Architectural Blueprints for Interpretable Narrative Classification Systems: From Theory to Production

1. Introduction: The Imperative of Explainability in Computational Linguistics

The deployment of machine learning models in production environments has historically been plagued by a trade-off between predictive performance and interpretability. In the domain of Natural Language Processing (NLP), this dichotomy is particularly acute. As architectures have evolved from simple Bag-of-Words models to complex transformers and ensemble methods, the decision boundary of these classifiers has become increasingly opaque. For a genre classification system—tasked with categorizing narrative texts into genres such as Fantasy, Horror, or Science Fiction—the ability to explain a prediction is not merely a debugging convenience; it is a prerequisite for trust, bias detection, and rigorous engineering validation.

This report details the comprehensive construction of a machine learning-driven Genre Classification API that prioritizes explainability alongside accuracy. The architecture described herein follows a tripartite structure: the Model Pipeline, the Explainability Layer utilizing Local Interpretable Model-agnostic Explanations (LIME), and the Deployment Layer orchestrated via FastAPI. This document serves as an exhaustive technical reference, synthesizing theoretical underpinnings with practical implementation details derived from contemporary research benchmarks and software engineering best practices.[1]

The objective is to move beyond the superficial implementation of libraries and explore the mechanical and mathematical realities of building such a system from scratch. We will analyze the selection of Support Vector Machines (SVM) over deep learning alternatives for specific interview contexts, the nuance of Term Frequency-Inverse Document Frequency (TF-IDF) vectorization in high-dimensional sparse spaces, and the critical concurrency patterns required to serve CPU-bound explainability tasks in an asynchronous web framework.

---

2. Phase 1: The Genre Classification Pipeline

The foundational stratum of any intelligent application is the data pipeline. In the context of genre classification, this involves the transformation of unstructured, variable-length narrative text into structured, numerical feature vectors that preserve semantic discriminability.

2.1 Data Infrastructure and Acquisition Strategies

The efficacy of a supervised learning model is bounded by the quality and diversity of its training data. For genre classification, we must curate a dataset that reflects the linguistic complexity of real-world narratives. Research identifies several candidate datasets that serve as industry standards for this task.

2.1.1 Dataset Selection and Schema Analysis

The **CMU Book Summary Dataset** represents a gold standard in this domain, containing 16,559 book summaries extracted from Wikipedia, aligned with metadata from Freebase.[4] This dataset is particularly valuable because it includes a rich taxonomy of genres, allowing for robust multi-class or multi-label classification tasks. The schema typically includes:

- **Wikipedia ID:** A unique identifier for lineage tracking.
- **Book Title:** Useful for auxiliary feature engineering but often excluded from the primary text body to prevent overfitting on specific franchises (e.g., "Harry Potter" usually implies Fantasy).
- **Plot Summary:** The primary corpus for NLP analysis, ranging from 50 to 1,000 words.
- **Genre Labels:** Often presented as a list (e.g., {"Fantasy", "Young Adult", "Fiction"}).

Alternative datasets include the **"TinyStories"** or **"1000 Stories"** datasets found on repositories like Kaggle.[1] These are smaller but often cleaner, containing columns such as text, label, and title. For the purpose of building an interview-ready portfolio piece, utilizing a subset of the CMU dataset or the 1000 Stories dataset provides a balance between computational manageability and statistical significance.

2.1.2 Distributional Challenges and Class Imbalance

A critical insight derived from exploratory data analysis (EDA) on these datasets is the prevalence of class imbalance. As noted in the OpenDataBay dataset analysis, broad genres like "Thriller" (22%) and "Fantasy" (19%) often dominate, while niche genres constitute a "long tail".[7]

This distribution poses a risk to model training. A classifier trained on imbalanced data may converge to a trivial solution—predicting the majority class (e.g., "Fiction") for every input—to maximize global accuracy while failing to capture the nuances of minority classes. Techniques to mitigate this include:

1. **Stratified Sampling:** Ensuring that the train/test split maintains the same proportion of

genres as the original dataset.
2. **Oversampling/Undersampling:** adjusting the dataset size, though this risks overfitting (oversampling) or information loss (undersampling).
3. **Class Weights:** Modifying the loss function of the SVM to penalize misclassifications of minority classes more heavily. For this implementation, we prioritize Stratified Sampling during the preprocessing phase to ensure valid evaluation metrics.

2.2 Text Preprocessing: Signal Enhancement and Noise Reduction

Raw text is inherently noisy. The goal of preprocessing is to strip away non-informative variation, collapsing the vocabulary size to a manageable dimension without destroying the semantic signal required for genre differentiation.

2.2.1 Normalization and Tokenization

The first step is normalization, primarily lowercasing all text. In a Vector Space Model (VSM), the tokens "Alien" and "alien" are treated as orthogonal dimensions if not normalized. Collapsing them reduces the feature space dimensionality significantly.
Tokenization—the process of splitting text into atomic units—must handle linguistic nuances. While simple whitespace splitting is computationally cheap, it fails on punctuation (e.g., "end." vs "end"). We utilize the **NLTK (Natural Language Toolkit)** tokenizer, which uses regular expressions to correctly separate punctuation marks from words.[8]

Preprocessing Step	**Input Example**	**Output Example**	**Rationale**
Lowercasing	"The Dark Tower"	"the dark tower"	Reduces vocabulary size; "Dark" and "dark" have same semantic value.
Tokenization	"It's alive!"	["It", "'s", "alive", "!"]	Separates clitics and punctuation for precise feature counting.
Punctuation Removal	["alive", "!"]	["alive"]	Punctuation rarely defines genre (unlike sentiment analysis).

2.2.2 Stopword Removal: Theoretical Implications

Stopwords are high-frequency function words (e.g., "the", "is", "at", "which") that serve grammatical purposes but carry low lexical content. According to Zipf's Law, these words account for the vast majority of tokens in a corpus. Retaining them creates noise in a TF-IDF model because their high frequency across all documents results in a near-zero IDF score, rendering them useless for discrimination.[10]

We employ the NLTK English stopword list for filtration.[8] However, blind removal can be dangerous. In some contexts, the *style* of writing (heavily dependent on function words) helps define the genre (e.g., Victorian Lit vs. Hardboiled Detective fiction). For a plot-based classifier, however, the semantic content (nouns and verbs) is paramount, justifying aggressive stopword removal.

2.2.3 Lemmatization vs. Stemming

To further reduce sparsity, we consider morphological normalization. Stemming (e.g., Porter Stemmer) aggressively chops words to their base (e.g., "computing" -> "comput"), often resulting in non-words. Lemmatization (e.g., WordNet Lemmatizer) uses a dictionary to map words to their canonical root (lemma). While lemmatization is computationally more expensive, it preserves readability—a crucial factor when we later display these features in the LIME explanation. An explanation saying "run" contributed to the prediction is clearer than "runn".

2.3 Feature Engineering: The TF-IDF Vector Space

Machine learning algorithms operate on numerical vectors, not strings. We must map our document collection D to a vector space V. We select **Term Frequency-Inverse Document Frequency (TF-IDF)** over modern embeddings (Word2Vec, BERT) for this specific architectural pattern.

2.3.1 Justification for TF-IDF in Explainable AI

While transformer models (BERT, GPT) achieve state-of-the-art accuracy, they produce dense embedding vectors where dimensions represent abstract, distributed semantic concepts. Explaining a decision made by a BERT model is complex because input tokens interact non-linearly through self-attention layers.

In contrast, TF-IDF produces **interpretable** features. Each dimension in the vector corresponds to a specific word in the vocabulary. If Dimension 452 has a high weight, we know exactly which word is responsible. This direct mapping is vital for the "Step 5" Explainability phase using LIME, as it allows for a clear, causal link between input words and model output.[13]

2.3.2 Mathematical Formulation

The TF-IDF score $w_{t,d}$ for a term t in document d is calculated as:

$$w_{t,d} = \text{tf}(t,d) \times \text{idf}(t)$$
Where:

- $\text{tf}(t,d)$ is the raw frequency of term t in d (often log-scaled to dampen the impact of very long documents).
- $\text{idf}(t) = \log(\frac{N}{1 + df(t)})$, where N is the total number of documents and $df(t)$ is the number of documents containing term t.

This formula naturally penalizes words that appear everywhere (low discriminative power) and rewards words that are rare in the corpus but frequent in the specific document—exactly the behavior needed to identify genre-specific lexicon like "spaceship" or "spellbook".[13]

2.3.3 N-Grams and Dimensionality

Single words (unigrams) often lack context. "Court" could mean a legal drama or a royal fantasy. By setting the ngram_range=(1, 2) in the TfidfVectorizer, we capture bigrams like "Supreme Court" vs. "Royal Court," significantly resolving ambiguity.[13] This expands the feature space exponentially, necessitating the use of sparse matrices (where only non-zero values are stored) to prevent memory overflow.

2.4 Discriminative Modeling: Support Vector Machines (SVM)

With the text vectorized, we proceed to model selection. We employ a Support Vector Machine (SVM) as the classifier of choice.

2.4.1 SVM Theory for Text Classification

SVMs are uniquely suited for text classification due to the **High Dimensionality** and **Sparsity** of the data. Text feature spaces are often linearly separable; there exists a hyperplane that can separate "Space Opera" from "Regency Romance" with high accuracy. The SVM algorithm finds the "maximum margin" hyperplane, which offers robust generalization.[15]
We utilize the LinearSVC or SVC(kernel='linear') from scikit-learn. The linear kernel is computationally efficient and sufficient for high-dimensional text data, often outperforming complex non-linear kernels (RBF, Polynomial) which are prone to overfitting in this domain.[14]

2.4.2 Comparison with Alternatives

- **Naive Bayes (MultinomialNB):** A strong baseline that assumes feature independence. While faster to train, it often underperforms SVMs when features are correlated (e.g., "haunted" and "house" appearing together).[17]
- **Random Forest:** Provides non-linearity and robustness but can be slow to predict with large numbers of trees and deep distinctions.
- **Deep Learning (LSTM/BERT):** While potentially more accurate, the training and inference latency are orders of magnitude higher, and the interpretability is lower. For an API demonstration, SVM offers the optimal balance of speed, accuracy, and interpretability.

2.4.3 The Training Pipeline Implementation

To ensure reproducibility and prevent data leakage, we utilize scikit-learn's Pipeline feature. This encapsulates the Vectorizer and the Classifier into a single object.

Python

```python
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC

# Define the pipeline
# Note: probability=True is essential for LIME integration later
text_clf = Pipeline()
```

By calling text_clf.fit(X_train, y_train), the pipeline learns the vocabulary from the training set and the decision boundaries of the SVM simultaneously. Crucially, when we later call text_clf.predict(X_test), the pipeline automatically applies the *exact same* tokenization and TF-IDF weighting (using the training set's IDF stats) to the test data.[17]

2.5 Serialization and Artifact Management

The final step of the pipeline phase is persistence. We use joblib (preferred over pickle for large numpy arrays) to save the trained pipeline to disk.[6] This binary file, typically named genre_classifier.joblib, contains:

1. The mapping of words to integer indices (Vocabulary).

2. The IDF weights for every word.
3. The support vectors and coefficients of the trained SVM.

This serialization effectively freezes the "knowledge" of the system, allowing it to be transported to the API server.

---

3. Phase 2: Implementing Explainability with LIME

The ability to predict a genre is useful; the ability to explain *why* is transformative. This section details the integration of **Local Interpretable Model-agnostic Explanations (LIME)**, a technique designed to illuminate the decision-making process of black-box models.

3.1 The Theoretical Framework of LIME

LIME operates on the intuition that explaining a model globally (understanding the entire complex hyperplane) is difficult, but explaining it locally (around a single prediction) is manageable. It assumes that in the immediate neighborhood of a data point, the decision boundary is locally linear.[2]

3.1.1 The Perturbation Mechanism

To explain a classification for a specific story $x$, LIME generates a synthetic dataset of perturbed samples $x'$. For text, this is achieved by randomly removing words from the original story.

- Original: "The dark knight rose."
- Perturbation 1: "The knight rose." (Removed "dark")
- Perturbation 2: "The dark." (Removed "knight", "rose")

3.1.2 Probing the Black Box

These perturbed samples are fed into the trained SVM pipeline to obtain probability scores.

- P(Horror | "The dark knight rose") = 0.85
- P(Horror | "The knight rose") = 0.40
- P(Horror | "The dark") = 0.60

By observing how the probability drops when specific words are removed, LIME infers the contribution of each word. If removing "dark" causes a massive drop in the "Horror" score, "dark" is assigned a high positive weight for that class.[20]

3.1.3 Local Surrogate Modeling

LIME weights these perturbed samples by their similarity to the original text (using a distance metric like Cosine Similarity or Jaccard Distance) and trains a simple, interpretable model—typically a sparse linear regression (Lasso)—on this weighted dataset. The coefficients of this linear model serve as the explanation.[21]

3.2 LIME vs. SHAP: A Critical Comparative Analysis

In a professional interview setting, distinguishing between LIME and SHAP (SHapley Additive exPlanations) demonstrates deep domain expertise. While both are surrogate methods, they serve different needs.[19]

Feature	**LIME**	**SHAP**
Foundation	Local Linear Approximation	Game Theory (Shapley Values)
Speed	Fast (Perturbation based)	Slow (Combinatorial complexity)
Consistency	Low (Random sampling can vary)	High (Guaranteed properties)
Output	"Word X caused Y"	"Feature X contribution to marginal probability"
Use Case	Real-time APIs, Text visualization	Scientific rigor, Global feature analysis

For this specific architecture—a real-time API for text—**LIME is superior** due to its computational efficiency and the intuitive nature of its text highlighting explanations. SHAP's exact computation is often NP-hard, requiring approximations (KernelSHAP) that can still be too slow for an interactive HTTP endpoint.[19]

3.3 Implementation Details: The LimeTextExplainer

We utilize the lime library's LimeTextExplainer. A key implementation detail is the interface between LIME and our Scikit-Learn pipeline. LIME requires a function that takes raw text strings and returns probability arrays.

Python

```
from lime.lime_text import LimeTextExplainer
```

```
# Initialize the explainer with class names from the pipeline
explainer = LimeTextExplainer(class_names=pipeline.classes_)

# The explanation process
exp = explainer.explain_instance(
    data_row=input_text,
    predict_fn=pipeline.predict_proba,
    num_features=10,
    num_samples=5000
)
```

3.3.1 Parameter Tuning

- num_features=10: We limit the explanation to the top 10 most influential words to prevent cognitive overload for the user.
- num_samples=5000: This controls the number of perturbations. Higher values yield more stable explanations (lower variance between runs) but increase latency linearly. 5,000 is a standard trade-off for text.[25]

3.3.2 Extracting Structure from Explanations

The exp object returned by LIME is complex. For API transmission, we must serialize it. The method .as_list() provides a list of tuples (word, weight), such as [('dragon', 0.45), ('space', -0.20)]. This is easily converted to JSON, unlike the visual HTML output .show_in_notebook().[26]

---

4. Phase 3: API Deployment Architecture with FastAPI

The final phase transforms our serialized model and explanation logic into a robust, scalable microservice. We select **FastAPI** for this task. Unlike legacy frameworks (Flask), FastAPI is built on Starlette and supports the ASGI (Asynchronous Server Gateway Interface) standard, making it ideal for modern high-concurrency applications.[3]

4.1 The Lifespan Event Pattern: Efficient Resource Management

A naive implementation might load the ML model inside the route function. This is catastrophic for performance:

1. **Latency:** Deserializing a 500MB model takes seconds. Doing this on every request destroys throughput.
2. **Memory:** Loading multiple copies of the model for concurrent requests will trigger Out-Of-Memory (OOM) errors.

We adopt the **Lifespan Context Manager** pattern (the modern replacement for startup events). This ensures the model is loaded exactly once into the application's memory when the worker process starts.[29]

Python

```python
from contextlib import asynccontextmanager
from fastapi import FastAPI
import joblib

ml_models = {}

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Load resources
    print("Loading Model Artifacts...")
    ml_models["pipeline"] = joblib.load("genre_classifier.joblib")
    yield
    # Clean up resources
    ml_models.clear()

app = FastAPI(lifespan=lifespan)
```

4.2 Pydantic: Strict Schema Validation

To ensure system reliability, we strictly type our inputs and outputs using **Pydantic** models. This acts as a contract between the client and server. If a user sends an integer instead of a string, or an empty string, Pydantic intercepts the request and returns a clear 422 Validation Error before the data ever touches the fragile ML pipeline.[28]

Python

```python
from pydantic import BaseModel, Field, field_validator
from typing import List, Tuple

class StoryRequest(BaseModel):
    text: str = Field(..., description="The story text to classify.", min_length=50)

    @field_validator('text')
    def check_content(cls, v):
        if v.strip() == "":
            raise ValueError('Text cannot be empty or whitespace only')
        return v

class ExplanationResponse(BaseModel):
    genre: str
    confidence: float
    explanation: List]
```

4.3 Concurrency Management: The Async/Sync Hazard

A critical engineering challenge arises when integrating CPU-bound ML tasks with FastAPI's async event loop.

4.3.1 The Event Loop Bottleneck

FastAPI runs on a single-threaded event loop. If we define our endpoint with async def, the code runs directly on this loop. Since pipeline.predict and lime.explain_instance are heavy CPU operations (taking seconds), calling them inside an async def function will **block** the loop. The server will become unresponsive to all other requests (even simple health checks) until the calculation finishes.[32]

4.3.2 The ThreadPool Solution

To mitigate this without setting up complex task queues (like Celery) for an MVP, we leverage FastAPI's internal thread handling. By defining the route with standard def (instead of async def), FastAPI automatically migrates the execution to a separate thread pool. This leaves the main event loop free to handle incoming connections while the heavy lifting happens in the background thread.[32]

Python

```
# Note: Using standard 'def' to offload CPU work to threadpool
@app.post("/predict", response_model=ExplanationResponse)
def predict_genre(request: StoryRequest):
    pipeline = ml_models["pipeline"]

    # Prediction (CPU Bound)
    pred_probs = pipeline.predict_proba([request.text])
    pred_label = pipeline.classes_[np.argmax(pred_probs)]
    confidence = np.max(pred_probs)

    # Explanation (Heavy CPU Bound)
    explainer = LimeTextExplainer(class_names=pipeline.classes_)
    exp = explainer.explain_instance(
        request.text,
        pipeline.predict_proba,
        num_features=5
    )

    return {
        "genre": pred_label,
        "confidence": float(confidence),
        "explanation": exp.as_list()
    }
```

4.4 Dockerization and Production Readiness

To ensure the API runs consistently across development and production environments, we containerize the application using **Docker**.
The Dockerfile must:

1. Start from a lightweight Python base image (python:3.9-slim).
2. Install system dependencies (gcc) often required for numpy/scikit-learn optimization.
3. Copy the requirements.txt and install packages.
4. Copy the app code and the *serialized model file*.
5. Entry point: Use uvicorn (an ASGI server) to run the FastAPI app.[28]

Dockerfile

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt
COPY..
# Expose port and run
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

5. Analysis of Limitations and Future Directions

While the architecture described is robust for a portfolio or MVP, professional transparency requires acknowledging its limitations.

1. **Latency:** LIME explanations are slow. In a high-throughput system, this synchronous /predict endpoint would be a bottleneck. The solution is to decouple prediction and explanation: return the prediction immediately, and queue the explanation task to a worker (Celery/Redis), delivering it later via WebSocket or a secondary endpoint.[36]
2. **Model Staticity:** The model is loaded from a static file. If the dataset changes, the application must be redeployed. A mature MLOps pipeline would pull the latest model version from a Model Registry (like MLflow) dynamically.
3. **Accuracy vs. Complexity:** SVMs are powerful, but Transformers (BERT) generally outperform them on semantic tasks. Future iterations could employ **Knowledge Distillation**, training a fast, simple model to mimic the output of a heavy Transformer, thereby keeping the speed required for LIME while capturing deeper semantic nuance.

6. Conclusion

The construction of an ML-driven Genre Classification API with Explainable AI is a multidimensional engineering challenge. It requires mastery of the data pipeline (handling sparsity and distribution), the modeling layer (SVM theory and serialization), the interpretability layer (LIME perturbation logic), and the deployment layer (FastAPI concurrency patterns).

By strictly adhering to the principles of reproducible preprocessing, selecting interpretable features (TF-IDF) where possible, and managing the event loop correctly, we create a system that is not only accurate but also transparent and scalable. This architecture serves as a definitive blueprint for engineers seeking to bridge the gap between black-box AI and human-centric software reliability.

7. Appendix: Implementation Reference Guide

7.1 Full Pipeline Training Script (train.py)

Python

```
import pandas as pd
import joblib
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

def train_pipeline():
    # 1. Ingest Data (Simulated loading from CSV)
    print("Loading data...")
    # Ideally load from [1] or [4] sources
    df = pd.read_csv("data/stories.csv")

    # 2. Split Data
    X_train, X_test, y_train, y_test = train_test_split(
        df['text'], df['genre'], stratify=df['genre'], random_state=42
    )

    # 3. Define Pipeline
    # Tfidf: remove english stopwords, allow bigrams
    # SVC: Linear kernel, probability=True for LIME
    pipeline = Pipeline()

    # 4. Train
    print("Training model...")
    pipeline.fit(X_train, y_train)

    # 5. Evaluate
    print("Evaluating...")
    print(classification_report(y_test, pipeline.predict(X_test)))
    # 6. Serialize
    print("Saving artifacts...")
    joblib.dump(pipeline, "genre_classifier.joblib")
    print("Done.")

if __name__ == "__main__":
    train_pipeline()
```

7.2 Full FastAPI Application (main.py)

Python

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from contextlib import asynccontextmanager
import joblib
from lime.lime_text import LimeTextExplainer
import numpy as np

# Global Model Store
ml_models = {}

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Load model on startup
    try:
        ml_models["pipeline"] = joblib.load("genre_classifier.joblib")
        print("Model loaded successfully.")
    except Exception as e:
        print(f"Failed to load model: {e}")
    yield
    ml_models.clear()

app = FastAPI(lifespan=lifespan)

# Pydantic Schemas
class StoryRequest(BaseModel):
    text: str

class Explanation(BaseModel):
    feature: str
    weight: float

class Response(BaseModel):
    genre: str
    confidence: float
    explanation: list[Explanation]

@app.post("/explain", response_model=Response)
def explain_story(request: StoryRequest):
    pipeline = ml_models.get("pipeline")
    if not pipeline:
        raise HTTPException(status_code=500, detail="Model not initialized")

    # Inference
    probas = pipeline.predict_proba([request.text])
    pred_idx = np.argmax(probas)
    genre = pipeline.classes_[pred_idx]
    confidence = probas[pred_idx]

    # LIME Explanation
    explainer = LimeTextExplainer(class_names=pipeline.classes_)
    exp = explainer.explain_instance(
        request.text,
        pipeline.predict_proba,
        num_features=5
    )

    # Format output
    exp_list = exp.as_list(label=pred_idx)
    formatted_exp = [{"feature": k, "weight": v} for k, v in exp_list]

    return {
        "genre": genre,
        "confidence": float(confidence),
        "explanation": formatted_exp
    }
```

Works cited

1. TinyStories - Kaggle, accessed on November 27, 2025, https://www.kaggle.com/datasets/thedevastator/tinystories-narrative-classification
2. marcotcr/lime: Lime: Explaining the predictions of any ... - GitHub, accessed on November 27, 2025, https://github.com/marcotcr/lime
3. Deploying ML Models as API using FastAPI - GeeksforGeeks, accessed on November 27, 2025, https://www.geeksforgeeks.org/machine-learning/deploying-ml-models-as-api-using-fastapi/
4. CMU Book Summary Dataset, accessed on November 27, 2025, https://www.cs.cmu.edu/~dbamman/booksummaries.html
5. CMU Book Summary Dataset - Kaggle, accessed on November 27, 2025, https://www.kaggle.com/datasets/ymaricar/cmu-book-summary-dataset
6. 1000 Stories 100 Genres - Kaggle, accessed on November 27, 2025, https://www.kaggle.com/datasets/fareedkhan557/1000-stories-100-genres
7. Book Genre Classification Dataset CSV Download Free, accessed on November 27, 2025, https://www.opendatabay.com/data/ai-ml/918efff4-07ae-47d3-9cf5-de98879aa091
8. Removing stop words with NLTK in Python - GeeksforGeeks, accessed on November 27, 2025, https://www.geeksforgeeks.org/nlp/removing-stop-words-nltk-python/
9. Getting rid of stop words and document tokenization using NLTK - Stack Overflow, accessed on November 27, 2025, https://stackoverflow.com/questions/17390326/getting-rid-of-stop-words-and-document-tokenization-using-nltk
10. A Comprehensive Guide to Text Preprocessing with NLTK - Codefinity, accessed on November 27, 2025, https://codefinity.com/blog/A-Comprehensive-Guide-to-Text-Preprocessing-with-NLTK
11. Removing NLTK Stopwords with Python - Vectorize, accessed on November 27, 2025, https://vectorize.io/blog/removing-nltk-stopwords-with-python
12. NLP Series: Day 4 — Stopword Removal and Normalization | by Ebrahim Mousavi | Medium, accessed on November 27, 2025, https://medium.com/@ebimsv/nlp-series-day-4-stopword-removal-and-normalization-418e0ec0016b
13. TfidfVectorizer — scikit-learn 1.7.2 documentation, accessed on November 27, 2025, https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
14. Text Classification using scikit-learn in NLP - GeeksforGeeks, accessed on November 27, 2025, https://www.geeksforgeeks.org/nlp/text-classification-using-scikit-learn-in-nlp/
15. How to plot the text classification using tf-idf svm sklearn in python - Stack Overflow, accessed on November 27, 2025,

https://stackoverflow.com/questions/50334915/how-to-plot-the-text-classification-using-tf-idf-svm-sklearn-in-python
16. Text classification with NB / SVM SciKit - how to get my text in? - Kaggle, accessed on November 27, 2025, https://www.kaggle.com/getting-started/75095
17. Working With Text Data — scikit-learn 1.4.2 documentation, accessed on November 27, 2025, https://scikit-learn.org/1.4/tutorial/text_analytics/working_with_text_data.html
18. FareedKhan-dev/NLP-1K-Stories-Dataset-Genres-100 - GitHub, accessed on November 27, 2025, https://github.com/FareedKhan-dev/NLP-1K-Stories-Dataset-Genres-100
19. Two minutes NLP — Explain predictions with LIME | by Fabio Chiusano | Generative AI, accessed on November 27, 2025, https://medium.com/nlplanet/two-minutes-nlp-explain-predictions-with-lime-aec46c7c25a2
20. 14 LIME – Interpretable Machine Learning, accessed on November 27, 2025, https://christophm.github.io/interpretable-ml-book/lime.html
21. Interpreting an NLP model with LIME and SHAP | by Kalia Barkai - Medium, accessed on November 27, 2025, https://medium.com/@krbarkai/interpreting-an-nlp-model-with-lime-and-shap-834ccfa124e4
22. LIME vs SHAP: A Comparative Analysis of Interpretability Tools - MarkovML, accessed on November 27, 2025, https://www.markovml.com/blog/lime-vs-shap
23. Comparison between SHAP (Shapley Additive Explanation) and LIME (Local Interpretable Model-Agnostic Explanations) - Cross Validated, accessed on November 27, 2025, https://stats.stackexchange.com/questions/379744/comparison-between-shap-shapley-additive-explanation-and-lime-local-interpret
24. LIME vs. SHAP: Which is Better for Explaining Machine Learning Models?, accessed on November 27, 2025, https://towardsdatascience.com/lime-vs-shap-which-is-better-for-explaining-machine-learning-models-d68d8290bb16/
25. Efficient way to generate Lime explanations for full dataset - Stack Overflow, accessed on November 27, 2025, https://stackoverflow.com/questions/70984947/efficient-way-to-generate-lime-explanations-for-full-dataset
26. Exploring LIME (Local Interpretable Model-agnostic Explanations) — Part 2 | by Sze Zhong LIM | Data And Beyond | Medium, accessed on November 27, 2025, https://medium.com/data-and-beyond/exploring-lime-local-interpretable-model-agnostic-explanations-part-2-760da4340fb0
27. lime package — lime 0.1 documentation, accessed on November 27, 2025, https://lime-ml.readthedocs.io/en/latest/lime.html
28. Creating a Secure Machine Learning API with FastAPI and Docker ..., accessed on November 27, 2025, https://machinelearningmastery.com/creating-a-secure-machine-learning-api-with-fastapi-and-docker/

29. Lifespan Events - FastAPI, accessed on November 27, 2025, https://fastapi.tiangolo.com/advanced/events/
30. Use a ml model loaded in the LIFESPAN function in the main file, in another file (ie: api router) #9234 - GitHub, accessed on November 27, 2025, https://github.com/fastapi/fastapi/discussions/9234
31. Using Pydantic for Dynamic LLM Response Models - Instructor, accessed on November 27, 2025, https://python.useinstructor.com/concepts/models/
32. Should every FastAPI route be async def? : r/Python - Reddit, accessed on November 27, 2025, https://www.reddit.com/r/Python/comments/1np3jz8/should_every_fastapi_route_be_async_def/
33. FastAPI - Why does synchronous code do not block the event Loop? - Stack Overflow, accessed on November 27, 2025, https://stackoverflow.com/questions/79382645/fastapi-why-does-synchronous-code-do-not-block-the-event-loop
34. Functions with "async def" will block the "def" functions · Issue #2488 - GitHub, accessed on November 27, 2025, https://github.com/tiangolo/fastapi/issues/2488
35. Using `async def` vs `def` in FastAPI and testing blocking calls [duplicate] - Stack Overflow, accessed on November 27, 2025, https://stackoverflow.com/questions/70123888/using-async-def-vs-def-in-fastapi-and-testing-blocking-calls
36. Running Deep Learning Models as Applications with FastAPI | by André Carvalho, accessed on November 27, 2025, https://revs.runtime-revolution.com/running-deep-learning-models-as-applications-with-fastapi-ecac57239e64