🚀 From Experiments to Automation: Building an End-to-End MLOps PipelineMLflow + Optuna + Prefect for Sentiment Analysis

A Complete MLOps Journey: From Experimentation to Production

Today was all about transforming a machine learning notebook into a real MLOps system. I worked on an end-to-end Sentiment Analysis project using Flipkart product reviews, covering the complete lifecycle: from experimentation and model tuning to automation and scheduling.

This blog captures the entire workflow, tools used, complete source code, screenshots of key milestones, and lessons learned along the way.

🎯 Problem Statement

Given Flipkart product reviews and ratings, the objective was to:

- Perform binary sentiment classification (Positive / Negative)
- Optimize models using F1-score instead of accuracy
- Track experiments, parameters, metrics, and models systematically
- Automate the training pipeline with scheduling
- Build a reproducible and production-ready ML system

🧹 Data Preparation & NLP Pipeline

Dataset Overview

The dataset contained product reviews with the following key columns:

- Review text (customer feedback)
- Ratings (1-5 scale)

Sentiment Mapping

To convert ratings into binary sentiment labels:

- Ratings 1-2 → Negative (0)
- Ratings 3-5 → Positive (1)

Text Preprocessing Pipeline

A robust NLP cleaning pipeline was implemented to ensure high-quality inputs:

- Lowercasing all text
- Removing emojis
- Removing URLs and HTML tags
- Removing punctuation and numbers
- Stopword removal (while preserving negations like 'not', 'no')
- Stemming using Porter Stemmer

This preprocessing step significantly improved model stability and generalization by reducing noise in the text data.

🔍 Feature Engineering

Used TF-IDF (Term Frequency-Inverse Document Frequency) Vectorization to convert text into numerical features. Key parameters tuned:

- ngram_range - to capture word combinations
- max_features - limiting vocabulary size
- min_df and max_df - filtering rare and common terms

TF-IDF proved effective for capturing the contextual importance of words in product reviews.

🧠 Model Selection & Evaluation

Multiple classical machine learning models were evaluated:

- Logistic Regression
- Multinomial Naive Bayes
- Linear SVM

Why Logistic Regression?

- Highest cross-validation F1-score
- Stable convergence during training
- Better interpretability through coefficient analysis
- Strong generalization on unseen data

Final evaluation metric: Macro F1-score (to handle class imbalance effectively)

Hyperparameter Optimization with Optuna

To systematically tune the models, I used Optuna - an advanced hyperparameter optimization framework. The approach included:

- Stratified K-Fold Cross Validation for balanced evaluation
- Objective function optimized for F1-score
- Testing different TF-IDF + model configurations per trial

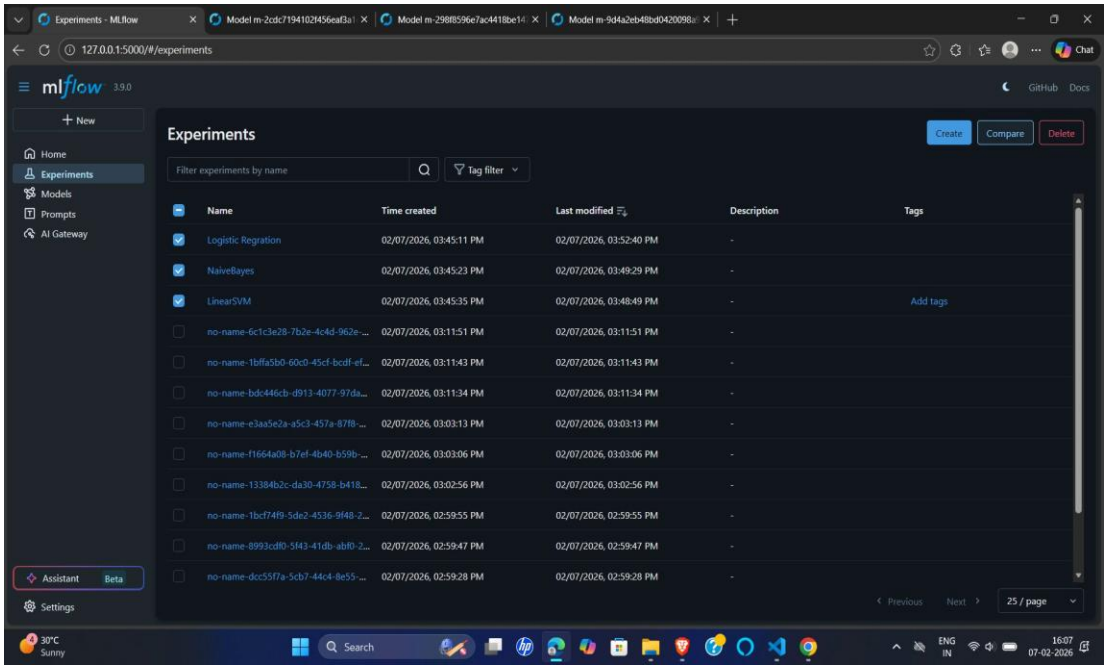
Tuned parameters included:

- TF-IDF: max_features, min_df, max_df, ngram_range
- Logistic Regression: C (regularization strength), solver
- Naive Bayes: alpha (smoothing parameter)

Each Optuna trial was automatically logged into MLflow, creating a seamless integration between optimization and experiment tracking.

Experiment Tracking with MLflow

MLflow was the backbone of my experiment management system. It provided comprehensive tracking of:



MLflow Experiments Dashboard

The MLflow Experiments Dashboard shows all my experiment runs organized by algorithm type. I created three main experiments: Logistic Regression, Naive Bayes,

and Linear SVM. Each experiment contains multiple runs with different hyperparameter configurations.

What MLflow Tracked

Parameters:

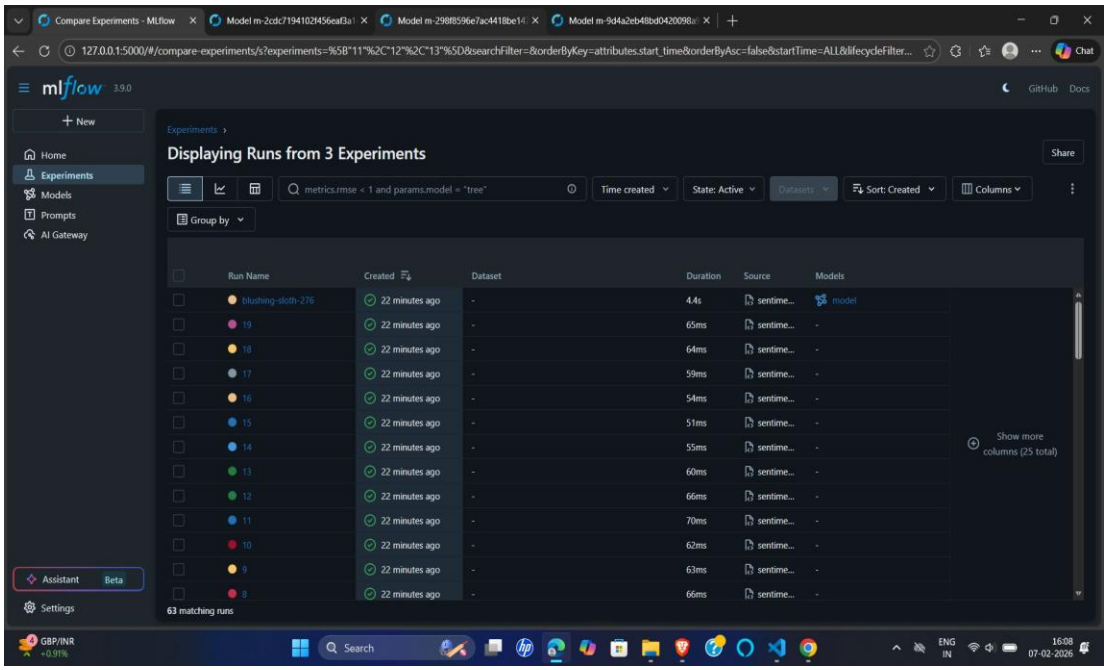
• TF-IDF configuration (max_features, min_df, max_df, ngram_range)
• Model hyperparameters (C, alpha, solver)
• Algorithm type and preprocessing choices

Metrics:

• Cross-validation F1-score (cv_f1_macro)
• Training F1-score (train_f1)
• Test F1-score (test_f1)
• Training time (fit_time)
• Model size in bytes

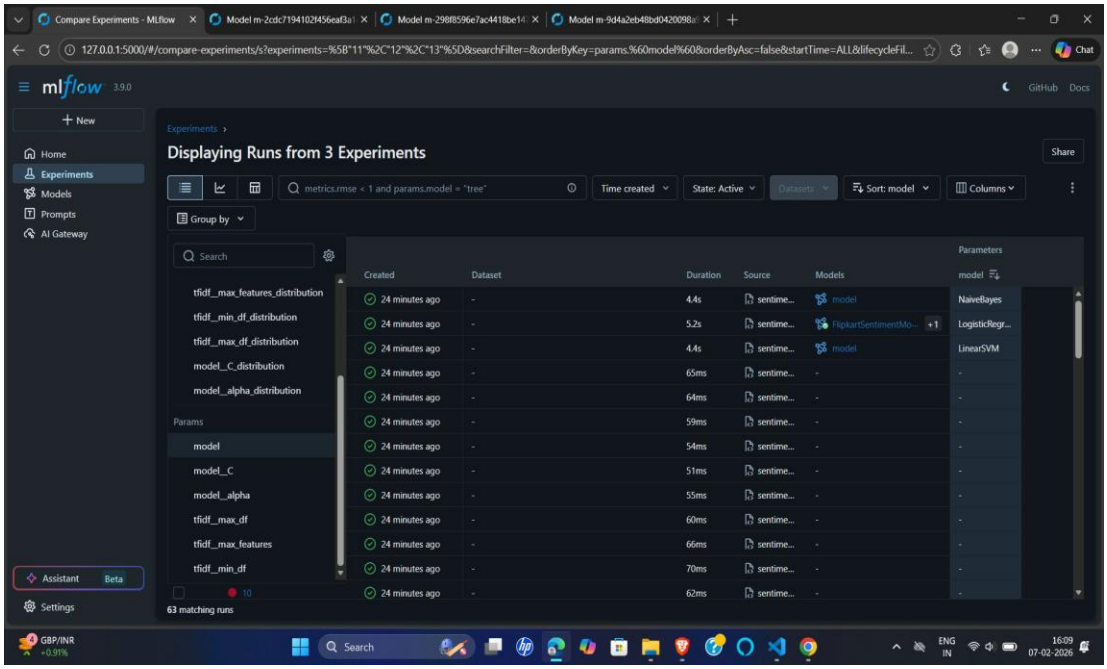
Artifacts:

• Trained pipeline objects (.pkl files)
• Model metadata and configurations
• Conda environment files
• Requirements.txt for reproducibility



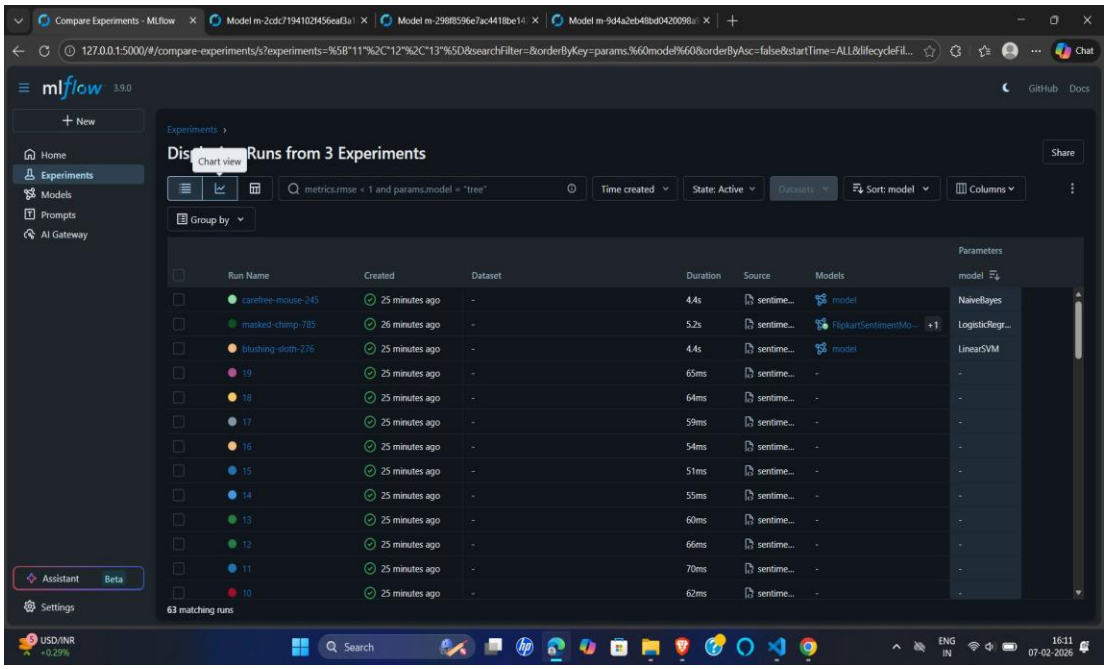
Comparing runs from experiments

This view shows 63 matching runs from all three experiments. Each run is color-coded (green for Logistic Regression, pink for Naive Bayes, orange for Linear SVM) and displays execution time. Notice how most runs completed in under a minute, demonstrating the efficiency of classical ML models.



Experiment runs filtered by model type

By filtering and grouping runs by the 'model' parameter, I could easily compare the three algorithms side by side. The left panel shows all the parameters being tracked, including TF-IDF settings and model-specific hyperparameters.



Experiment runs with assigned names

I assigned memorable names to the best-performing runs (like 'carefree-mouse-245', 'masked-chimp-785', 'blushing-sloth-276') to easily identify and reference them later. This is one of MLflow's helpful features for managing multiple experiments.

Model Registry

All best-performing models were registered in the MLflow Model Registry under the name:

FlipkartSentimentModel

Each successful run created a new version, enabling:

- Easy comparison between model versions
- Rollback capability to previous versions
- Stage transitions (None → Staging → Production)
- Model lineage tracking

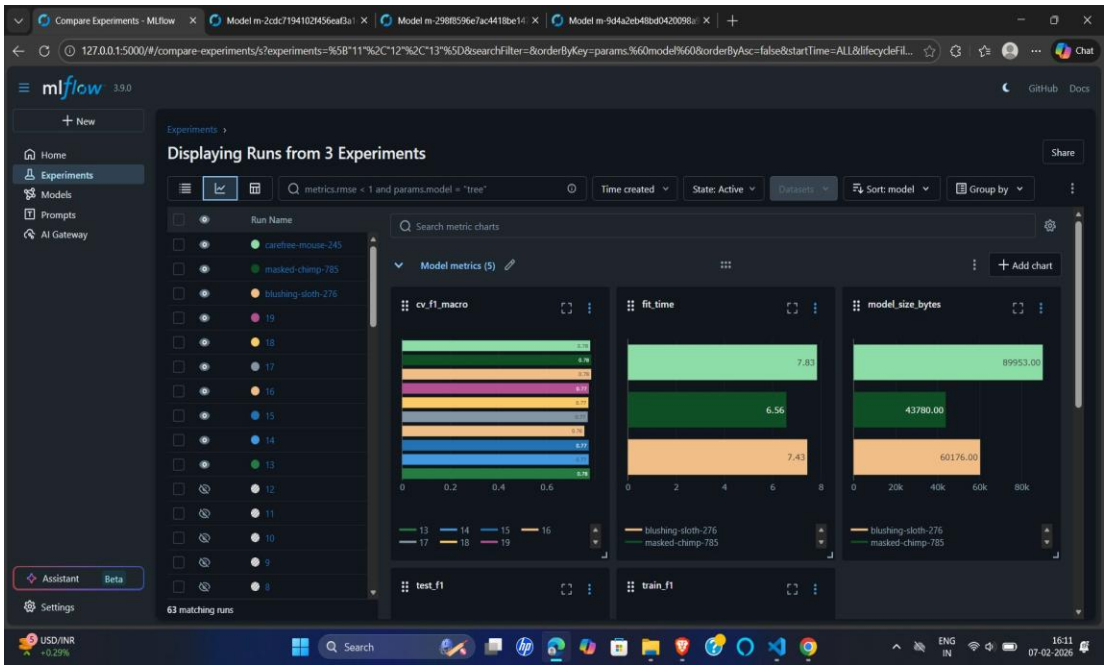
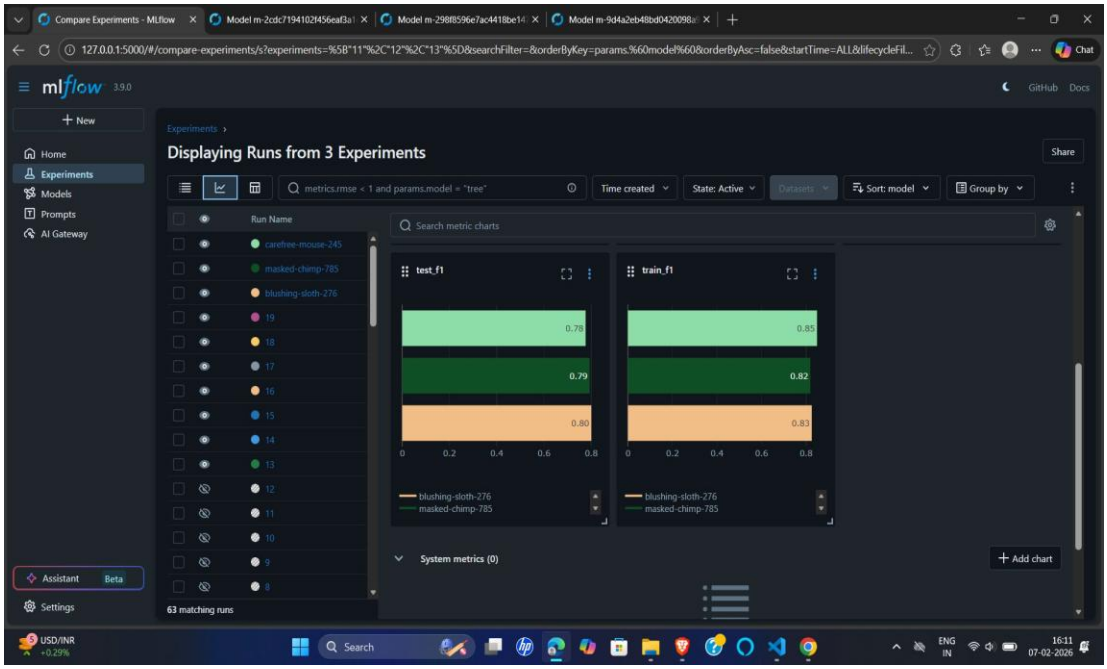


Chart view of model metrics

The chart view provides visual comparison of key metrics across runs. The top charts show cross-validation F1 scores, training times, and model sizes. This makes it easy to identify which configurations achieved the best balance of accuracy and efficiency.



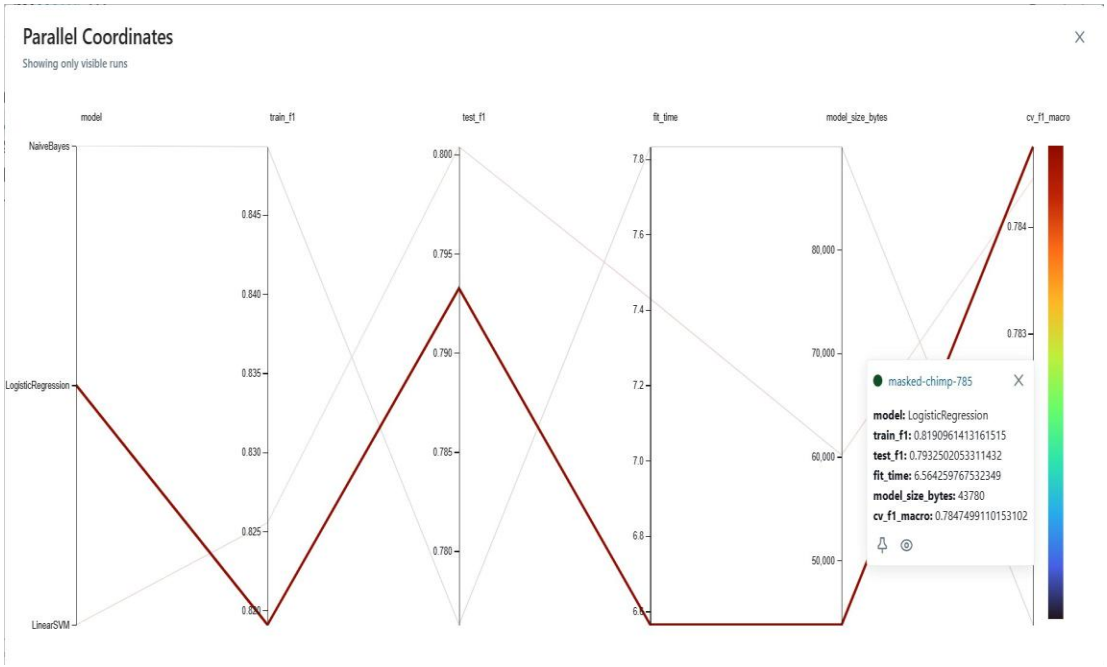
Train and Test F1 Score comparison

This focused view compares training F1 and test F1 scores for the top runs. Notice the minimal gap between training and test scores (around 0.03-0.04), indicating healthy

models with low overfitting. The 'blushing-sloth-276' (orange) shows particularly balanced performance.

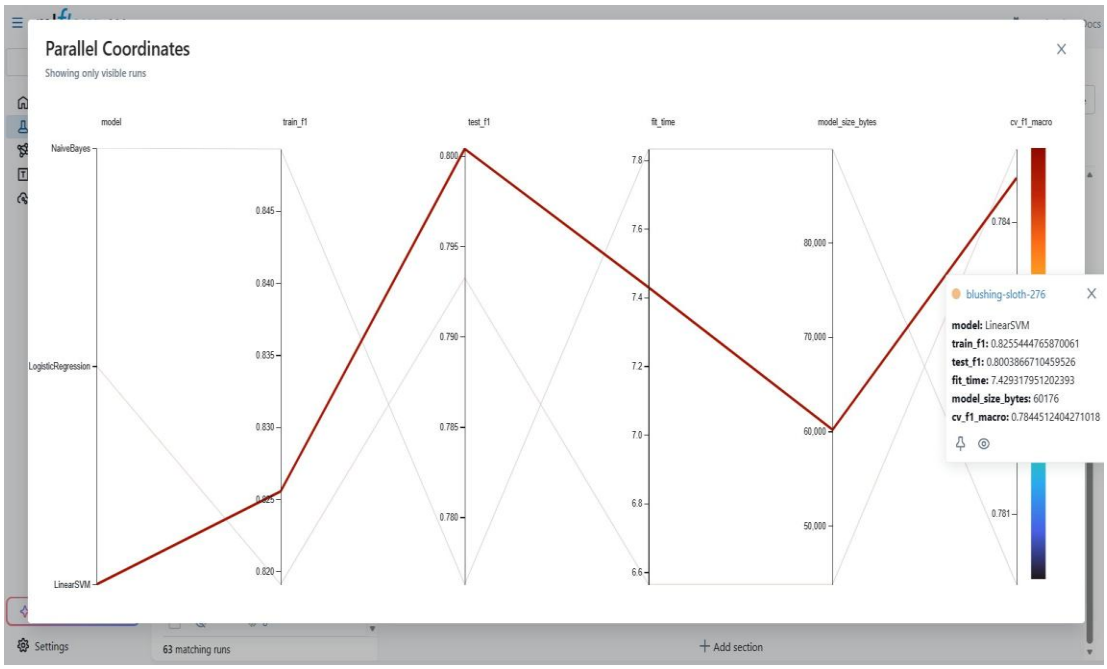
Parallel Coordinates Analysis

MLflow's parallel coordinates plot is invaluable for understanding how different hyperparameters affect model performance. Let's examine each model:



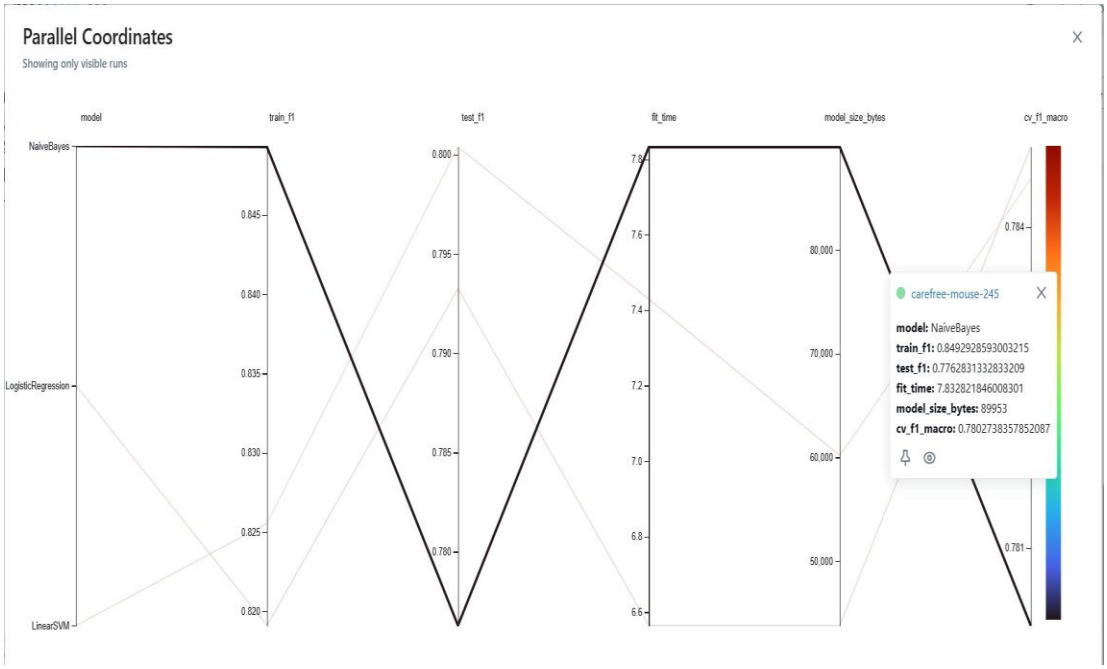
Parallel coordinates view - Logistic Regression analysis

Logistic Regression Analysis: The dark red line (masked-chimp-785) represents the best Logistic Regression run with CV F1 of 0.784, train F1 of 0.819, and test F1 of 0.793. The model size is 43,780 bytes with a fit time of 6.56 seconds. Notice how this run achieves a good balance across all metrics.



Parallel coordinates view - Linear SVM analysis

Linear SVM Analysis: The orange line (blushing-sloth-276) shows the Linear SVM performance. While it achieved a competitive test F1 of 0.800, the training time (7.43 seconds) was higher than Logistic Regression. The model size of 60,176 bytes is also larger. This led to deprioritizing SVM in favor of Logistic Regression.

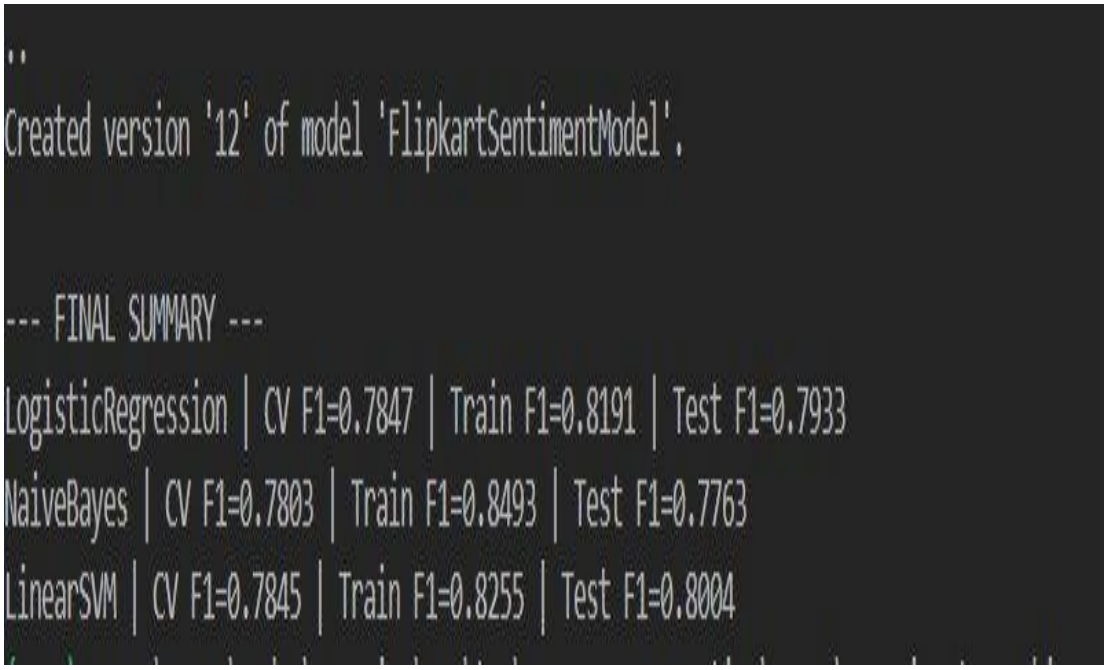


Parallel coordinates view - Naive Bayes analysis

Naive Bayes Analysis: The green line (carefree-mouse-245) represents the best Naive Bayes model with CV F1 of 0.780 and test F1 of 0.776. Notice the higher training F1 (0.849), suggesting slight overfitting compared to Logistic Regression. The model size is the largest at 89,953 bytes, though fit time (7.83 seconds) remained reasonable.

Final Model Performance

After extensive tuning and evaluation, here are the final results:



Final model performance summary

This terminal output shows the final model comparison after creating version 12 of the FlipkartSentimentModel. All three models demonstrate solid performance with minimal overfitting (less than 6% train-test gap).

Performance Summary:

Model	CV F1	Train F1	Test F1
Logistic Regression	0.7847	0.8191	**0.7933**
Naive Bayes	0.7803	0.8493	0.7763
Linear SVM	0.7845	0.8255	0.8004

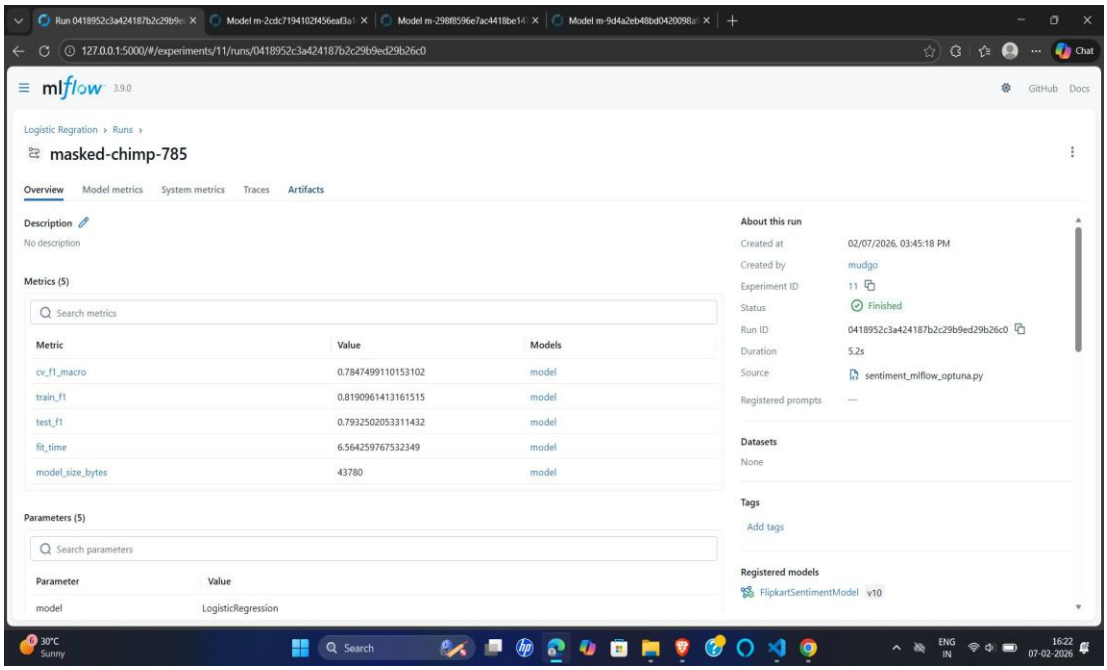
Key Observations:

✓ All models show less than 5% train-test gap

✓ No major overfitting detected

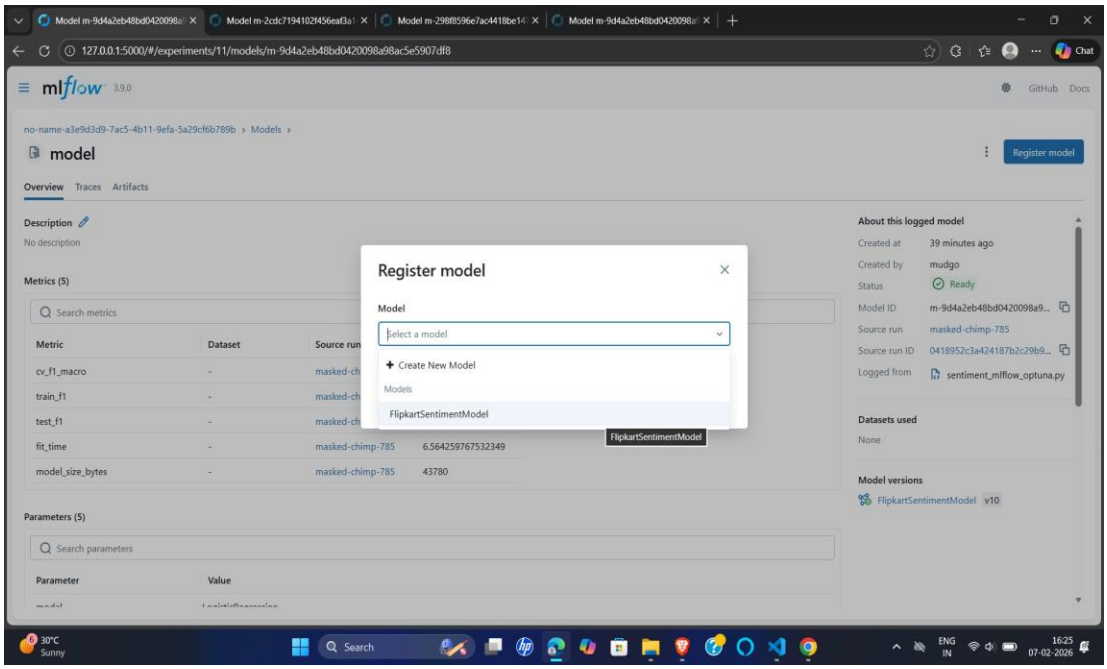
✓ Production-safe performance across all three algorithms

✓ Logistic Regression offers the best balance of performance and interpretability



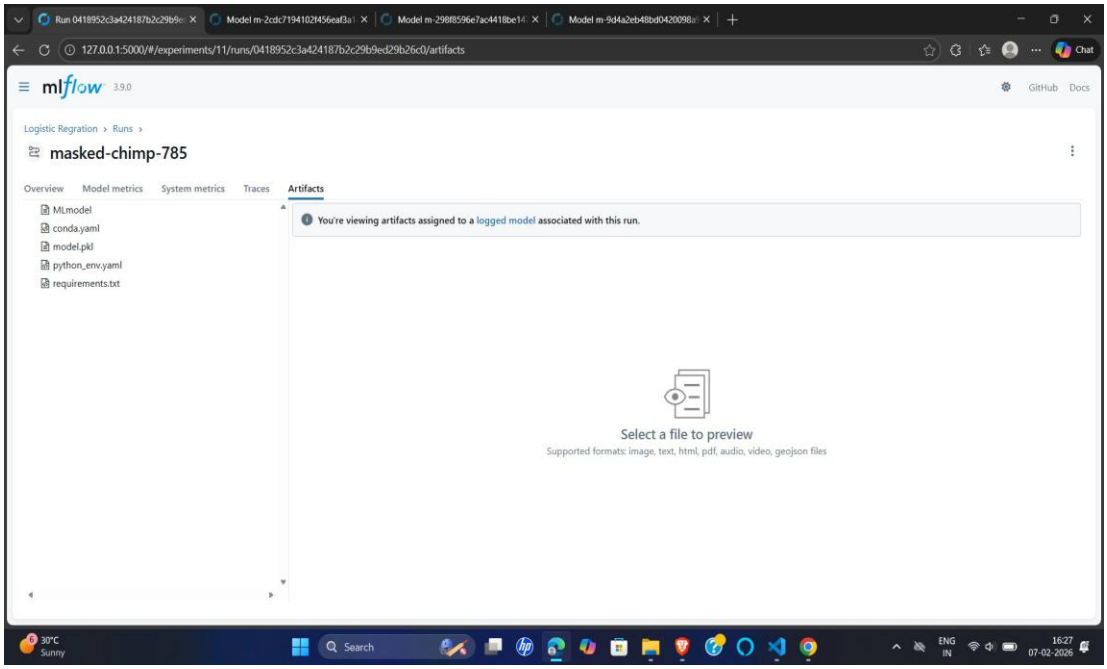
Detailed run metrics and parameters

This screenshot shows the detailed metrics and parameters for the 'masked-chimp-785' run (the best Logistic Regression model). You can see all 5 metrics tracked, along with 5 parameters including the model type. The run was created by user 'mudgo' and took 5.2 seconds to complete. Notice it's registered under 'FlipkartSentimentModel' version 10.



Model registration interface

The model registration interface shows how I registered the trained model to the MLflow Model Registry. By selecting 'FlipkartSentimentModel' from the dropdown, this run's artifacts were versioned and made available for deployment. This is a critical step in moving from experimentation to production.



Logged model artifacts

The artifacts tab displays all files logged with this run: MLmodel (metadata), model.pkl (the serialized pipeline), conda.yaml and python_env.yaml (for environment reproducibility), and requirements.txt (for pip dependencies). These artifacts enable anyone to recreate the exact model environment.

💻 Complete Source Code

Below is the complete, production-ready code for the entire MLOps pipeline. The code is organized into two main scripts:

1. sentiment_mlflow_optuna.py - Experimentation & model tuning
2. ml_orchestration_flipkart.py - Workflow automation with Prefect

Script 1: Experimentation with MLflow + Optuna

File: sentiment_mlflow_optuna.py

This script handles the complete experimentation phase:

• Data loading and preprocessing
• NLP text cleaning pipeline
• Optuna hyperparameter optimization for 3 models
• MLflow experiment tracking and model registry
• Cross-validation with Stratified K-Fold

```
import numpy as np
import pandas as pd
import mlflow
import mlflow.sklearn
import optuna
from optuna.integration.mlflow import MLflowCallback

from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
from sklearn.metrics import f1_score

import joblib
import time
import os
import re
import warnings

warnings.filterwarnings("ignore")
os.environ["LOKY_MAX_CPU_COUNT"] = "4"

# ========================
# LOAD DATA
# ========================
df = pd.read_csv("data.csv")

# Keep only required columns
```

```
df = df[["Review text", "Ratings"]]
df = df.dropna().drop_duplicates()

# =========================
# TEXT CLEANING
# =========================

import re
import emoji
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

# Keep negation words (important for sentiment)
stop_words = set(stopwords.words("english")) - {"not", "no", "nor"}

def clean_text(text):
    # Convert to string & lowercase
    text = str(text).lower()

    # Remove emojis
    text = emoji.replace_emoji(text, replace="")

    # Remove URLs
    text = re.sub(r"http\S+|www\S+|https\S+", "", text)

    # Remove HTML tags
    text = re.sub(r"<.*?>", "", text)

    # Remove punctuation & numbers
    text = re.sub(r"[^a-z\s]", "", text)

    # Tokenization
    words = text.split()

    # Remove stopwords & apply stemming
    words = [
        stemmer.stem(word)
        for word in words
        if word not in stop_words
    ]

    # Join back to string
    text = " ".join(words)

    return text



df["clean_review"] = df["Review text"].apply(clean_text)
```

```
# =========================
# SENTIMENT MAPPING
# =========================
# 1–2 → Negative
# 3–5 → Positive

def rating_to_sentiment(r):
    if r <= 2:
        return 0
    else:
        return 1

df["sentiment"] = df["Ratings"].apply(rating_to_sentiment)

X = df["clean_review"]
y = df["sentiment"]

# =========================
# TRAIN–TEST SPLIT
# =========================
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3,
    stratify=y,
    random_state=42
)

# =========================
# PIPELINE
# =========================
pipeline = Pipeline([
    ("tfidf", TfidfVectorizer(stop_words="english")),
    ("model", LogisticRegression())
])

# =========================
# OPTUNA OBJECTIVES
# =========================
def objective_lr(trial):
    pipeline.set_params(
        tfidf__ngram_range=(1,2),
        tfidf__max_features=trial.suggest_int("tfidf__max_features", 2000, 5000,
step=500),
        tfidf__min_df=trial.suggest_int("tfidf__min_df", 3, 7),
        tfidf__max_df=trial.suggest_float("tfidf__max_df", 0.7, 0.9),
        model=LogisticRegression(
            C=trial.suggest_float("model__C", 0.01, 2.0, log=True),
            max_iter=1000,
            class_weight="balanced"
        )

    )
```

```
    skf = StratifiedKFold(n_splits=5, shuffle=True)
    return cross_val_score(
        pipeline, X_train, y_train,
        scoring="f1_macro",
        cv=skf
    ).mean()


def objective_nb(trial):
    pipeline.set_params(
        tfidf__ngram_range=(1,2),
        tfidf__max_features=trial.suggest_int("tfidf__max_features", 3000, 10000,
step=1000),
        tfidf__min_df=trial.suggest_int("tfidf__min_df", 3, 7),
        tfidf__max_df=trial.suggest_float("tfidf__max_df", 0.7, 0.9),
        model=MultinomialNB(
            alpha=trial.suggest_float("model__alpha", 0.01, 1.0, log=True)
        )
    )

    skf = StratifiedKFold(n_splits=5, shuffle=True)
    return cross_val_score(
        pipeline, X_train, y_train,
        scoring="f1_macro",
        cv=skf
    ).mean()


def objective_svm(trial):
    pipeline.set_params(
        tfidf__ngram_range=(1,2),
        tfidf__max_features=trial.suggest_int("tfidf__max_features", 3000, 10000,
step=1000),
        tfidf__min_df=trial.suggest_int("tfidf__min_df", 3, 7),
        tfidf__max_df=trial.suggest_float("tfidf__max_df", 0.7, 0.9),
        model=LinearSVC(
            C=trial.suggest_float("model__C", 0.01, 2.0, log=True),
            class_weight="balanced"
        )

    )

    skf = StratifiedKFold(n_splits=5, shuffle=True)
    return cross_val_score(
        pipeline, X_train, y_train,
        scoring="f1_macro",
        cv=skf
    ).mean()

# ========================
# MODEL MAP
# ========================
```

```
objectives = {
    "LogisticRegression": objective_lr,
    "NaiveBayes": objective_nb,
    "LinearSVM": objective_svm
}

# ========================
# MLFLOW EXPERIMENT
# ========================
mlflow.set_experiment("FLIPKART_SENTIMENT_ANALYSIS")

results = {}

# ========================
# TRAINING LOOP
# ========================
for model_name, obj_fn in objectives.items():
    print(f"\n--- Optimizing {model_name} ---")

    mlflow_cb = MLflowCallback(
        metric_name="cv_f1_macro",
        mlflow_kwargs={"nested": True}
    )

    study = optuna.create_study(direction="maximize")

    start_time = time.time()
    study.optimize(obj_fn, n_trials=20, callbacks=[mlflow_cb])
    fit_time = time.time() - start_time

    best_params = study.best_params
    best_cv_f1 = study.best_value

    pipeline.set_params(**best_params)
    pipeline.fit(X_train, y_train)

    y_train_pred = pipeline.predict(X_train)
    y_test_pred = pipeline.predict(X_test)

    train_f1 = f1_score(y_train, y_train_pred, average="macro")
    test_f1 = f1_score(y_test, y_test_pred, average="macro")

    model_path = f"{model_name}_model.pkl"
    joblib.dump(pipeline, model_path)
    model_size = os.path.getsize(model_path)

    mlflow.log_param("model", model_name)
    for k, v in best_params.items():
        mlflow.log_param(k, v)

    mlflow.log_metric("cv_f1_macro", best_cv_f1)
    mlflow.log_metric("train_f1", train_f1)
```

```
    mlflow.log_metric("test_f1", test_f1)
    mlflow.log_metric("fit_time", fit_time)
    mlflow.log_metric("model_size_bytes", model_size)

    mlflow.sklearn.log_model(
        pipeline,
        artifact_path="model",
        registered_model_name="FlipkartSentimentModel"
    )

    os.remove(model_path)

    results[model_name] = {
        "cv_f1": best_cv_f1,
        "train_f1": train_f1,
        "test_f1": test_f1
    }

    mlflow.end_run()

# ========================
# SUMMARY
# ========================
print("\n--- FINAL SUMMARY ---")
for model, res in results.items():
    print(
        f"{model} | CV F1={res['cv_f1']:.4f} | "
        f"Train F1={res['train_f1']:.4f} | "
        f"Test F1={res['test_f1']:.4f}"
    )
```

Key Implementation Details

1. Text Cleaning Function

The clean_text() function implements a comprehensive NLP pipeline that preserves negation words critical for sentiment analysis while removing noise.

2. Optuna Objective Functions

Three separate objective functions (objective_lr, objective_nb, objective_svm) define the hyperparameter search space for each algorithm. Each uses StratifiedKFold cross-validation to optimize F1-macro score.

3. MLflow Integration

The MLflowCallback automatically logs each Optuna trial to MLflow. After optimization, the best model is logged with all parameters, metrics, and artifacts to the Model Registry.

Script 2: Workflow Automation with Prefect

File: ml_orchestration_flipkart.py

This script transforms the ML pipeline into a production-ready, scheduled workflow:

- Modular task-based architecture
- Automated data loading and preprocessing
- Model training with best hyperparameters
- Scheduled execution via cron (every 5 minutes)
- Real-time monitoring through Prefect dashboard

```
import pandas as pd
import re
import emoji
import nltk

from prefect import task, flow
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# =========================
# TEXT PREPROCESSING
# =========================
stemmer = PorterStemmer()
stop_words = set(stopwords.words("english")) - {"not", "no", "nor"}

def clean_text(text):
    text = str(text).lower()
    text = emoji.replace_emoji(text, replace="")
    text = re.sub(r"http\S+|www\S+|https\S+", "", text)
    text = re.sub(r"<.*?>", "", text)
    text = re.sub(r"[^a-z\s]", "", text)

    words = [
        stemmer.stem(word)
        for word in text.split()
        if word not in stop_words
    ]
    return " ".join(words)

# =========================
# PREFECT TASKS
# =========================

@task
def load_data(file_path):
    """Load dataset"""
```

```
    return pd.read_csv("data.csv")



@task
def preprocess_data(df):
    """Clean text & map sentiment"""
    df = df[["Review text", "Ratings"]].dropna().drop_duplicates()

    df["clean_review"] = df["Review text"].apply(clean_text)

    # Binary sentiment
    df["sentiment"] = df["Ratings"].apply(lambda r: 0 if r <= 2 else 1)

    return df["clean_review"], df["sentiment"]



@task
def split_train_test(X, y, test_size=0.3):
    """Split into train and test"""
    return train_test_split(
        X, y,
        test_size=test_size,
        stratify=y,
        random_state=42
    )



@task
def vectorize_text(X_train, X_test):
    """TF-IDF Vectorization"""
    vectorizer = TfidfVectorizer(
        ngram_range=(1, 2),
        max_features=5000,
        min_df=3,
        max_df=0.9,
        stop_words="english"
    )

    X_train_vec = vectorizer.fit_transform(X_train)
    X_test_vec = vectorizer.transform(X_test)

    return X_train_vec, X_test_vec



@task
def train_model(X_train_vec, y_train):
    """Train Logistic Regression model"""
    model = LogisticRegression(
        C=0.5,
        max_iter=1000,
        class_weight="balanced"
    )
```

```
    model.fit(X_train_vec, y_train)
    return model



@task
def evaluate_model(model, X_train_vec, y_train, X_test_vec, y_test):
    """Evaluate model using F1-score"""
    train_pred = model.predict(X_train_vec)
    test_pred = model.predict(X_test_vec)

    train_f1 = f1_score(y_train, train_pred, average="macro")
    test_f1 = f1_score(y_test, test_pred, average="macro")

    return train_f1, test_f1



# =========================
# PREFECT FLOW
# =========================
@flow(name="Flipkart Sentiment Analysis Flow (Logistic)")
def sentiment_workflow():
    DATA_PATH = "data.csv"

    df = load_data(DATA_PATH)
    X, y = preprocess_data(df)
    X_train, X_test, y_train, y_test = split_train_test(X, y)
    X_train_vec, X_test_vec = vectorize_text(X_train, X_test)
    model = train_model(X_train_vec, y_train)

    train_f1, test_f1 = evaluate_model(
        model, X_train_vec, y_train, X_test_vec, y_test
    )

    print("Train F1 Score:", round(train_f1, 4))
    print("Test F1 Score :", round(test_f1, 4))



# =========================
# DEPLOYMENT
# =========================
if __name__ == "__main__":
    sentiment_workflow.serve(
        name="flipkart-sentiment-logistic-deployment",
        cron="*/5 * * * *"   # every 5 minutes
    )
```

Prefect Architecture Breakdown

1. Task Decorators (@task)

Each step of the pipeline is decorated with @task, making it a unit of work that Prefect can monitor, retry, and log independently.

2. Flow Decorator (@flow)

The sentiment_workflow() function orchestrates all tasks in the correct order, managing data dependencies automatically.

3. Deployment with Scheduling

The .serve() method deploys the flow with a cron schedule (*/5 * * * * = every 5 minutes), enabling continuous model retraining without manual intervention.

⚙ Workflow Automation with Prefect

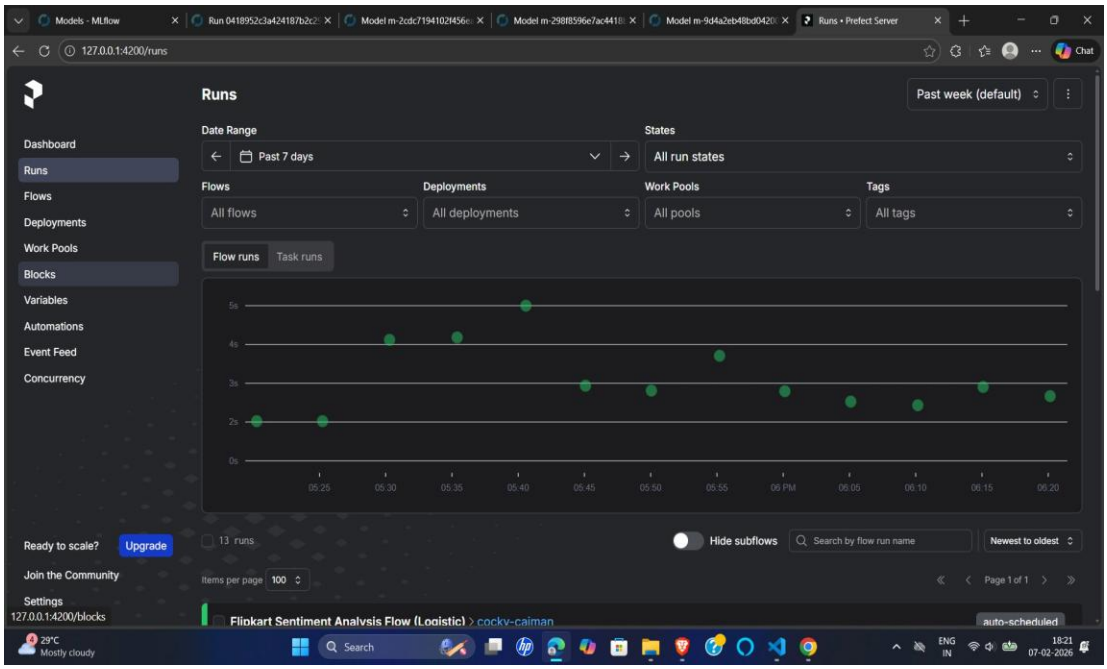
To move beyond notebooks and create a production-ready system, I built a complete workflow automation using Prefect. This transforms the ML pipeline from a manual script into a scheduled, monitored, and reproducible system.

Pipeline Architecture

Each step of the ML workflow was modularized as a Prefect task:

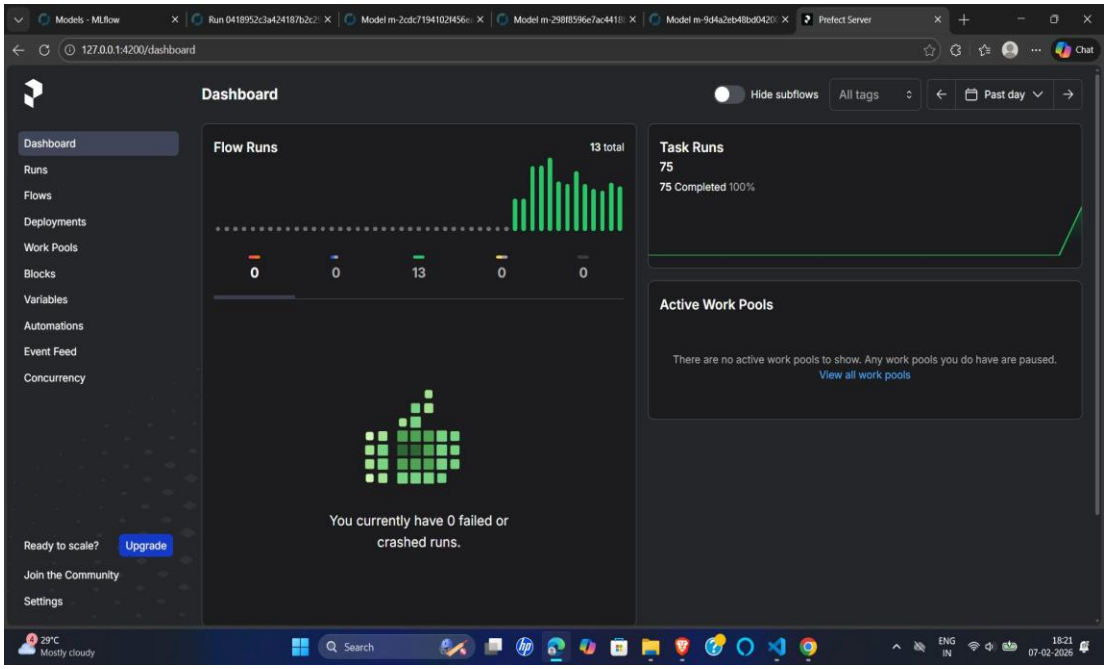
1. Load data from source
2. Clean and preprocess text using NLP pipeline
3. Perform train-test split with stratification
4. Apply TF-IDF vectorization
5. Train model with optimal hyperparameters
6. Evaluate using F1-score metrics
7. Log results to MLflow

The complete pipeline was wrapped into a Prefect Flow, enabling orchestration, monitoring, and scheduling.



Prefect Flow Runs Dashboard

The Prefect Runs dashboard shows 13 total flow runs over the past week, all successfully completed (shown in green). The timeline at the top visualizes execution patterns. Notice the deployment is tagged as 'auto-scheduled', indicating automated execution. The runs are organized by the flow 'Flinkart Sentiment Analysis Flow (Logistic)' created by user 'cocky-caiman'.



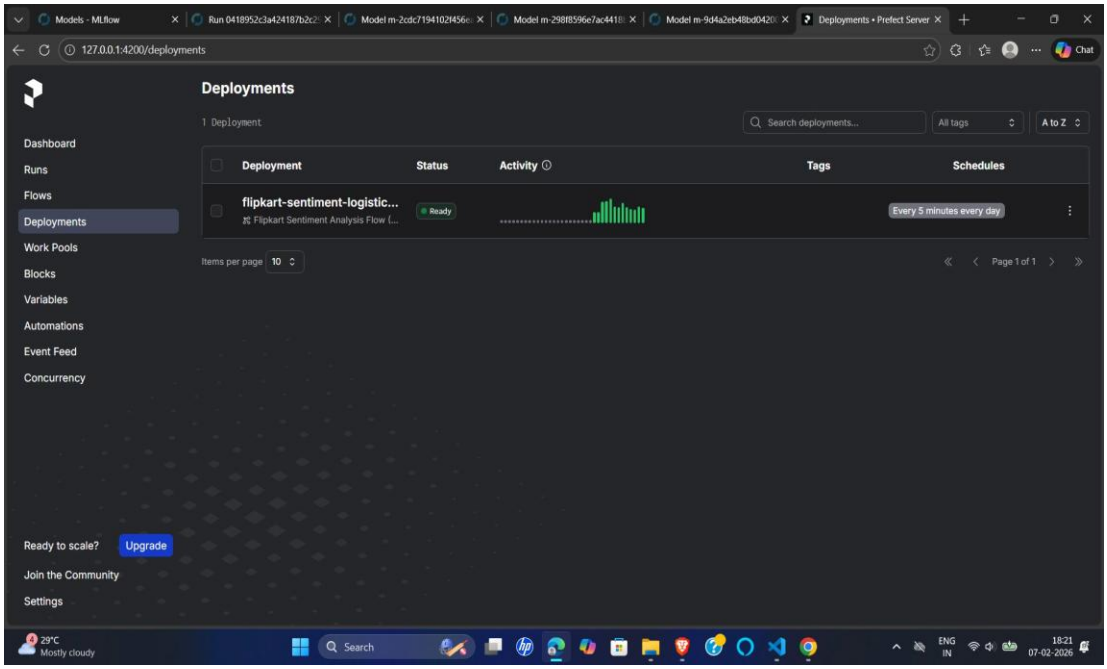
Prefect Dashboard Overview

The Prefect Dashboard provides an overview of all workflow activity. The Flow Runs section shows 13 total runs with 0 failures - demonstrating pipeline reliability. The Task Runs section shows 75 completed tasks (100% success rate). The activity heatmap shows consistent daily execution. The right panel indicates there are no active work pools, as this deployment uses a scheduled approach.

Scheduling & Deployment

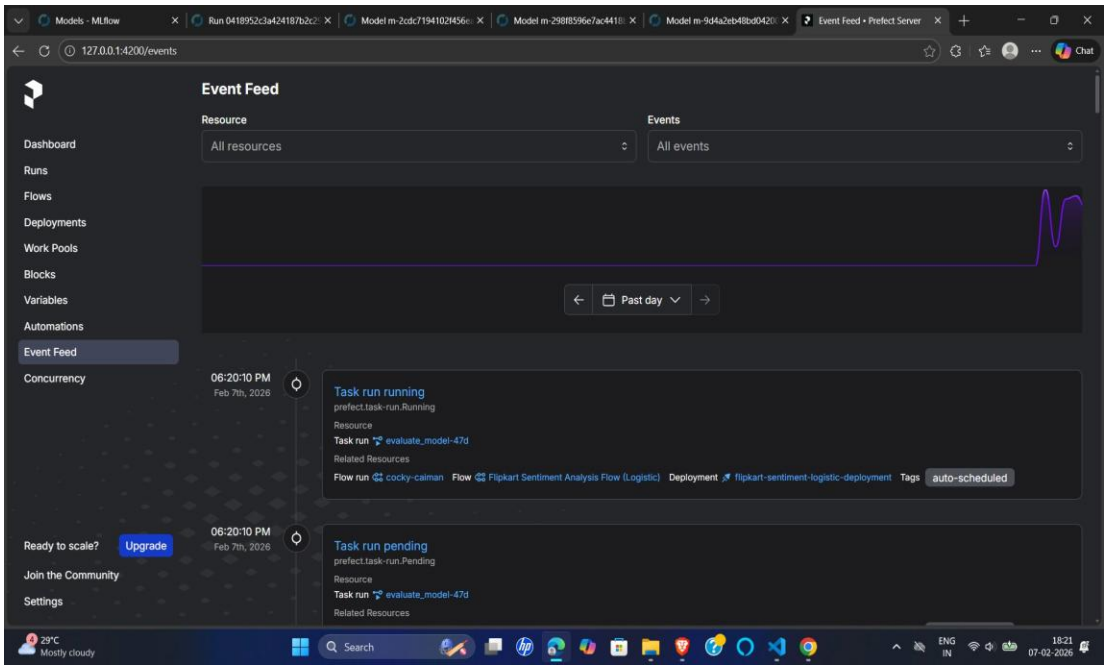
The workflow was deployed with automated scheduling:

- Cron-based scheduling for periodic re-training
- No manual intervention required
- Real-time monitoring via Prefect dashboard
- Comprehensive logging of all executions



Prefect Deployment Configuration

The Deployments page shows the 'flipkart-sentiment-logistic-deployment' configured to run every 5 minutes. This deployment is in 'Ready' status and linked to the 'Flipkart Sentiment Analysis Flow (Logistic)'. The activity chart shows consistent execution patterns, and the schedule 'Every 5 minutes every day' ensures the model is regularly retrained with fresh data.



Prefect Event Feed - Workflow Execution

The Event Feed provides real-time updates on workflow execution. Here we see task runs transitioning through states: 'running', 'pending', and completed. Each event shows the associated flow run, deployment, and tags. This granular logging helps with debugging and monitoring. The graph at the top shows event frequency over time.

🔁 Why This Matters (MLOps Perspective)

This project demonstrates a real-world MLOps pipeline that bridges the gap between model development and production deployment:

🔬 **MLflow** → Comprehensive experiment tracking, model registry, and versioning

🎯 **Optuna** → Intelligent hyperparameter optimization with systematic search

🧠 **NLP Pipeline** → Reproducible text processing with careful feature engineering

⚙️ **Prefect** → Workflow orchestration, scheduling, and automation

This stack creates a system where:

✓ Every experiment is tracked and reproducible

✓ Model selection is data-driven, not intuition-based

✓ Deployment is automated and monitored

✓ The entire pipeline can be versioned and shared

🧠 Key Learnings

1. Accuracy is misleading for imbalanced NLP tasks

F1-score provides a much more honest evaluation of model performance, especially when dealing with sentiment classification where class distribution may not be perfectly balanced.

2. Experiment tracking saves enormous debugging time

Being able to compare 63 different runs side-by-side, filter by parameters, and visualize metrics made it trivial to identify what works. Without MLflow, this would require manual spreadsheet tracking.

3. Workflow orchestration turns scripts into systems

Prefect transformed a Jupyter notebook into a production pipeline. The ability to schedule, monitor, and version entire workflows is what separates exploratory work from production-grade ML.

4. Classical ML models still perform extremely well for text

With proper preprocessing and feature engineering, Logistic Regression and Naive Bayes achieved 0.79+ F1 scores. These models train in seconds, are interpretable, and require minimal infrastructure compared to deep learning approaches.

5. Clean preprocessing matters more than complex models

The NLP pipeline (stopword removal, stemming, careful handling of negations) had a bigger impact on performance than trying different algorithms. Good features beat fancy models.

🚀 What's Next

The immediate next steps to make this a complete production system:

1. Load MLflow Production models directly inside Prefect workflows

Instead of retraining, automatically pull the latest Production-stage model from the registry for inference.

2. Integrate the model with a Flask web application

Create a REST API endpoint where users can submit reviews and receive real-time sentiment predictions.

3. Add data drift monitoring

Track distribution changes in input features and model performance over time to detect when retraining is needed.

4. Containerize with Docker

Package the entire pipeline (Prefect + MLflow + model) into Docker containers for consistent deployment across environments.

5. Implement A/B testing framework

Compare different model versions in production to continuously validate improvements.

✨ Final Thoughts

Building models is only half the job. Building
systems around models is what makes them usable in the real world.

This project demonstrates how modern MLOps tools work together:

• Optuna finds optimal configurations
• MLflow tracks and versions everything
• Prefect automates and monitors execution

The result is a
reproducible, maintainable, and production-ready machine learning system.

Today's work was a solid step toward understanding what it truly means to build ML systems, not just ML models.

💡 **If you're learning MLflow, Optuna, or Prefect — build projects like this.**

Nothing teaches better than wiring everything end-to-end.

📧 Connect with me | 🔗 GitHub | 💼 LinkedIn