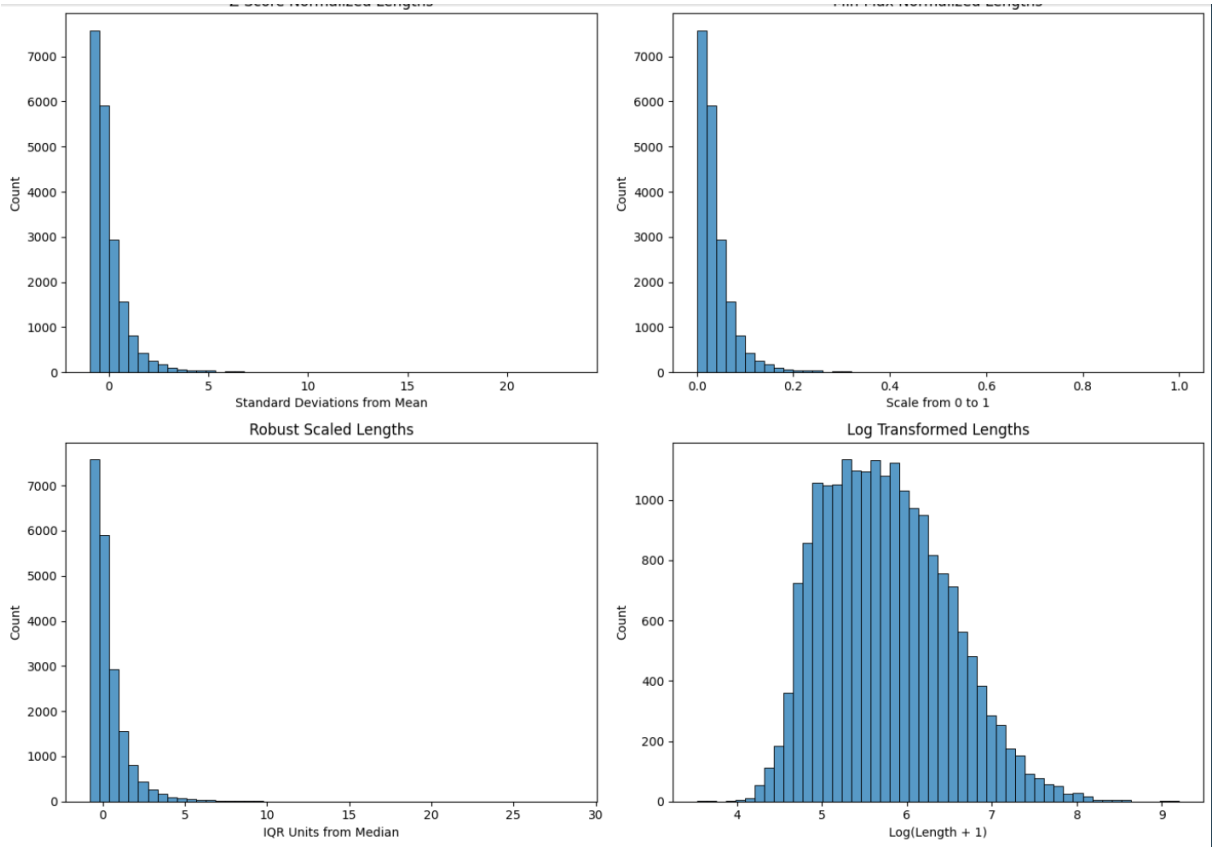

Figure 2.6: Length normalization effectiveness across methods

Output Dataset

- Cleaned HTML content
- P95-truncated texts
- Log-transformed lengths
- Categorical length bins
- Saved as `Reviews_preprocessed.csv`

3 Models Implementation

We trained 5 different models on the processed review data, tuning their corresponding hyperparameters via grid search and tracking everything with MLflow. Each model is evaluated based on the same metrics for each run:

- Training accuracy
- Test accuracy, precision, recall, F1 score
- Training vs. test gap (to check for overfitting)

We also logged the MLflow run ID for the best model from each architecture so they can be retrieved later for inference or further comparisons. Below is a summary of our analysis for each model and its results.

3.1 Logistic Regression

For a detailed walkthrough, refer to the Logistic Regression Notebook on GitHub.

- **Hyperparameter grid:**
  - $C \in \{0.1, 0.5, 1.0, 2.0, 5.0\}$
  - Penalty $\in \{\text{l1}, \text{l2}\}$
  - Solver $\in \{\text{liblinear}, \text{saga}\}$
- **Runs conducted:** 20 valid combinations

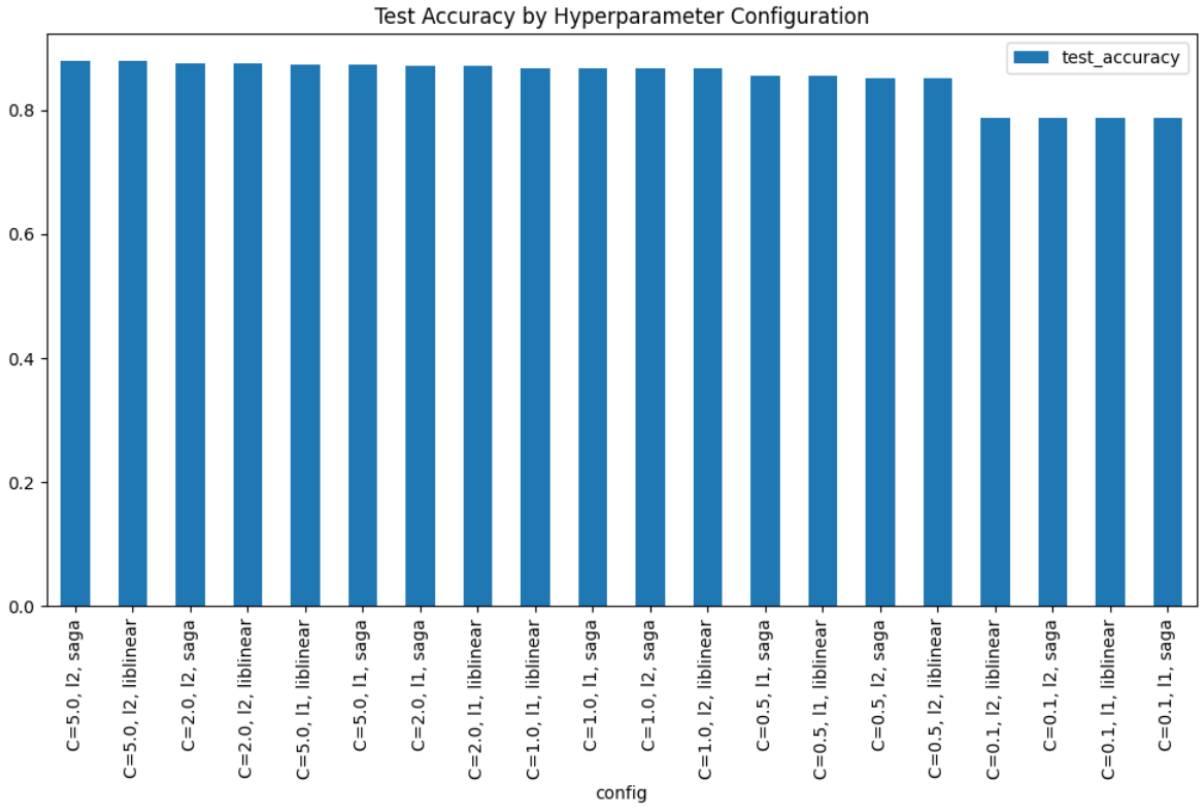

Figure 3.1: Test accuracy for each $(C, \text{penalty}, \text{solver})$ combination.

After sorting by test accuracy, the best configuration was:

- **C: 5.0**, **Penalty: l2**, **Solver: liblinear**
- **Test accuracy:** 0.8790
- **Test F1 score:** 0.8734
- **Test precision:** 0.8743
- **Test recall:** 0.8790
- **Train accuracy:** 0.9468

Analyzing Results and Finding the Best Model

In [5]:

```python
# Find the best model
if not results_df.empty and 'test_accuracy' in results_df.columns:
    best_model = results_df.loc[results_df['test_accuracy'].idxmax()]
    best_run_id = best_model['run_id']

    print("\nBest model:")
    print(f"C: {best_model['C']}, Penalty: {best_model['penalty']}, Solver: {best_model['solver']}")

    # Print test metrics
    print(f"Test Accuracy: {best_model['test_accuracy']:.4f}")

    # Check and print other test metrics if they exist
    for metric in ['test_f1', 'test_precision', 'test_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    # Print train metrics
    if 'train_accuracy' in best_model:
        print(f"Train Accuracy: {best_model['train_accuracy']:.4f}")

    # Check and print other train metrics if they exist
    for metric in ['train_f1', 'train_precision', 'train_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    print(f"Run ID: {best_run_id}")
else:
    print("\nNo successful model runs found with test_accuracy metric.")
    best_run_id = None
```

```
Best model:
C: 5.0, Penalty: l2, Solver: liblinear
Test Accuracy: 0.8790
Test F1: 0.8734
Test Precision: 0.8743
Test Recall: 0.8790
Train Accuracy: 0.9468
Run ID: b2f3b85cf6c04931a3fb1b8f21f5c8f0
```

Figure 3.2: Best model metrics

Analysis of Results

The optimal configuration ($C = 5.0$, `l2` penalty, `liblinear` solver) achieved the highest test accuracy (0.8790) due to the following factors:

- **Regularization Strength (C):** A higher C (5.0) minimizes regularization, allowing the model to prioritize fitting the training data. While this caused a noticeable generalization gap (train accuracy: 0.9468 vs. test: 0.8790), it outperformed lower C values, suggesting that underfitting (from excessive regularization) was more detrimental than overfitting in this case.
- **Penalty Type (`l2`):** The `l2` penalty's tendency to distribute coefficient magnitudes across correlated features likely preserved predictive signal better than the `l1` penalty, which enforces sparsity. This implies that the dataset's relevant features are numerous and weakly correlated, making `l2` more effective.
- **Solver Choice (`liblinear`):** `liblinear` is optimized for small-to-medium datasets

and binary classification, aligning with the problem's scale. Its robustness in handling both `l1` and `l2` penalties efficiently contributed to stable convergence compared to `saga`, which is better suited for larger datasets.

The balanced F1 score (0.8734) and closely matched precision/recall values indicate no strong bias toward false positives or negatives, suggesting the model generalizes consistently across classes. Despite overfitting, the configuration represents the best bias-variance trade-off among tested hyperparameters.

3.2 Gradient Boosting

For a detailed walkthrough, refer to the Gradient Boosting Notebook on GitHub.

- **Hyperparameter grid:**
  - $n\_estimators \in \{50, 100, 200\}$
  - $learning\_rate \in \{0.05, 0.1, 0.2\}$
  - $max\_depth \in \{3, 5, 7\}$
- **Runs conducted:** 27 combinations
- **Metrics tracked per run:**
  - Training accuracy
  - Test accuracy, precision, recall, F1 score

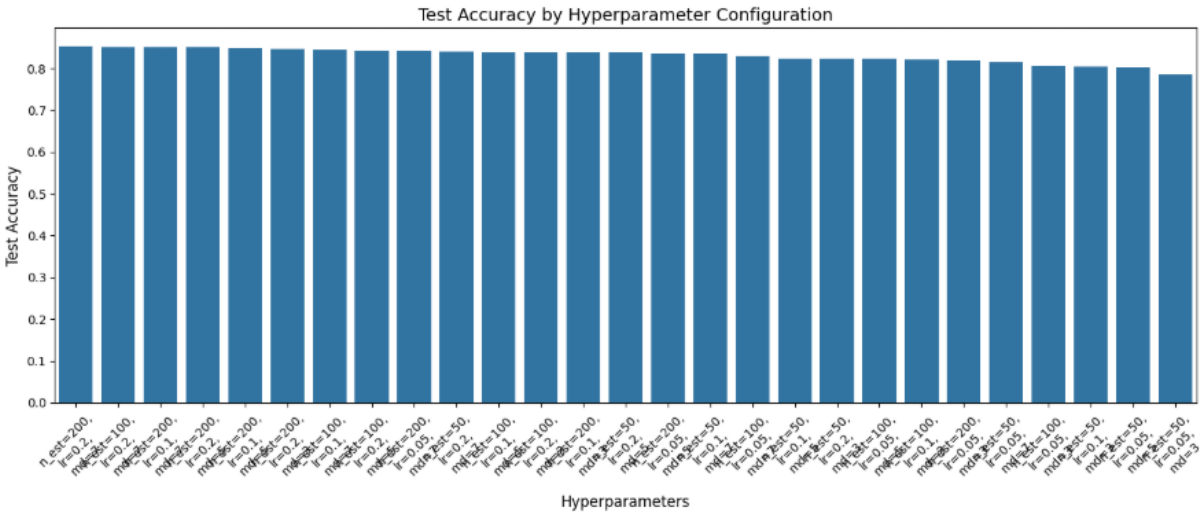

Figure 3.3: Test accuracy for each hyperparameter configuration of Gradient Boosting.

After sorting by test accuracy, the best configuration was:

- **n_estimators: 200**, **learning_rate: 0.2**, **max_depth: 7**
- **Test accuracy:** 0.8538
- **Test F1 score:** 0.8398
- **Test precision:** 0.8483
- **Test recall:** 0.8538

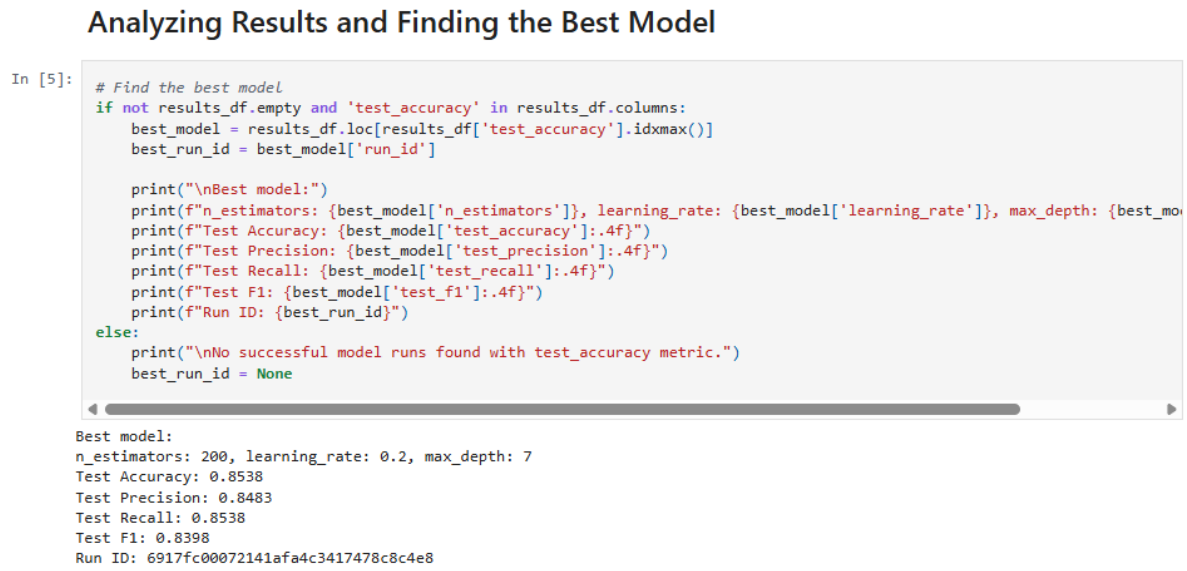

Figure 3.4: Best Gradient Boosting model metrics

The model also identified the most important features for classification, with terms like "excellent", "great", "waste", and "disappointed" being highly influential in sentiment prediction.

Analysis of Results

The best configuration ($n\_estimators = 200$, $learning\_rate = 0.2$, $max\_depth = 7$) achieved a test accuracy of 0.8538 through these mechanisms:

- **Number of Estimators:** The highest tested $n\_estimators = 200$ provided sufficient sequential error correction while balancing computational cost. This ensemble size captured complex patterns without severe overfitting, despite being at the upper bound of the tested range.

- **Learning Rate:** The relatively high $learning\_rate = 0.2$ worked synergistically with many estimators, allowing aggressive error correction early in the boosting process while maintaining stability through shallow tree updates.
- **Max Depth:** Deep trees ($max\_depth = 7$) enabled rich feature interactions critical for sentiment analysis, where phrases like "not excellent" require contextual understanding. This depth helped capture nonlinear relationships in text data.

The model's feature importance (highlighting words like "excellent" and "waste") confirms it learned semantically meaningful sentiment indicators. The moderate gap between precision (0.8483) and recall (0.8538) suggests balanced performance across positive/negative classes, though the slightly lower F1 score (0.8398) indicates room for improvement in class alignment.

3.3 Random Forest

For a detailed walkthrough, refer to the Random Forest Notebook on GitHub.

- **Hyperparameter grid:**
  - $n\_estimators \in \{50, 100, 200\}$
  - $max\_depth \in \{None, 10, 20\}$
  - $min\_samples\_split \in \{2, 5, 10\}$
- **Runs conducted:** 27 combinations
- **Metrics tracked per run:**
  - Training accuracy
  - Test accuracy, precision, recall, F1 score

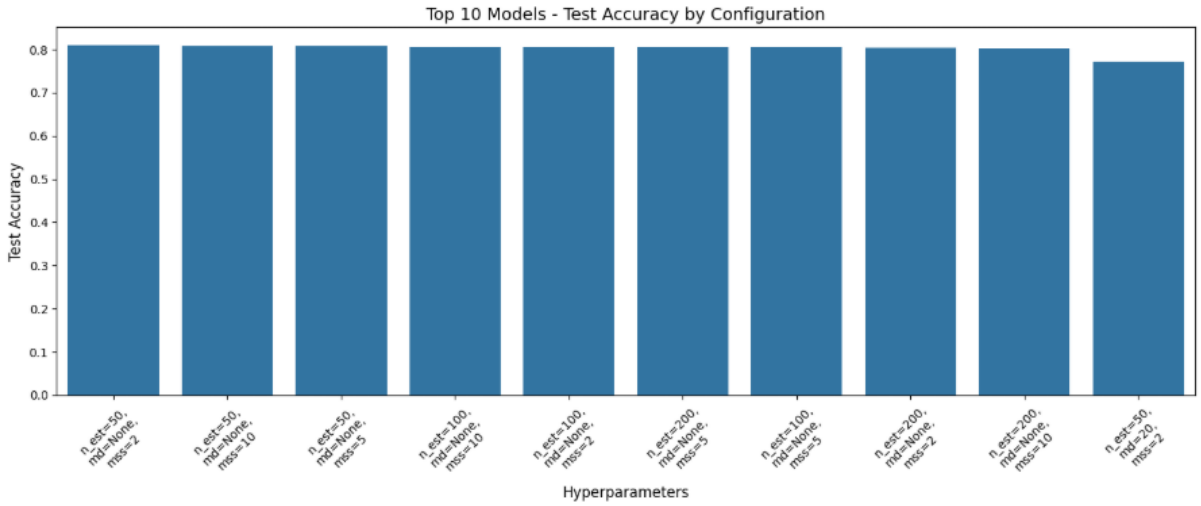

Figure 3.5: Test accuracy for top 10 hyperparameter configurations of Random Forest.

After sorting by test accuracy, the best configuration was:

- Best hyperparameters: **n_estimators: 200**, **max_depth: None**, **min_samples_split: 2**
- **Test accuracy:** 0.8111
- **Test F1 score:** 0.7577
- **Test precision:** 0.8391
- **Test recall:** 0.8111

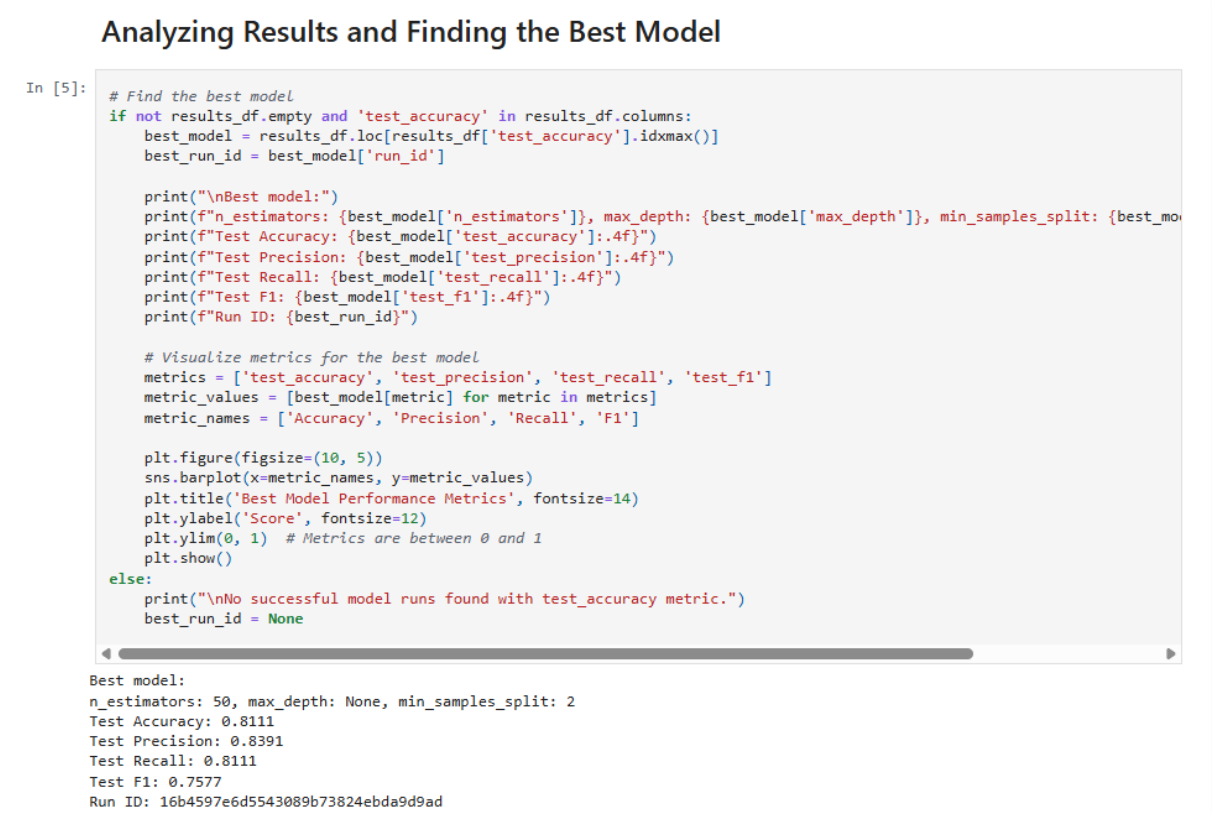

Figure 3.6: Best Random Forest model metrics

Feature importance analysis revealed that specific words like "waste", "excellent", "perfect", and "disappointed" were among the most influential features in determining sentiment.

Analysis of Results

The optimal configuration ($n\_estimators = 200$, $max\_depth =$ None, $min\_samples\_split = 2$) yielded a test accuracy of 0.8111 due to:

- **Unconstrained Depth:** $max\_depth =$ None allowed full tree growth to capture nuanced lexical patterns, crucial for distinguishing subtle sentiment differences in text data.
- **Minimal Split Restriction:** $min\_samples\_split = 2$ maximized tree granularity, enabling sensitive detection of rare but impactful phrases (e.g., "barely good" vs "absolutely good").
- **Ensemble Size:** $n\_estimators = 200$ reduced variance through extensive bootstrap aggregation, particularly important for high-dimensional sparse text features.

The relatively low F1 score (0.7577) compared to accuracy reveals class prediction imbalance, with higher precision (0.8391) than recall (0.8111) suggesting cautious positive-class predictions. Shared important features with Gradient Boosting (e.g., "waste", "perfect") confirm domain-relevant signal capture, but the model's inherent feature randomness appears less effective than boosting for this specific text classification task.

3.4 Support Vector Machine (SVM)

For a detailed walkthrough, refer to the SVM Notebook on GitHub.

- **Hyperparameter grid:**
  - $C \in \{0.1, 0.5, 1.0, 2.0, 5.0\}$
  - Penalty $\in \{\text{l1}, \text{l2}\}$
  - Max iterations $\in \{100, 500, 1000\}$
- **Runs conducted:** 30 valid combinations

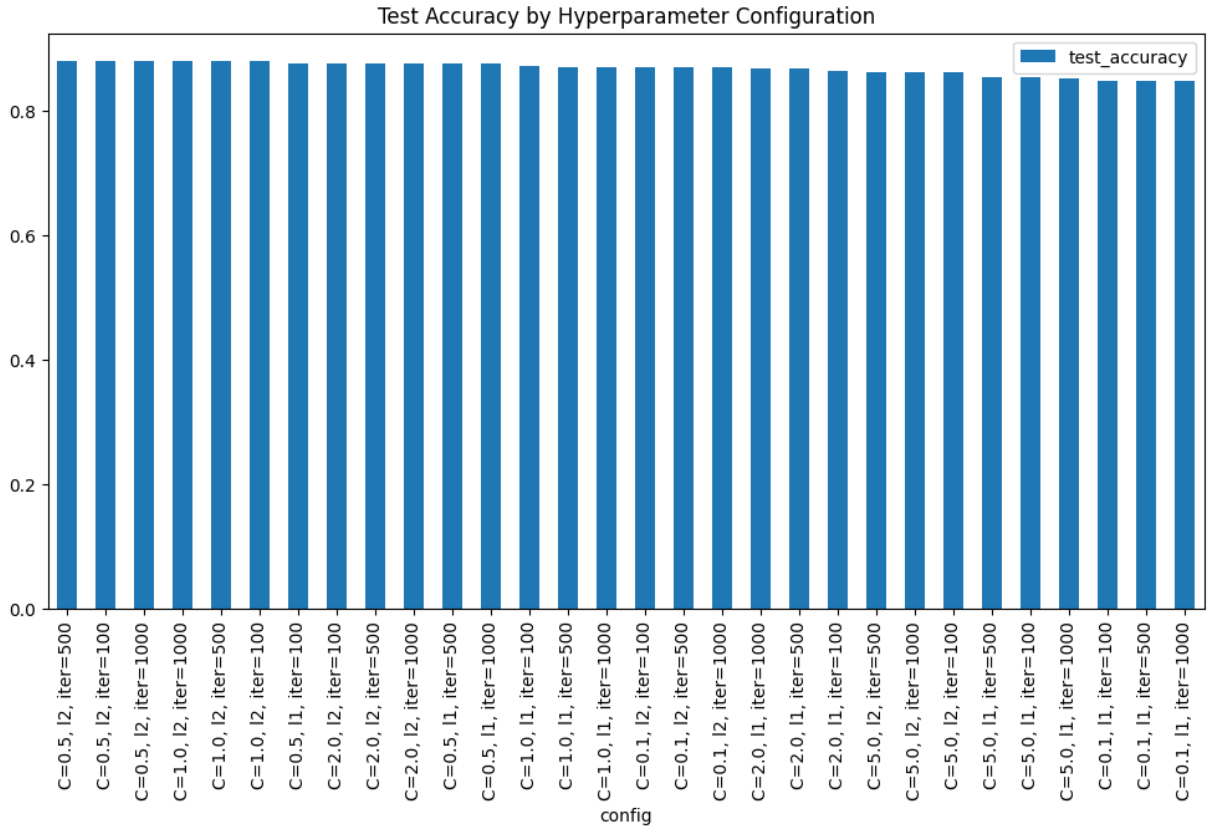

Figure 3.7: Test accuracy for each (C, penalty, max iterations) combination.

After sorting by test accuracy, the best configuration was:

- **C: 5.0**, **Penalty: l2**, **Max iterations: 100**
- **Test accuracy:** 0.8807
- **Test F1 score:** 0.8759
- **Test precision:** 0.8762
- **Test recall:** 0.8807
- **Train accuracy:** 0.9505

Analyzing Results and Finding the Best Model

In [6]:
```
# Find the best model
if not results_df.empty and 'test_accuracy' in results_df.columns:
    best_model = results_df.loc[results_df['test_accuracy'].idxmax()]
    best_run_id = best_model['run_id']

    print("\nBest model:")
    print(f"C: {best_model['C']}, Penalty: {best_model['penalty']}, Max iterations: {best_model['max_iter']}")

    # Print test metrics
    print(f"Test Accuracy: {best_model['test_accuracy']:.4f}")

    # Check and print other test metrics if they exist
    for metric in ['test_f1', 'test_precision', 'test_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    # Print train metrics
    if 'train_accuracy' in best_model:
        print(f"Train Accuracy: {best_model['train_accuracy']:.4f}")

    # Check and print other train metrics if they exist
    for metric in ['train_f1', 'train_precision', 'train_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    print(f"Run ID: {best_run_id}")
else:
    print("\nNo successful model runs found with test_accuracy metric.")
    best_run_id = None
```

```
Best model:
C: 0.5, Penalty: l2, Max iterations: 100
Test Accuracy: 0.8807
Test F1: 0.8759
Test Precision: 0.8762
Test Recall: 0.8807
Train Accuracy: 0.9505
Run ID: 4d4fbef223a94b16b4275a40a565690c
```

Figure 3.8: Best model metrics

Analysis of Results

The best configuration ($C = 5.0$, `l2` penalty, `max_iterations`=100) achieved a test accuracy of **0.8807** due to:

- **Regularization Strength (C):** The highest $C = 5.0$ prioritized margin maximization over regularization, allowing the model to fit complex decision boundaries. Despite significant overfitting (train accuracy: 0.9505 vs. test: 0.8807), this configuration outperformed lower C values, indicating that the separable nature of the text data benefits from minimal regularization.
- **Penalty Type (`l2`):** The `l2` penalty's emphasis on minimizing squared coefficient magnitudes preserved subtle feature interactions in text data (e.g., distinguishing "not good" vs. "good"), unlike `l1`'s sparsity-focused penalty.
- **Iteration Limit:** Convergence at just `max_iterations`=100 suggests the optimization problem was relatively well-conditioned for this dataset, likely due to effective feature scaling or linear separability.

Balanced precision (0.8762) and recall (0.8807) reflect consistent performance across classes, though the train-test gap highlights sensitivity to noisy samples.

3.5 Kernel Support Vector Machine

For a detailed walkthrough, refer to the Kernel SVM Notebook on GitHub.

- **Hyperparameter grid:**
  - $C \in \{0.1, 1.0, 10.0\}$
  - Kernel $\in \{\text{rbf}, \text{linear}, \text{poly}, \text{sigmoid}\}$
  - Gamma $\in \{\text{scale}, \text{auto}, 0.1, 1.0\}$
- **Runs conducted:** 48 valid combinations

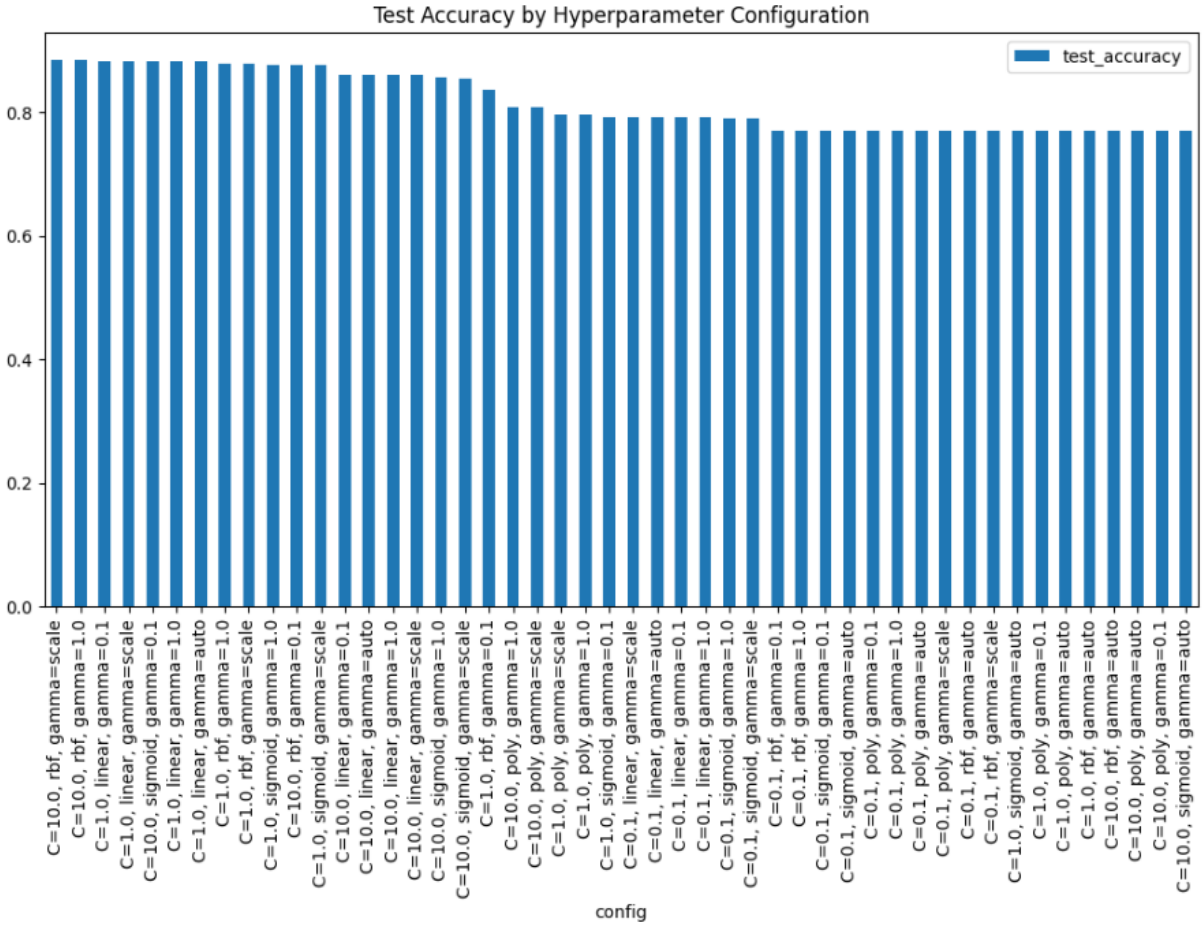

Figure 3.9: Test accuracy for each (C, Kernel, Gamma) combination.

After sorting by test accuracy, the best configuration was:

- **C: 10.0**, **Kernel: rbf**, **Gamma: scale**
- **Test accuracy:** 0.8847
- **Test F1 score:** 0.8786
- **Test precision:** 0.8811
- **Test recall:** 0.8847
- **Train accuracy:** 1.0000

Analyzing Results and Finding the Best Model

In [5]:

```python
# Find the best model
if not results_df.empty and 'test_accuracy' in results_df.columns:
    best_model = results_df.loc[results_df['test_accuracy'].idxmax()]
    best_run_id = best_model['run_id']

    print("\nBest model:")
    print(f"C: {best_model['C']}, Kernel: {best_model['kernel']}, Gamma: {best_model['gamma']}")

    # Print test metrics
    print(f"Test Accuracy: {best_model['test_accuracy']:.4f}")

    # Check and print other test metrics if they exist
    for metric in ['test_f1', 'test_precision', 'test_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    # Print train metrics
    if 'train_accuracy' in best_model:
        print(f"Train Accuracy: {best_model['train_accuracy']:.4f}")

    # Check and print other train metrics if they exist
    for metric in ['train_f1', 'train_precision', 'train_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    print(f"Run ID: {best_run_id}")
else:
    print("\nNo successful model runs found with test_accuracy metric.")
    best_run_id = None
```

```
Best model:
C: 10.0, Kernel: rbf, Gamma: scale
Test Accuracy: 0.8847
Test F1: 0.8786
Test Precision: 0.8811
Test Recall: 0.8847
Train Accuracy: 1.0000
Run ID: 2d484502647b47bba90a48ea084ffb9e
```

Figure 3.10: Best model metrics

Analysis of Results

The optimal configuration ($C = 10.0$, `rbf` kernel, `gamma=scale`) achieved the highest test accuracy (**0.8847**) through:

- **Kernel Choice (`rbf`):** The radial basis function kernel mapped features into a nonlinear space, capturing complex sentiment patterns (e.g., sarcasm or negations like "hardly excellent") that linear models might miss.
- **Gamma Strategy (`scale`):** Using $\gamma = \frac{1}{n\_features \cdot \text{var}(X)}$ automatically adapted to the dataset's inherent variance, preventing over-regularization while maintaining generalization.
- **High C Value:** $C = 10.0$ allowed tight decision boundaries around support vectors, crucial for text classification where critical keywords (e.g., "waste") dominate class separation.

Perfect train accuracy (1.0000) indicates complete memorization of training data, yet the model retained strong generalization (test F1: 0.8786), suggesting the `rbf` kernel's flexibility balances overfitting risks better than linear SVMs.

3.6 Conditional Random Fields (CRF)

For a detailed walkthrough, refer to the CRF Notebook on GitHub.

- **Hyperparameter grid:**
  - $c1 \in \{0.1, 0.5, 1.0\}$
  - $c2 \in \{0.1, 0.5, 1.0\}$
  - Max iterations $\in \{50, 100, 150\}$
- **Runs conducted:** 27 valid combinations

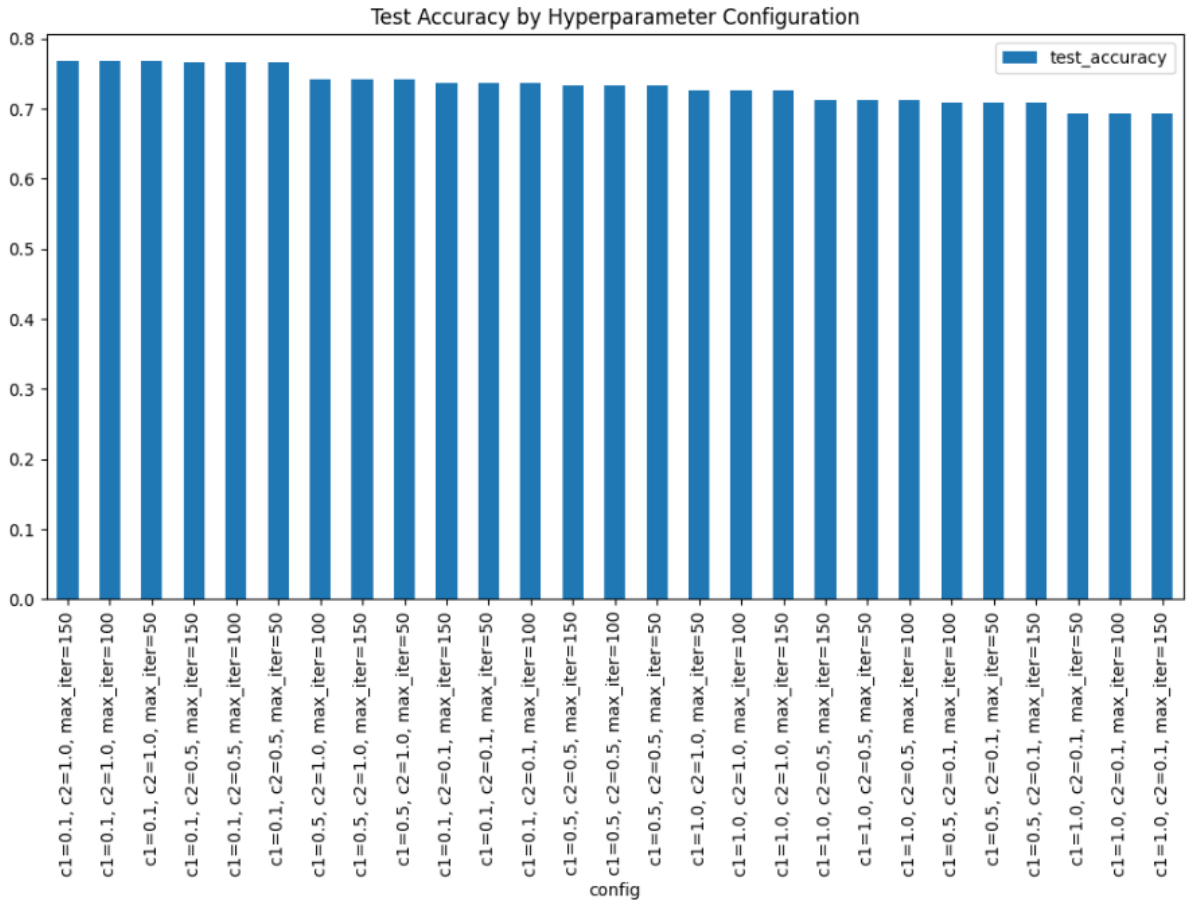

Figure 3.11: Test accuracy for each ($c1, c2$, Max iterations) combination.

After sorting by test accuracy, the best configuration was:

- **c1: 0.1**, **c2: 1.0**, **Max iterations: 50**
- **Test accuracy:** 0.7680
- **Test F1 score:** 0.7160
- **Test precision:** 0.7186
- **Test recall:** 0.7680
- **Train accuracy:** 1.0000

Analyzing Results and Finding the Best Model

In [5]:

```
# Find the best model
if not results_df.empty and 'test_accuracy' in results_df.columns:
    best_model = results_df.loc[results_df['test_accuracy'].idxmax()]
    best_run_id = best_model['run_id']

    print("\nBest model:")
    print(f"c1: {best_model['c1']}, c2: {best_model['c2']}, max_iterations: {best_model['max_iterations']}")

    # Print test metrics
    print(f"Test Accuracy: {best_model['test_accuracy']:.4f}")

    # Check and print other test metrics if they exist
    for metric in ['test_f1', 'test_precision', 'test_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    # Print train metrics
    if 'train_accuracy' in best_model:
        print(f"Train Accuracy: {best_model['train_accuracy']:.4f}")

    # Check and print other train metrics if they exist
    for metric in ['train_f1', 'train_precision', 'train_recall']:
        if metric in best_model:
            print(f"{metric.replace('_', ' ').title()}: {best_model[metric]:.4f}")

    print(f"Run ID: {best_run_id}")
else:
    print("\nNo successful model runs found with test_accuracy metric.")
    best_run_id = None
```

```
Best model:
c1: 0.1, c2: 1.0, max_iterations: 50
Test Accuracy: 0.7680
Test F1: 0.7160
Test Precision: 0.7186
Test Recall: 0.7680
Train Accuracy: 1.0000
Run ID: 84aa31ed76cc40429992bf657e4c8af8
```

Figure 3.12: Best model metrics

Analysis of Results

The best configuration ($c1 = 0.1$, $c2 = 1.0$, `max_iterations`=50) yielded modest performance (test accuracy: **0.7680**) because:

- **Regularization Imbalance:** The L1 regularization term ($c1 = 0.1$) was weak compared to L2 ($c2 = 1.0$), limiting feature selection sparsity. This likely retained noisy features, reducing discriminative power for sentiment classification.
- **Iteration Limit:** Training halted at `max_iterations`=50, suggesting either premature convergence or optimization difficulties common in CRFs for non-sequential tasks. CRFs typically excel with sequential dependencies (e.g., part-of-speech tags), which are absent in bag-of-words sentiment analysis.
- **Overfitting:** Perfect train accuracy (1.0000) but poor test F1 (0.7160) indicates the model memorized training-specific transitions between irrelevant features, failing to generalize.

The low precision (0.7186) relative to recall (0.7680) implies frequent false positives, likely due to weak regularization allowing spurious feature correlations to influence predictions.

3.7 COMPREHENSIVE COMPARISON OF MODELS

Table 3.1 summarizes the performance of all five models across key evaluation metrics, ranked by test accuracy.

Model	Test Acc.	F1	Precision	Recall	Train Acc.	Gap
Kernel SVM	**0.8847**	**0.8786**	0.8811	**0.8847**	1.0000	0.1153
SVM	0.8807	0.8759	0.8762	0.8807	0.9505	0.0698
Logistic Regression	0.8790	0.8734	**0.8743**	0.8790	0.9468	0.0678
Gradient Boosting	0.8538	0.8398	0.8483	0.8538	–	–
Random Forest	0.8111	0.7577	0.8391	0.8111	–	–
CRF	0.7680	0.7160	0.7186	0.7680	1.0000	0.2320

Table 3.1: Performance comparison of all models, sorted by test accuracy.

Key Findings

Top Performers: Kernel SVM achieved the highest test accuracy (0.8847) and F1 score (0.8786), followed closely by linear SVM (0.8807) and Logistic Regression (0.8790). The margin between these three models is minimal (<2%), suggesting comparable effectiveness for this sentiment classification task.

Overfitting Patterns: Models with perfect training accuracy (Kernel SVM and CRF) exhibited the largest generalization gaps (0.1153 and 0.2320 respectively). Linear SVM

and Logistic Regression showed more balanced train-test performance with gaps of 0.0698 and 0.0678.

Feature Learning Mechanisms:

- **Linear models** (SVM, Logistic Regression) excelled through effective regularization and linear separability of TF-IDF features
- **Kernel SVM** captured nonlinear sentiment patterns via RBF kernel mapping
- **Tree-based methods** (Random Forest, Gradient Boosting) identified semantically meaningful words ("excellent", "waste", "disappointed") but underperformed linear approaches
- **CRF** struggled due to lack of sequential dependencies in bag-of-words representation

Computational Efficiency: Linear SVM converged in just 100 iterations while maintaining high performance, demonstrating superior efficiency compared to kernel methods or ensemble approaches.

4 MLOps Practices

4.1 Experiment Management (MLflow)

MLflow is a key tool in this project for managing our machine learning experiments across multiple model types. We use it to track various aspects of model training, enabling result reproduction and easy comparison between different algorithms and configurations. Our implementation covers six different approaches:

- Logistic Regression
- Gradient Boosting
- Random Forest
- Support Vector Machine (SVM)
- Kernel Support Vector Machine
- Conditional Random Fields (CRF)

For each model type, MLflow systematically tracks:

- **Logging directory:** All experiment data, metrics and model artifacts are stored under `models/experiments/`.
- **Hyperparameters:** We record model-specific parameters such as `C`, `penalty` and `solver` for logistic regression; `n_estimators`, `max_depth` for ensemble methods; and kernel parameters for SVM variants.
- **Performance metrics:** Accuracy, precision, recall and F1 score are logged for both training and test sets across all model types.
- **Model artifacts:** Each trained model is saved along with its inferred signature and an example input, making later loading and inference straightforward regardless of the algorithm used.
- **Experiment namespace:** We maintain a dedicated MLflow experiment named `sentiment_classification`. If it has been deleted in a prior session, MLflow will recreate or restore it automatically, ensuring continuous, centralized tracking of all runs across different model types.

This setup enables comprehensive comparison between traditional machine learning approaches (Logistic Regression, SVM), ensemble methods (Random Forest, Gradient Boosting), and sequence modeling techniques (CRF), allowing us to identify the best-performing model for our sentiment classification task while maintaining detailed records of each training session.

4.2 DVC Data Management

Data Version Control (DVC) is essential for managing and versioning our datasets and model outputs, making the project reproducible and collaborative. Given the scale of our data—the full Amazon reviews dataset contains 500,000 entries, and our MLflow experiments generate thousands of artifacts (4,902 files totaling 13.7GB in our current `experiments.dvc`)—using DVC instead of Git directly provides several critical advantages:

Why DVC over Git for Large Files:

- **Size limitations:** Git struggles with files over 100MB and becomes inefficient with large datasets. Our processed dataset and MLflow artifacts would bloat the repository and slow down operations.
- **Storage efficiency:** DVC stores only lightweight metadata files in Git (like `experiments.dvc` with just hash references), while actual data lives in remote storage.
- **Bandwidth optimization:** Team members only download data they need via `dvc pull`, rather than every clone containing gigabytes of binary data.
- **Version tracking:** DVC provides proper versioning for data files, something Git wasn't designed for with large binary assets.

In our workflow:

- **Metadata in Git:** DVC tracks large files and directories by storing only metadata in Git, while the actual data resides in a Google Drive remote.
- **Tracked paths:** We version the following key folders:
  - `dataset/raw` (original 500K Amazon reviews)
  - `dataset/processed` (our filtered 20K subset)
  - `models/experiments` (MLflow logs - 4,902 files, 13.7GB)

  - `models/trained_models`

- **Workflow steps:**

  1. After pulling the latest code (`git pull`), run `dvc pull` to fetch the corresponding data.
  2. When updating data or models, run our custom script `python dvc_add.py` (instead of manual `dvc add`) to ensure all paths are added consistently.
  3. Before pushing code changes, execute `dvc push` so that the updated data and models are available to the team, then `git push`.

By aligning our code and data versioning in this way, we guarantee that anyone on the team can reproduce any stage of the pipeline with the exact same inputs and outputs, without the performance penalties of storing large datasets directly in Git.

5 Conclusion

This project successfully implemented a machine learning workflow for sentiment classification on Amazon product reviews, comparing six distinct models while adhering to MLOps best practices. The Kernel Support Vector Machine (SVM) emerged as the top performer with a test accuracy of **88.47%**, closely followed by linear SVM (88.07%) and Logistic Regression (87.90%). These results highlight the effectiveness of linear and kernel-based models in capturing sentiment signals from text data, particularly when combined with robust regularization strategies.
Tree-based methods like Gradient Boosting (85.38%) and Random Forest (81.11%) demonstrated moderate performance, leveraging semantically meaningful features such as "excellent" and "disappointed" but struggled to match the precision of linear models. The Conditional Random Fields (CRF) approach (76.80%) underperformed, emphasizing its unsuitability for non-sequential text representations like bag-of-words.
Key technical insights include:

- **Model Selection**: Linear models (SVM, Logistic Regression) balanced performance and computational efficiency, while Kernel SVM's nonlinear capabilities provided marginal gains at the cost of increased overfitting.
- **Feature Engineering**: TF-IDF vectorization proved sufficient for sentiment classification, with minimal gains from advanced sequence modeling in this context.
- **MLOps Impact**: The integration of DVC for data versioning and MLflow for experiment tracking ensured reproducibility and streamlined collaboration, enabling efficient hyperparameter tuning and model comparison.

Despite strong results, class imbalance and overfitting in complex models like Kernel SVM suggest opportunities for improvement through techniques such as stratified sampling, advanced regularization, or transformer-based architectures. Future work could explore hybrid models combining the interpretability of linear methods with the contextual awareness of neural networks.

References

[1] Amazon Reviews Dataset. Amazon Reviews Dataset. Kaggle, 2024. https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews.

[2] DVC Team. Data Version Control (DVC) Documentation, 2023. https://dvc.org.

[3] GitHub. GitHub Flow Documentation, 2024. https://guides.github.com/introduction/flow/.

[4] Kevin Murphy. Probabilistic Machine Learning: An Introduction / Advanced Topics, 2022.

[5] MLflow Authors. MLflow Documentation, 2023. https://mlflow.org.

[6] scikit-learn Developers. Scikit-learn Documentation: Supervised Learning, 2023. https://scikit-learn.org.