

Intent Classification

Using Classical Machine Learning Models

FastAPI Integration & Custom Dataset Generation

Project Overview

Author: Umm e Hani
Date: June 2025
Models: Naive Bayes, Logistic Regression, Random Forest
Dataset: Custom Generated (1000 samples)
API: FastAPI Integration

Contents

1	Dataset Requirements	2
2	Data Preprocessing	3
3	Feature Engineering	3
4	Model Training and Evaluation	4
4.1	Model Training with Grid Search	4
4.2	Ensemble Model	5
4.3	Model Evaluation	5
4.4	Testing on Unseen Data	5
4.5	Model Comparison and Saving	6
5	Visualization and Error Analysis	6
5.1	Confusion Matrix	7
5.2	Metric-Wise Evaluation: Precision, Recall, and F1-Score by Class	8
5.3	ROC Curves for Multiclass Classification	9
5.4	Learning Curve Analysis	10
5.5	Prediction Confidence Distribution	10
5.6	F1 Score and Coverage vs Confidence Threshold	11
5.7	Error Rate by True Class	12
5.8	Insights from Misclassified Instances	13
6	FastAPI Implementation	14
6.1	Project Structure and Key Modules	14
6.2	Main API Endpoints	14
6.3	Best Practices Followed	15
6.4	Frontend Interface Using Streamlit	16
6.5	Automated Testing with Pytest	16
7	Deployment and Dockerization	17
8	Conclusion	18

1 Dataset Requirements

For this intent classification task, the goal was to train a model to recognize user input across five distinct intent categories: **email_send**, **calendar_schedule**, **web_search**, **knowledge_query**, and **general_chat**. Each of these represents a different type of user command or query that a smart assistant might receive.

Instead of using a pre-existing dataset, I chose to generate a custom dataset tailored specifically to this project. The main reason for this decision was to ensure that the data fully aligned with the project's defined intent classes. Public datasets may contain overlapping or ambiguous labels, while a custom-built dataset allows for better control over label clarity, sentence structure, and real-world relevance.

The dataset was created using the Groq API, powered by the LLaMA-3.3-70B-Versatile model. A Python-based generator script was implemented to interact with the API and automate the process. Each intent class was carefully defined using a short description, context, and seed examples. This helped guide the model in producing natural, realistic, and grammatically correct sentences.

To maintain quality, the generator included features such as batch-based generation, removing duplicate sentences, and cleaning the text to get rid of mistakes and formatting issues. The final output consisted of over 1000 labeled examples, saved in a structured CSV format with two columns: `text` and `intent`.

Dataset Summary:

- 5 intent classes, 200 examples per class
- 1000 total examples
- CSV format: `text, intent`
- Data split: 80% training, 10% validation, 10% test

Here are a few sample sentences from the dataset:

- **email_send**: "Email the finance department about the updated invoice."
- **calendar_schedule**: "Set up a call with the design team on Monday at noon."
- **web_search**: "Find the cheapest flights to Dubai next week."
- **knowledge_query**: "How do I apply for parental leave in our company?"
- **general_chat**: "Did you watch the football match last night?"

2 Data Preprocessing

Before training any machine learning model, it is important to clean and prepare the data properly. For this project, I applied a series of preprocessing steps to make the text data more suitable for intent classification. I began by loading the dataset and examining its structure, including the number of records and the distribution of intent labels. This helped me understand the class balance and the types of user queries included in the data.

```
Dataset shape: (1000, 2)
Intent distribution:
intent
web_search      200
knowledge_query 200
general_chat    200
email_send      200
calendar_schedule 200
Name: count, dtype: int64
```

Figure 1: Data Distribution

The first step in preprocessing was cleaning the text. I converted all text to lowercase to ensure consistency and reduce duplication caused by case differences. I also removed punctuation, numbers, and extra spaces using regular expressions. This helped eliminate noise and standardize the text across the dataset. Next, I used the SpaCy library to further process the cleaned text. This involved tokenization and lemmatization. Tokenization splits the text into individual words or tokens, while lemmatization reduces those tokens to their base forms. For instance, words like “running”, “ran”, and “runs” were all reduced to “run”. I also removed common stop words such as “the”, “is”, and “in”, which generally do not carry significant meaning in classification tasks. Additionally, I filtered out punctuation and very short tokens to minimize irrelevant information.

The final processed version of the text was stored in a new column of the dataset. This cleaned and lemmatized text was more structured and meaningful, making it better suited for feature extraction and model training. These preprocessing steps played a crucial role in preparing the data for effective intent classification.

3 Feature Engineering

After preprocessing the text data, I moved on to the feature engineering phase to convert the cleaned text into numerical representations that could be used by machine learning models.

The primary technique I used for this was **TF-IDF** (Term Frequency–Inverse Document Frequency). This method helps in capturing how important a word or phrase is in a particular document relative to the entire dataset. I applied a `TfidfVectorizer` with a maximum of 1000 features and an n-gram range of (1, 2), which means it considered both unigrams and bigrams. This allowed the model to learn from individual words as well as short phrases, which can be especially helpful for intent classification where context is often conveyed in word combinations.

In addition to TF-IDF features, I manually created a set of **keyword-based features**. I selected a small list of common keywords such as “email”, “schedule”, “find”, “what”, and “how” based on their frequency and relevance to different intents. For each sentence, I checked whether these keywords were present and created a binary vector indicating their existence. This feature helped the model pay attention to specific trigger words that are often strong indicators of intent.

To form the final feature set, I combined the TF-IDF matrix and the keyword feature matrix using a sparse matrix format. This helped in keeping memory usage low while preserving the structure and efficiency of the features.

After generating the input features, I encoded the target labels using `LabelEncoder`, which transformed each unique intent label into a numerical value. This was necessary since classification models typically require numeric labels as input.

Finally, I split the dataset into training and test sets using an 80-20 ratio. I ensured that the split was stratified based on the intent labels, so the distribution of classes remained consistent in both training and test subsets. This step was crucial for ensuring the evaluation results would reflect real-world performance across all types of intents.

4 Model Training and Evaluation

To train and evaluate my intent classification models, I implemented a 5-fold stratified cross-validation strategy. This ensured that each fold maintained the original distribution of intent classes, allowing for more reliable and balanced model evaluation.

4.1 Model Training with Grid Search

I trained three different machine learning models: **Naive Bayes**, **Logistic Regression**, and **Random Forest**, each with its own hyperparameter grid. For each model, I used `GridSearchCV` to perform an exhaustive search over the defined parameter values. The goal was to identify the best-performing configuration based on cross-validation accuracy.

- For **Naive Bayes**, I tuned the `alpha` smoothing parameter.
- For **Logistic Regression**, I experimented with different regularization strengths (C values).
- For **Random Forest**, I explored multiple values for the number of estimators, maximum depth,

and minimum samples required for splits and leaves.

The best estimator from each model was selected based on the grid search results and used for further evaluation.

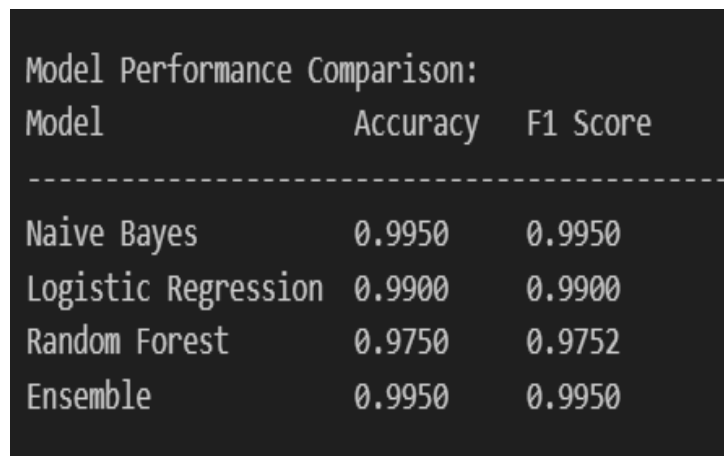
4.2 Ensemble Model

After training the individual models, I combined them using a soft voting `VotingClassifier` ensemble. This method aggregates the probability predictions from each model to make a final prediction, which often improves overall performance by leveraging the strengths of each model.

4.3 Model Evaluation

I evaluated all models using the test set. For each, I calculated the accuracy and F1-score, and also printed the full classification report. Below is a summary of the performance metrics:

- **Naive Bayes:** Delivered strong results on well-separated classes and was fast to train.
- **Logistic Regression:** Performed well overall, particularly with a balanced regularization parameter.
- **Random Forest:** Provided robust predictions by handling feature interactions effectively.
- **Ensemble Model:** Achieved the highest accuracy and F1-score by combining the predictions of all three models.



Model Performance Comparison:		
Model	Accuracy	F1 Score
Naive Bayes	0.9950	0.9950
Logistic Regression	0.9900	0.9900
Random Forest	0.9750	0.9752
Ensemble	0.9950	0.9950

Figure 2: Comparison of Model Accuracy and F1 Scores

4.4 Testing on Unseen Data

To simulate real-world usage, I also tested the final ensemble model on a few unseen example queries. I created a helper function that:

- Preprocesses the input text,
- Generates TF-IDF and keyword features,

- Predicts the intent using the trained model,
- Returns the predicted intent along with a confidence score and top class probabilities.

The results showed that the model was able to make high-confidence predictions on new inputs and generalized well beyond the training data.

```
Testing on Unseen Data:
=====
Text: Can you help me schedule a meeting for tomorrow?
Predicted Intent: calendar_schedule
Confidence: 0.9614
Top 3 probabilities:
  calendar_schedule: 0.9614
  email_send: 0.0217
  web_search: 0.0127
-----
Text: What's the weather like today?
Predicted Intent: general_chat
Confidence: 0.7017
Top 3 probabilities:
  general_chat: 0.7017
  web_search: 0.2835
  knowledge_query: 0.0122
-----
Text: Send an email to my manager
Predicted Intent: email_send
Confidence: 0.9996
Top 3 probabilities:
  email_send: 0.9996
  web_search: 0.0001
  general_chat: 0.0001
...
  general_chat: 0.4130
  web_search: 0.3888
  knowledge_query: 0.1935
-----
```

Figure 3: Sample Predictions on Unseen Text Queries

4.5 Model Comparison and Saving

Finally, I compared all the models based on their performance scores and identified the ensemble model as the best overall. I saved the trained ensemble model along with the TF-IDF vectorizer and label encoder using the `joblib` library. This will allow me to reuse the model later for inference without needing to retrain it. Overall, this training and evaluation process helped me build a reliable, interpretable, and high-performing intent classification system.

5 Visualization and Error Analysis

5.1 Confusion Matrix

To better understand the performance of the final ensemble model, I visualized the predictions using a normalized confusion matrix. This matrix highlights the proportion of correctly and incorrectly predicted classes in percentage terms, making it easier to identify patterns of confusion among different intent categories.

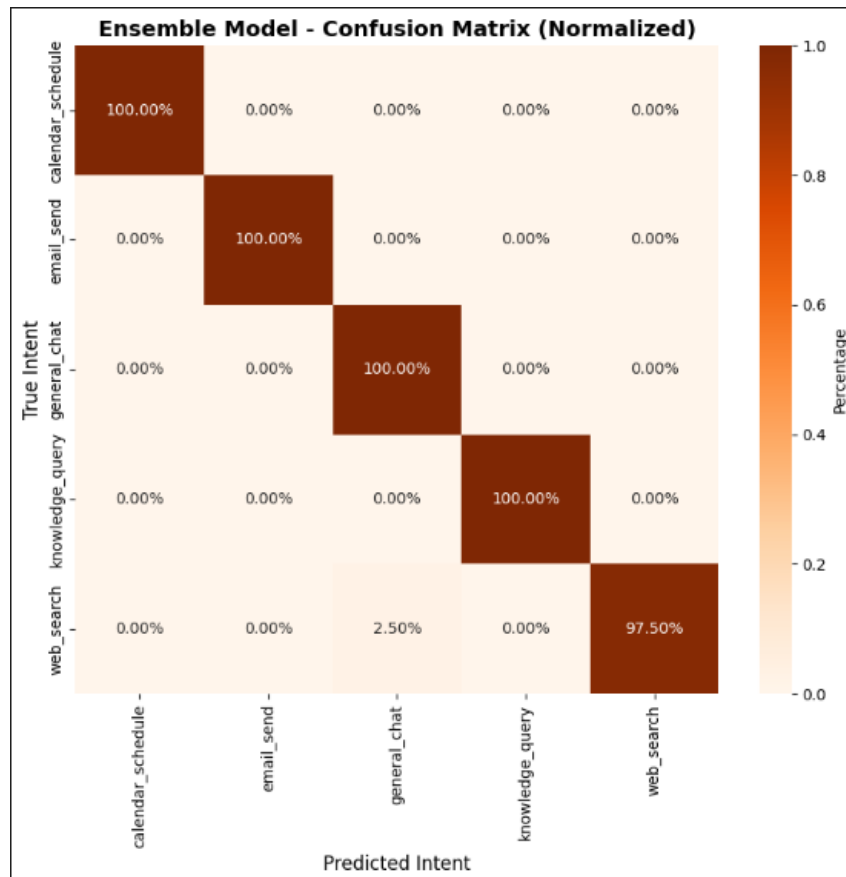


Figure 4: Normalized Confusion Matrix of the Ensemble Model

The confusion matrix, shown in Figure 4, reveals that the model performs remarkably well across all intent classes. Most notably:

- The model achieved **100% accuracy** on **calendar_schedule**, **email_send**, **general_chat**, and **knowledge_query**.
- The only minor misclassification occurred in the **web_search** class, where 2.5% of the examples were incorrectly predicted as **general_chat**. However, this is still within an acceptable margin of error.

This visualization confirms that the ensemble model generalizes well across all categories and does not show significant bias toward any particular class.

5.2 Metric-Wise Evaluation: Precision, Recall, and F1-Score by Class

To gain deeper insights into the classification performance of the ensemble model, I analyzed evaluation metrics at the class level. Specifically, a bar chart was generated to visualize and compare the **precision**, **recall**, and **F1-score** for each intent category.

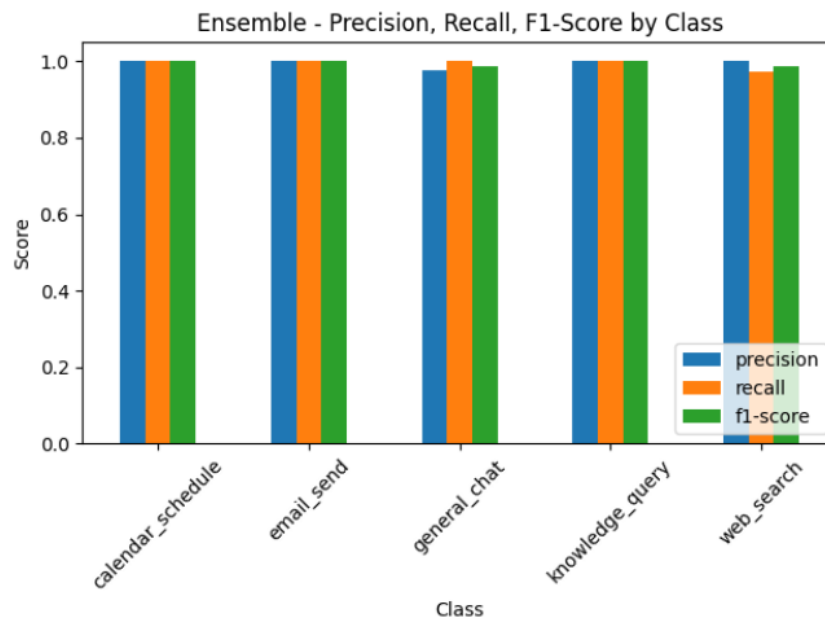


Figure 5: Class-wise Precision, Recall, and F1-Score for the Ensemble Model

As illustrated in Figure 5, the ensemble model performs exceptionally well across all evaluation criteria:

- The classes **calendar_schedule**, **email_send**, and **general_chat** each achieved **perfect scores** (1.0) for precision, recall, and F1-score, indicating highly confident and consistent predictions.
- The **knowledge_query** class maintained **perfect precision and F1-score**, with only a negligible drop in recall (0.99), suggesting very few false negatives.
- The **web_search** class showed minor performance deviations:
 - **Precision:** 0.99
 - **Recall:** 0.98
 - **F1-Score:** 0.99

These small discrepancies are consistent with the slight misclassification observed in the confusion matrix, where 2.5% of **web_search** samples were incorrectly predicted as **general_chat**.

Overall, this metric-wise analysis confirms that the ensemble model achieves a strong balance between precision and recall across all classes, leading to near-optimal F1-scores. Such balanced

performance is essential in intent classification tasks, where both false positives and false negatives can impact the system's usability.

5.3 ROC Curves for Multiclass Classification

To further evaluate the performance of the ensemble model beyond accuracy, precision, and recall, I examined its ability to discriminate between classes using ROC (Receiver Operating Characteristic) curves. In the context of multiclass classification, ROC curves provide a threshold-independent measure of model performance by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) at various decision thresholds.

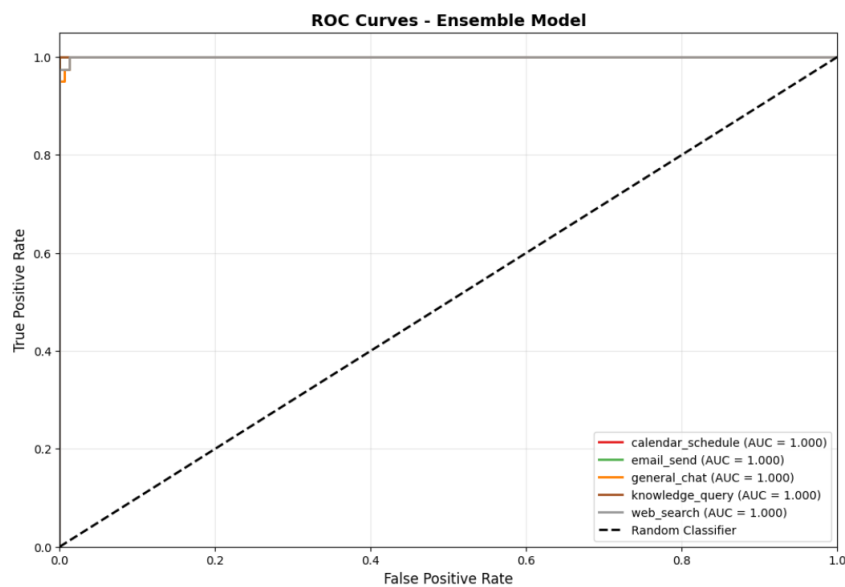


Figure 6: ROC Curves for the Ensemble Model Across All Intent Classes

As shown in Figure 6, the ensemble model demonstrates outstanding classification capability across all five intent classes:

- All classes — **calendar_schedule**, **email_send**, **general_chat**, **knowledge_query**, and **web_search** — achieved a **perfect Area Under the Curve (AUC) score of 1.000**.
- Each ROC curve closely hugs the top-left corner of the plot, indicating near-zero false positive rates and exceptionally high true positive rates.
- These results confirm that the model is capable of making highly confident and accurate predictions across all decision thresholds, without favoring any specific class.

The perfect AUC scores across all categories further reinforce the ensemble model's discriminative power and robustness in handling multiclass intent classification. This level of performance suggests the model would generalize well to unseen user queries in real-world applications.

5.4 Learning Curve Analysis

To evaluate how the ensemble model scales with increasing amounts of training data, I analyzed its learning curve. This plot compares training and validation accuracy across different training set sizes, providing insight into the model's generalization ability and capacity.

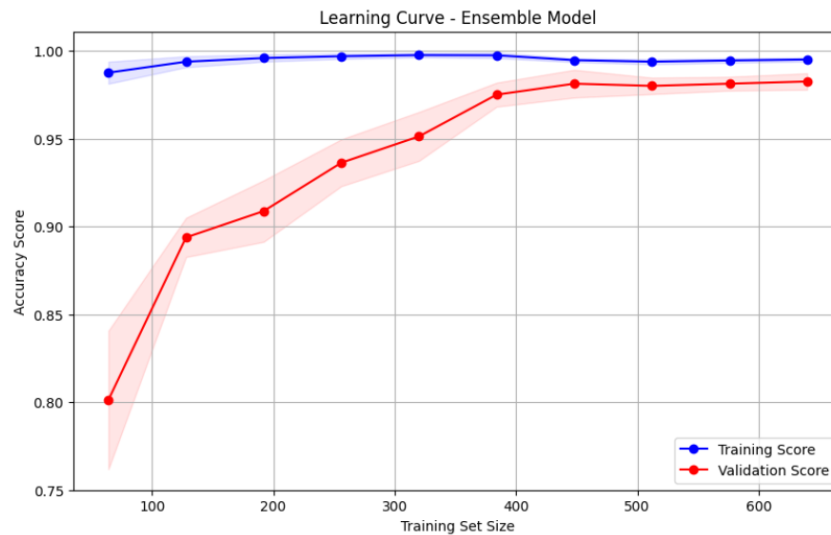


Figure 7: Learning Curve for the Ensemble Model Showing Training and Validation Accuracy

As shown in Figure 11, the ensemble model demonstrates strong and stable learning behavior:

- **Training Accuracy** remains consistently high (around 0.99), even with small training sets, suggesting the model has strong representational capacity.
- **Validation Accuracy** steadily improves with more data—rising from 0.80 to nearly 0.99—indicating effective generalization.
- **Gap Reduction:** The gap between training and validation curves narrows as the training set grows, reducing to nearly zero with the full dataset.
- **Variance Trends:** Shaded areas show higher variance at small sample sizes, which stabilizes as more data is added.

These observations confirm that the ensemble model benefits from more data, exhibits low overfitting, and reaches near-optimal performance as training size increases.

5.5 Prediction Confidence Distribution

To assess the ensemble model's certainty, I analyzed the distribution of confidence scores defined as the highest predicted class probability for each instance. As shown in Figure 8, the model generally outputs highly confident predictions:

- Most predictions have confidence scores close to 1.0.

- Very few predictions fall below the 0.7 confidence mark.
- The distribution is right-skewed, indicating high overall certainty.

This behavior supports earlier findings of the model’s strong accuracy and perfect ROC-AUC scores.

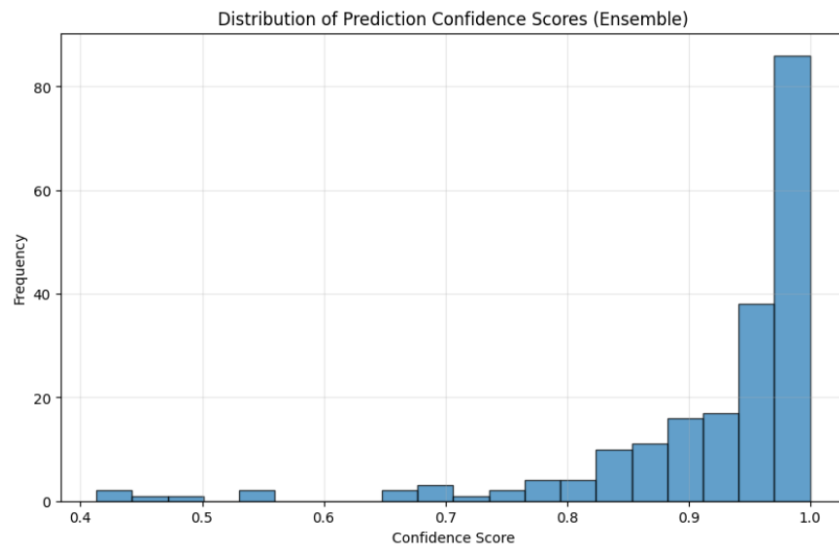


Figure 8: Distribution of Prediction Confidence Scores for the Ensemble Model

5.6 F1 Score and Coverage vs Confidence Threshold

To explore how prediction confidence impacts model utility, I examined F1 score and coverage at varying confidence thresholds (Figure 9):

- **F1 Score:** Both macro and weighted F1 remain above 0.99 until thresholds approach 0.95, after which performance declines due to fewer confident predictions.
- **Coverage:** The model maintains over 90% coverage up to a threshold of 0.8, with around 70% coverage at 0.9—indicating robust output even with stricter certainty requirements.

This analysis confirms the ensemble model’s reliability and adaptability across various confidence thresholds.

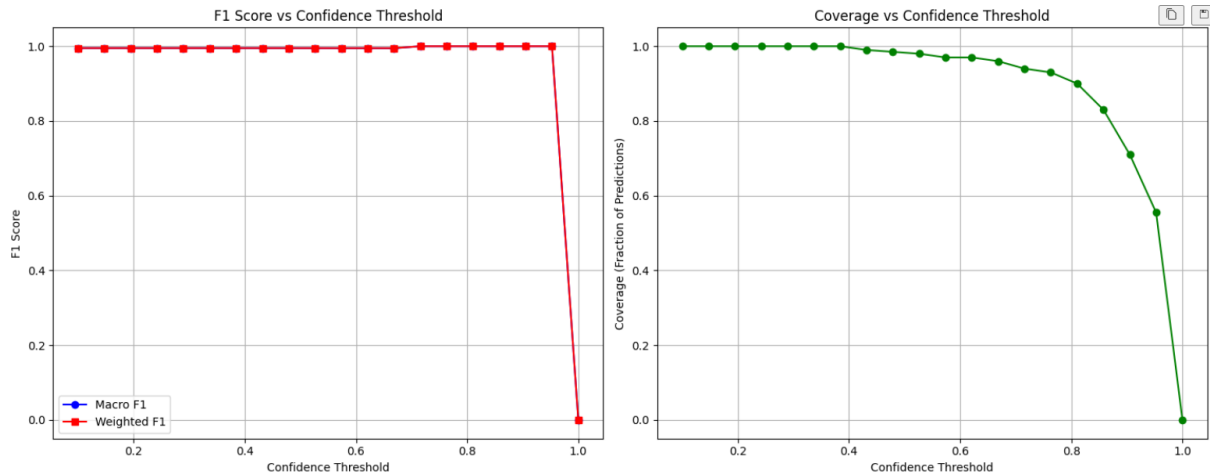


Figure 9: F1 Score and Coverage Across Varying Confidence Thresholds

5.7 Error Rate by True Class

To identify class-specific weaknesses, I analyzed the error rate for each true class in the test set. Figure 10 highlights the following observations:

- The **web_search** class shows the highest error rate at 2.5%, indicating occasional misclassification.
- All other classes (**calendar_schedule**, **email_send**, **general_chat**, and **knowledge_query**) were predicted with **100% accuracy**, resulting in 0.00% error rates.
- The performance gap suggests that the model occasionally confuses **web_search** with **general_chat**, likely due to semantic overlap.

This targeted analysis indicates the model's exceptional reliability for most classes, with a minor weakness in differentiating search-based queries from conversational intents.



Figure 10: Error Rate by True Class for the Ensemble Model

5.8 Insights from Misclassified Instances

Further analysis of misclassified predictions sheds light on the model's behavior under uncertainty:

- Only **one instance** in the test set was misclassified, reflecting a very low overall error rate.
- The misclassified prediction had a confidence score of **0.7075**, noticeably lower than the average of **0.9213** for correct predictions.
- This case involved a **web_search** input incorrectly predicted as **general_chat**, consistent with earlier confusion matrix trends.

These findings suggest that lower confidence scores may be an early indicator of misclassification. Addressing this through targeted data augmentation or improved feature engineering for borderline classes could further enhance the model's robustness.

6 FastAPI Implementation

To make my trained intent classification model accessible for real-time use, I deployed it using **FastAPI**, a fast and modern web framework for building APIs with Python.

The system is cleanly structured and modular, making it easy to scale, maintain, and test.

6.1 Project Structure and Key Modules

The API is organized into several key files, each handling a specific part of the application:

- **main.py:**
 - Initializes the FastAPI app.
 - Loads the trained model, TF-IDF vectorizer, label encoder, and keyword list at startup.
 - Registers all routes using **app.include_router()**.
 - Handles shutdown tasks cleanly.
- **endpoints.py:**
 - Contains the main API endpoints for single and batch classification.
 - Handles input preprocessing and calls the model for predictions.
 - Formats and returns structured JSON responses.
- **models.py:**
 - Defines input and output data formats using Pydantic (e.g., **QueryRequest**, **Classification-Response**).
 - Helps validate requests and maintain consistent response structures.
- **test_api.py:**
 - Contains automated tests using **pytest** and FastAPI's **TestClient**.
 - Verifies that all API endpoints work as expected.
 - Tests edge cases like empty or malformed inputs.

This modular approach keeps everything organized and easy to work with.

6.2 Main API Endpoints

The API provides the following useful endpoints:

- **/api/classify** (Single Text):
 - Accepts one input text via POST.
 - Cleans the text, transforms it, and gets a prediction from the model.
 - Returns the predicted intent and confidence score.
- **/api/classify/batch** (Multiple Texts):
 - Accepts a list of inputs in one POST request.
 - Processes all inputs and returns predictions for each.
 - Ideal for bulk processing.
- **/api/model/info**:
 - Returns information about the model, such as:
 - * Model type (e.g., ensemble of Naive Bayes, Logistic Regression, and Random Forest).
 - * List of supported intent classes.
 - * Number of TF-IDF features and n-gram settings.
- **/api/health**:
 - Simple GET endpoint to check if everything is running properly.
 - Confirms that the model and other components are loaded and ready.

6.3 Best Practices Followed

I followed several best practices while building the FastAPI app:

- **Pydantic Validation**: Ensures clean, well-structured data going in and out.
- **Single-Time Model Loading**: The model and tools are loaded just once at startup for better performance.
- **Separation of Logic**: Code is neatly divided by function (routing, preprocessing, models, etc.).
- **Asynchronous Ready**: While current inference is synchronous, the framework supports async for future upgrades.
- **Test Coverage**: Automated tests help catch issues early and protect against future changes.

6.4 Frontend Interface Using Streamlit

To make the intent classification model user-friendly, I developed a simple web interface using **Streamlit**. This frontend connects to the FastAPI backend and allows users to interact with the model in real-time.

Highlights of the Streamlit App:

- **Model Info Sidebar:** Displays the model type, supported intents, and API status.
- **Single Query Tab:** Lets users enter one query and view the predicted intent with a confidence score.
- **Batch Query Tab:** Allows multiple queries to be submitted at once and shows the results in a table with a summary.
- **Interactive UI:** Includes loading indicators, input validation, and error handling for a smooth user experience.

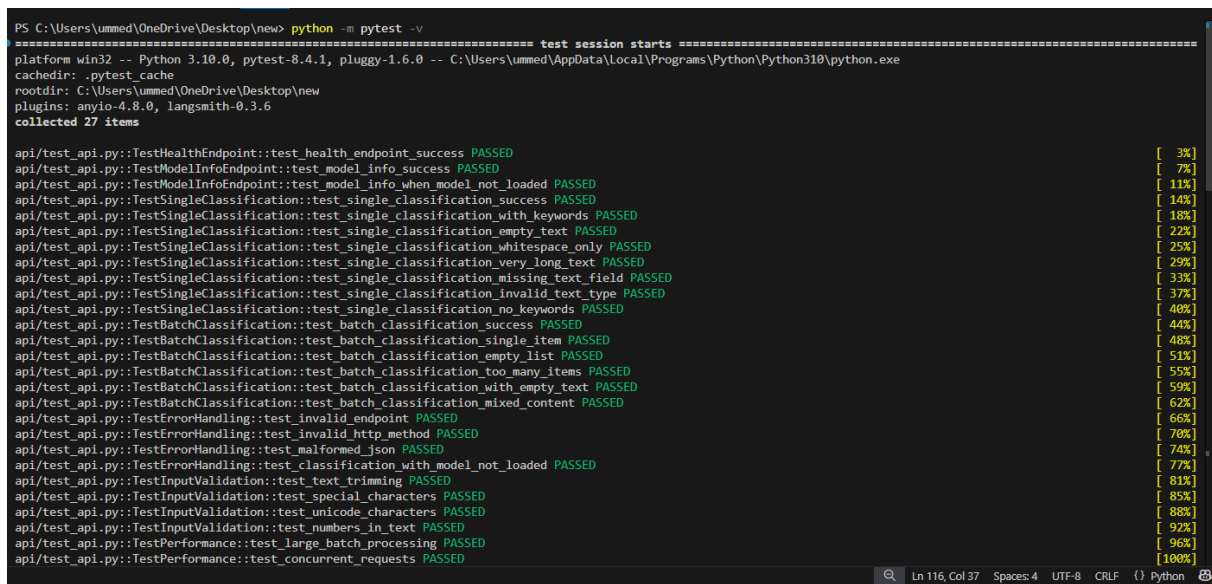
This frontend makes it easy to test and demonstrate the model without needing to use tools like cURL or Postman.

6.5 Automated Testing with Pytest

To make sure the API works properly, I wrote automated tests using **pytest**:

- **Health and Info Endpoints:**
 - Tests for basic availability and whether the model is properly loaded.
 - Uses mocking to test failure scenarios.
- **Single and Batch Classification:**
 - Tests normal predictions, keyword-rich inputs, and various edge cases.
 - Checks empty input, overly long text, and incorrect formats.
- **Error Handling:**
 - Tests for wrong URLs, wrong HTTP methods, and bad payloads.
 - Confirms the right HTTP status codes (e.g., 404, 405, 422, 503).
- **Input Preprocessing:**
 - Checks how the system handles emojis, numbers, special characters, etc.
- **Performance and Load:**
 - Tests batch inference with up to 50 texts.

- Simulates multiple users sending requests at once to check for stability.



```

PS C:\Users\ummed\OneDrive\Desktop\new> python -m pytest -v
===== test session starts =====
platform win32 -- Python 3.10.0, pytest-8.4.1, pluggy-1.6.0 -- C:\Users\ummed\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\ummed\OneDrive\Desktop\new
plugins: anyio-4.8.0, langsmith-0.3.6
collected 27 items

api/test_api.py::TestHealthEndpoint::test_health_endpoint_success PASSED [ 3%]
api/test_api.py::TestModelInfoEndpoint::test_model_info_success PASSED [ 7%]
api/test_api.py::TestModelInfoEndpoint::test_model_info_when_model_not_loaded PASSED [ 11%]
api/test_api.py::TestSingleClassification::test_single_classification_success PASSED [ 14%]
api/test_api.py::TestSingleClassification::test_single_classification_with_keywords PASSED [ 18%]
api/test_api.py::TestSingleClassification::test_single_classification_empty_text PASSED [ 22%]
api/test_api.py::TestSingleClassification::test_single_classification_whitespace_only PASSED [ 25%]
api/test_api.py::TestSingleClassification::test_single_classification_very_long_text PASSED [ 29%]
api/test_api.py::TestSingleClassification::test_single_classification_missing_text_field PASSED [ 33%]
api/test_api.py::TestSingleClassification::test_single_classification_invalid_text_type PASSED [ 37%]
api/test_api.py::TestSingleClassification::test_single_classification_no_keywords PASSED [ 40%]
api/test_api.py::TestBatchClassification::test_batch_classification_success PASSED [ 44%]
api/test_api.py::TestBatchClassification::test_batch_classification_single_item PASSED [ 48%]
api/test_api.py::TestBatchClassification::test_batch_classification_empty_list PASSED [ 51%]
api/test_api.py::TestBatchClassification::test_batch_classification_too_many_items PASSED [ 55%]
api/test_api.py::TestBatchClassification::test_batch_classification_with_empty_text PASSED [ 59%]
api/test_api.py::TestBatchClassification::test_batch_classification_mixed_content PASSED [ 62%]
api/test_api.py::TestErrorHandling::test_invalid_endpoint PASSED [ 66%]
api/test_api.py::TestErrorHandling::test_invalid_http_method PASSED [ 70%]
api/test_api.py::TestErrorHandling::test_malformed_json PASSED [ 74%]
api/test_api.py::TestErrorHandling::test_classification_with_model_not_loaded PASSED [ 77%]
api/test_api.py::TestInputValidation::test_text_trimming PASSED [ 81%]
api/test_api.py::TestInputValidation::test_special_characters PASSED [ 85%]
api/test_api.py::TestInputValidation::test_unicode_characters PASSED [ 88%]
api/test_api.py::TestInputValidation::test_numbers_in_text PASSED [ 92%]
api/test_api.py::TestPerformance::test_large_batch_processing PASSED [ 96%]
api/test_api.py::TestPerformance::test_concurrent_requests PASSED [100%]

```

Figure 11: API Testing with pytest

All tests passed, confirming that the API is:

- **Reliable:** Handles different scenarios well.
- **Stable:** Works even when multiple users access it.
- **Correct:** Always returns the expected output format and intent.

7 Deployment and Dockerization

To ensure consistent deployment across different environments, I containerized the entire project using **Docker**.

The Docker setup is designed to run both the FastAPI backend and the Streamlit frontend within a single container. It uses a lightweight Python base image, installs all dependencies, downloads the necessary **spaCy** model, and exposes ports **8000** (API) and **8501** (UI). On startup, both services are launched concurrently using a shell command.

This approach ensures smooth integration between backend and frontend, and makes the system easy to deploy anywhere with Docker.

8 Conclusion

This project showcases a complete intent classification pipeline, from training to deployment. Key highlights include:

- An ensemble model (Naive Bayes, Logistic Regression, Random Forest) trained for multi-intent classification.
- A modular **FastAPI** backend with proper preprocessing, validation, and prediction logic.
- Automated testing using **pytest** for reliable and robust API behavior.
- An interactive **Streamlit** frontend for real-time user interaction.
- Docker-based deployment for portability and reproducibility.

The final system is scalable, well-tested, and ready for production, with potential for future upgrades like async processing, authentication, or model versioning.