

Cuisine Classification from Ingredients

Exploratory Data Analysis and Multi-class Modeling

Anant Gupta and Meghender Pal

*Yardi School of AI
AIL 7024: Machine Learning*

November 27, 2025

Abstract

This report presents an end-to-end machine learning pipeline for predicting the cuisine of a recipe based solely on its list of ingredients. We work with the Cuisine Prediction Dataset, which contains tens of thousands of recipes labeled with 20 different cuisines. The project consists of four main stages: data loading and preprocessing, exploratory data analysis (EDA), feature engineering via a custom TF-IDF vectorizer, and multi-class classification using several models (Support Vector Classifier, custom Multinomial Logistic Regression, custom Decision Tree, custom Multinomial Naive Bayes, and XGBoost with hyperparameter tuning, Random Forest with hyperparameter tuning, LightGBM). We summarize and qualitatively compare the performance of these models, discuss their strengths and limitations, and outline possible extensions.

Contents

1	Introduction	3
2	Dataset Description	3
2.1	Source and Format	3
2.2	Basic Statistics	3
3	Exploratory Data Analysis (EDA)	4
3.1	Cuisine Distribution	4
3.2	Most Common Ingredients	4
3.3	Ingredients per Recipe	5
3.3.1	Cuisine Complexity Analysis (Box Plot)	5
3.3.2	Ingredient Co-occurrence Network	6
3.4	Word Clouds	7
3.5	Ingredient Co-occurrence Network	7
4	Feature Engineering	7
4.1	Text Preprocessing	7
4.2	Custom Tokenizer	7
4.3	Custom TF-IDF Vectorizer	8
4.4	Additional Numeric Feature	8
5	Experimental Setup	8
5.1	Train-Test Split	8
5.2	Models	8
5.2.1	Support Vector Classifier (SVC)	9
5.2.2	Multinomial Logistic Regression (Custom)	9
5.2.3	Custom Decision Tree	9
5.2.4	Custom Multinomial Naive Bayes	9
5.3	Random Forest Classifier (Ensemble Approach)	10
5.3.1	XGBoost with Grid Search	10
6	LightGBM Classifier	11
7	Results and Discussion	12
7.1	Evaluation Metric	12
7.2	Summary of Model Performance	12
7.3	Qualitative Observations	12
8	Limitations and Future Work	13
8.1	Limitations	13
8.2	Future Improvements	13
9	Project Contributions	13
10	Conclusion	14

1 Introduction

The goal of this project is to predict the cuisine type of a recipe given only its ingredients. Each recipe consists of:

- a unique recipe identifier,
- a list of textual ingredients,
- a cuisine label (for training data).

Automatically classifying cuisines has practical applications in:

- recipe recommendation systems,
- culinary analytics (e.g., studying ingredient usage patterns across cultures),
- assisting users in tagging or organizing large recipe collections.

The problem is formulated as a multi-class text classification task, where each recipe is represented by a bag-of-ingredients, and the target variable is one of 20 cuisines.

2 Dataset Description

2.1 Source and Format

The data is provided as JSON files:

- `train.json`: labeled recipes with fields `id`, `cuisine`, and `ingredients`,
- `test.json`: unlabeled recipes with fields `id` and `ingredients`.

Each training sample has:

- `id`: integer recipe identifier,
- `cuisine`: string label (*italian*, *mexican*, *indian*, etc.),
- `ingredients`: list of strings, each being a free-form ingredient name.

In the notebook, the data is loaded using:

```
with open("/content/train.json") as f:  
    data = json.load(f)  
df = pd.DataFrame(data)
```

A similar process is used for `test.json`. The resulting training DataFrame has 3 main columns: `id`, `cuisine`, and `ingredients` (a list).

2.2 Basic Statistics

The notebook inspects the dataset using standard pandas methods:

- `df.shape` to obtain the number of recipes and columns,
- `df.info()` to check dtypes and missing values,
- `df.head()` to visually inspect a few rows.

The number of classes is computed via:

```
len(df["cuisine"].value_counts())
```

which confirms that there are 20 distinct cuisines.

The dataset exhibits class imbalance: some cuisines (e.g., *italian*, *mexican*, *southern_us*) have many more examples than others (e.g., *brazilian*). This affects model performance and should be kept in mind when interpreting results.

3 Exploratory Data Analysis (EDA)

The notebook performs several EDA steps to understand the distribution of cuisines and ingredients and to build intuition for the classification task.

3.1 Cuisine Distribution

A bar plot of cuisine frequencies is generated using:

```
cuisine_counts = df["cuisine"].value_counts()  
cuisine_counts.plot(kind="bar")
```

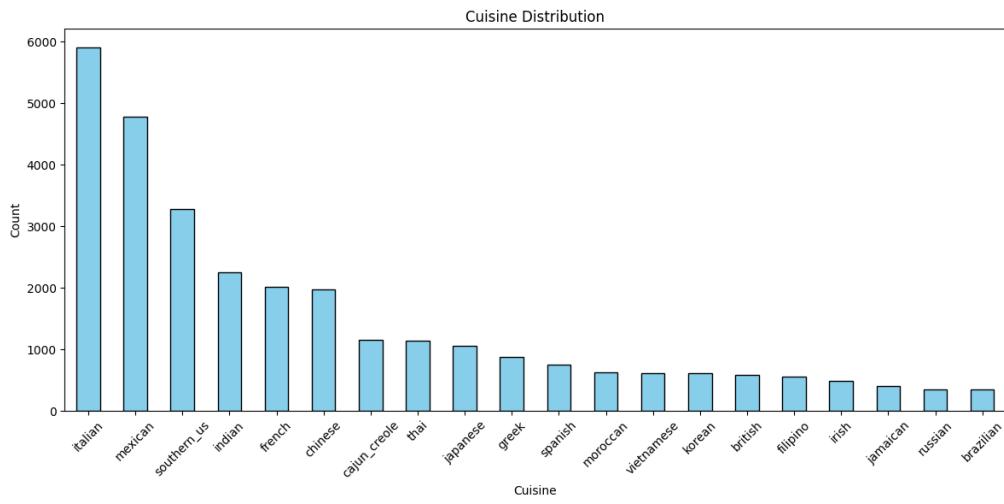


Figure 1: Distribution of cuisines in the training set. (Export this plot from the notebook.)

We clearly observe that a few cuisines dominate the dataset, which may bias models toward predicting those majority classes.

3.2 Most Common Ingredients

To analyze ingredient frequencies, all ingredients are flattened and counted:

```
all_ingredients = [ing.lower() for recipe in df["ingredients"]  
                  for ing in recipe]  
ingredient_counts = Counter(all_ingredients)  
top_ingredients = ingredient_counts.most_common(20)
```

A bar plot of the top 20 ingredients is created:

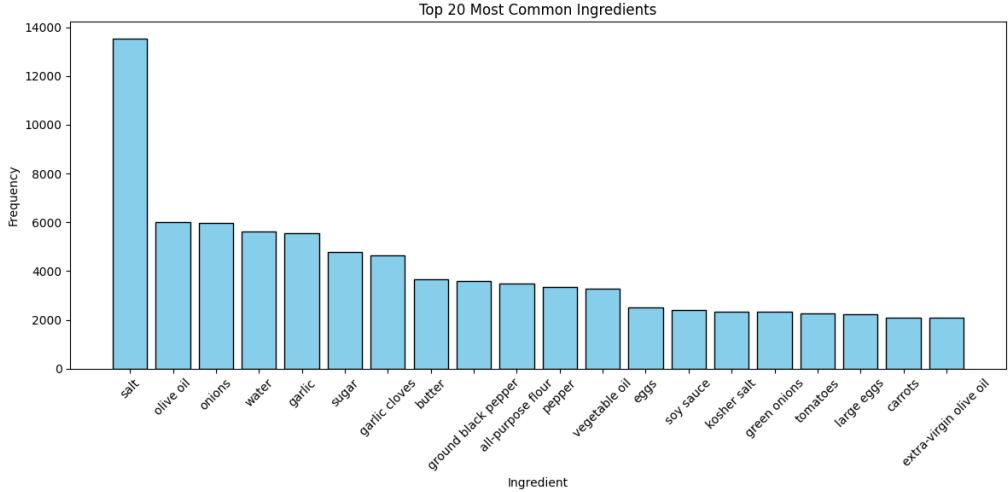


Figure 2: Top 20 most frequent ingredients across all cuisines.

Unsurprisingly, very generic ingredients such as salt, sugar, onions, oil, etc., appear frequently across many cuisines.

3.3 Ingredients per Recipe

The number of ingredients per recipe is computed and visualized:

```
df["num_ingredients"] = df["ingredients"].apply(len)
plt.hist(df["num_ingredients"], bins=20)
```

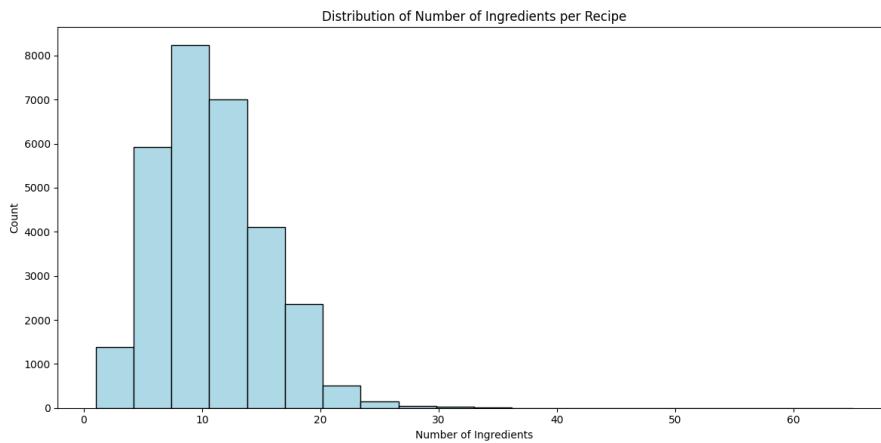


Figure 3: Histogram of the number of ingredients per recipe.

This shows the typical range and distribution of recipe lengths (in terms of ingredient count).

Additionally, a boxplot of the number of ingredients per cuisine is created to see whether certain cuisines systematically use more ingredients than others.

3.3.1 Cuisine Complexity Analysis (Box Plot)

We analyzed the distribution of ingredient counts per recipe to understand the complexity of different cuisines.

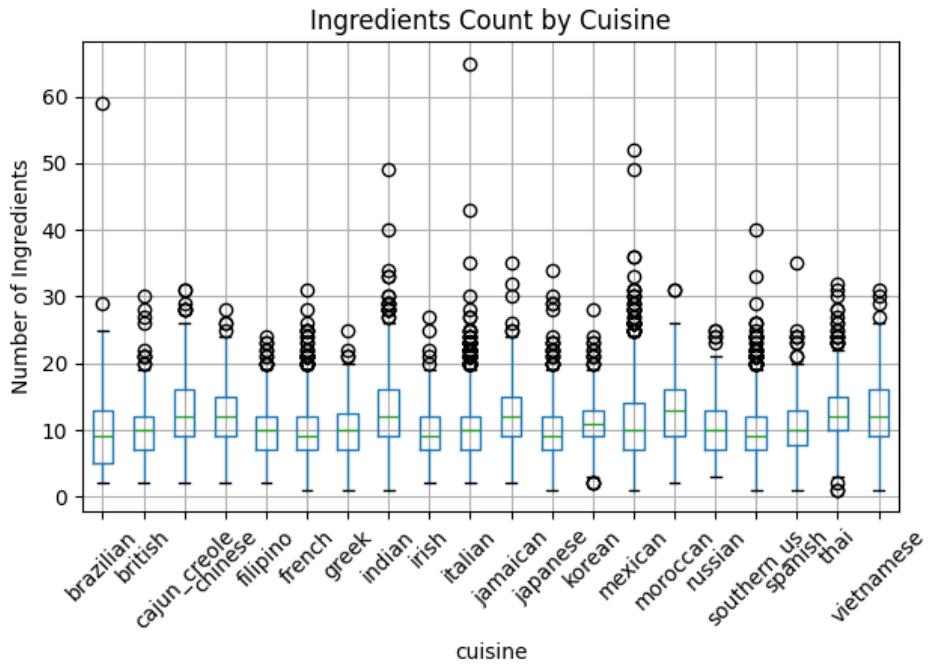


Figure 4: Box Plot showing Ingredient Counts across Cuisines

Insight: As shown in Figure 4, **Indian** and **Moroccan** cuisines are statistically more complex (Median \approx 12-15 ingredients) compared to Western cuisines like **Irish** and **British** (Median \approx 8-9 ingredients).

3.3.2 Ingredient Co-occurrence Network

To visualize the relationship between ingredients, we plotted a network graph where nodes are ingredients and edges represent co-occurrence in recipes.

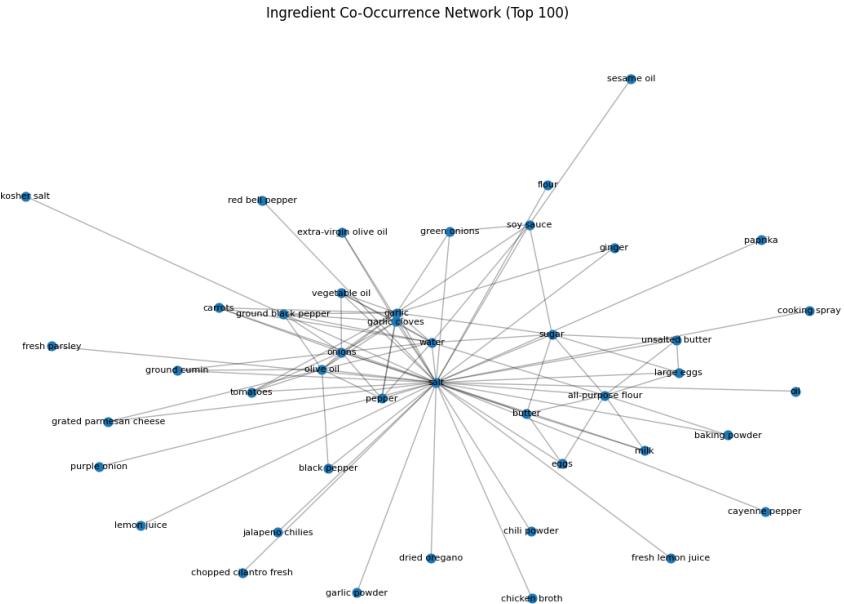


Figure 5: Network Analysis of Top Ingredients

Observation: Ingredients like *Garlic*, *Onion*, and *Oil* act as central "hubs", connecting disparate clusters of cuisines.

3.4 Word Clouds

For visualization, ingredients for each recipe are concatenated into a single string:

```
df["ingredients_str"] = df["ingredients"].apply(  
    lambda x: " ".join([ing.lower().replace(" ", "_") for ing in x])  
)  
all_ingredients = " ".join(df["ingredients_str"].tolist())
```

A global word cloud is constructed using the `WordCloud` library, and separate word clouds are generated for the top few cuisines (e.g., 6 most frequent cuisines). These visualizations highlight typical ingredient patterns per cuisine.

3.5 Ingredient Co-occurrence Network

To capture relationships between ingredients, a co-occurrence network is built:

- For each recipe, all unordered pairs of ingredients are counted.
- The top co-occurring pairs are selected.
- A graph is constructed with ingredients as nodes and co-occurrences as weighted edges.

Using NetworkX, the notebook plots this graph with a spring layout and visualizes the strongest ingredient ties, yielding insights like groups of ingredients that frequently appear together.

4 Feature Engineering

4.1 Text Preprocessing

The main text preprocessing steps are:

- Convert each ingredient to lowercase.
- Replace spaces within an ingredient by underscores to treat multi-word ingredients as single tokens (e.g., "olive oil" → "olive_oil").
- Concatenate all ingredient tokens for a recipe into a single string, stored in `ingredients_str`.

This creates a bag-of-ingredients representation that is suitable for vectorization.

4.2 Custom Tokenizer

A custom `Tokenizer` class is implemented to:

- compute document frequency (DF) for each token,
- apply document frequency thresholds:
 - `min_df`: minimum number of documents in which a token must appear,
 - `max_df`: maximum fraction of documents (relative to the total) in which a token can appear,
- build a vocabulary mapping each valid token to an index.

4.3 Custom TF-IDF Vectorizer

A `CustomTfidfVectorizer` class is implemented using the custom tokenizer. Let N be the number of documents, df_j be the document frequency of term j , and tf_{ij} be the raw count of term j in document i . The TF-IDF representation is defined as:

$$\text{idf}_j = \log \frac{1 + N}{1 + df_j} + 1, \quad (1)$$

$$\text{tfidf}_{ij} = \text{tf}_{ij} \cdot \text{idf}_j, \quad (2)$$

optionally with sublinear TF scaling:

$$\text{tf}'_{ij} = \begin{cases} 1 + \log(\text{tf}_{ij}) & \text{if } \text{tf}_{ij} > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

Internally, the vectorizer builds a sparse matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ using `scipy.sparse.csr_matrix`, where n is the number of recipes and d is the vocabulary size. In the notebook, the TF-IDF matrix is built as:

```
custom_tokenizer = Tokenizer(min_df=5, max_df=0.9)
custom_vectorizer = CustomTfidfVectorizer(tokenizer=custom_tokenizer)
X_custom_tfidf = custom_vectorizer.fit_transform(df["ingredients_str"])
```

4.4 Additional Numeric Feature

The number of ingredients per recipe (`num_ingredients`) is appended as an additional feature. After dropping the raw list-of-ingredients column, we construct:

$$\mathbf{X} = [\mathbf{X}_{\text{tfidf}} \mid \mathbf{x}_{\text{num_ingredients}}], \\ \mathbf{y} = \text{cuisine},$$

where concatenation is done via:

```
from scipy.sparse import hstack
X = hstack([X_custom_tfidf, df_new[["num_ingredients"]]])
y = df_new["cuisine"]
```

5 Experimental Setup

5.1 Train-Test Split

We perform a single train-test split using scikit-learn:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=seed
)
```

Thus, 75% of the data is used for training and 25% for testing, with a fixed random seed for reproducibility.

5.2 Models

The notebook trains and/or implements the following models:

5.2.1 Support Vector Classifier (SVC)

The SVC model is defined as:

```
from sklearn.svm import SVC
model = SVC(kernel="rbf", C=2, random_state=seed, verbose=False)
model.fit(X_train, y_train)
```

Evaluation uses `model.score` on both training and test sets:

```
print(model.score(X_train, y_train)) # Train accuracy
print(model.score(X_test, y_test)) # Test accuracy
```

5.2.2 Multinomial Logistic Regression (Custom)

A custom multinomial logistic regression is implemented from scratch using softmax and cross-entropy loss. For a feature vector $\mathbf{x}_i \in \mathbb{R}^d$ and weights $\mathbf{W} \in \mathbb{R}^{K \times d}$ and biases $\mathbf{b} \in \mathbb{R}^K$ (for K classes), the model defines:

$$\mathbf{z}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}, \quad (4)$$

$$p(y = k \mid \mathbf{x}_i) = \frac{\exp(z_{ik})}{\sum_{l=1}^K \exp(z_{il})}. \quad (5)$$

The cross-entropy loss over a batch is:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \log p(y_i \mid \mathbf{x}_i), \quad (6)$$

where m is the number of examples in the batch. Parameters are updated using gradient descent with a specified learning rate and number of iterations.

5.2.3 Custom Decision Tree

A custom decision tree classifier is implemented with:

- Node structure (`Node` class) storing feature index, threshold, children, and leaf label.
- Gini impurity as the splitting criterion:

$$G(S) = 1 - \sum_k p_k^2, \quad (7)$$

where p_k is the proportion of class k in node S .

- Hyperparameters like maximum depth, minimum samples to split, and minimum samples per leaf.
- A manual grid search over a small hyperparameter grid (e.g., max depth and min samples split/leaf).

The tree recursively selects the split (feature index and threshold) that maximizes impurity reduction until stopping conditions are met.

5.2.4 Custom Multinomial Naive Bayes

A custom multinomial Naive Bayes classifier is implemented for count-based features. For each class c , we estimate:

- the prior:

$$P(C = c) = \frac{N_c}{N}, \quad (8)$$

where N_c is the number of training samples in class c ,

- the conditional probability of term j given class c with Laplace smoothing (parameter α):

$$P(w_j | C = c) = \frac{N_{c,j} + \alpha}{\sum_{j'} N_{c,j'} + \alpha \cdot V}, \quad (9)$$

where $N_{c,j}$ is the total count of term j in class c and V is the vocabulary size.

In log-space, the prediction for a document is:

$$\log P(C = c | \mathbf{x}) \propto \log P(C = c) + \sum_j x_j \log P(w_j | C = c), \quad (10)$$

and the class with maximum log posterior is chosen.

The notebook performs a manual grid search over several α values and prints the test accuracy for each, selecting the best one.

5.3 Random Forest Classifier (Ensemble Approach)

Since the single Decision Tree performed poorly (44.5% accuracy) due to high dimensionality, we implemented **Random Forest**, which utilizes the concept of **Bagging (Bootstrap Aggregating)**.

- **Hypothesis:** Aggregating hundreds of weak learners (trees) will reduce variance and prevent the model from getting lost in sparse features.
- Random Forest Analysis: Random Forest significantly improved upon the single Decision Tree (44.5% \rightarrow 70.0%), but still these results indicating that tree-based ensembles struggle to generalize on this specific sparse, high-dimensional dataset compared to linear models like SVM

5.3.1 XGBoost with Grid Search

Finally, the notebook uses XGBoost with scikit-learn's `GridSearchCV` for hyperparameter tuning. Dense feature matrices are created:

```
X_train_dense = X_train.toarray()
X_test_dense = X_test.toarray()
```

Labels are encoded as integers:

```
from sklearn.preprocessing import LabelEncoder
lb = LabelEncoder()
y_train_encoded = lb.fit_transform(y_train)
y_test_encoded = lb.transform(y_test)
```

The XGBoost classifier and parameter grid are defined as:

```
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV

xgb_model = XGBClassifier(
    objective='multi:softmax',
```

```

        eval_metric='mlogloss',
        use_label_encoder=False,
        random_state=seed,
        n_jobs=-1
    )

param_grid = {
    'n_estimators': [100],
    'learning_rate': [0.1, 0.05],
    'max_depth': [3, 4]
}

```

Grid search is then run using accuracy as the scoring metric and 3-fold cross-validation:

```

grid_search = GridSearchCV(
    estimator=xgb_model,
    param_grid=param_grid,
    scoring='accuracy',
    cv=3,
    verbose=4,
    n_jobs=-1
)
grid_search.fit(X_train_dense, y_train_encoded)

```

The best hyperparameters and cross-validation accuracy are obtained from `grid_search.best_params_` and `grid_search.best_score_`.

6 LightGBM Classifier

Model Code:

```

from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score

# Initialize LGBMClassifier
# Using objective='multiclass' and num_class=len(lb.classes_) for multi-class classification
lgbm_model = LGBMClassifier(objective='multiclass', num_class=len(lb.classes_), random_state=seed)

# Train the LightGBM model
print("Training LightGBM model...")
lgbm_model.fit(X_train_dense, y_train_encoded)
print("LightGBM training complete.")

# Make predictions
y_train_pred_lgbm = lgbm_model.predict(X_train_dense)
y_test_pred_lgbm = lgbm_model.predict(X_test_dense)

# Calculate accuracies
train_accuracy_lgbm = accuracy_score(y_train_encoded, y_train_pred_lgbm)
test_accuracy_lgbm = accuracy_score(y_test_encoded, y_test_pred_lgbm)

print(f"\nLightGBM Training Accuracy: {train_accuracy_lgbm:.4f}")
print(f"LightGBM Test Accuracy: {test_accuracy_lgbm:.4f}")

```

Model Performance:

- LightGBM Training Accuracy: **0.9332**
- LightGBM Test Accuracy: **0.7567**

7 Results and Discussion

7.1 Evaluation Metric

All models are primarily evaluated using **classification accuracy** on the held-out test set:

$$\text{Accuracy} = \frac{\# \text{ correctly classified test examples}}{\# \text{ total test examples}}. \quad (11)$$

In the notebook, for the SVC, accuracy is directly computed using `model.score`, while for custom models it is computed as the fraction of matching labels:

```
accuracy = np.mean(y_pred == y_test)
```

7.2 Summary of Model Performance

A qualitative summary of expected behavior is as follows:

- The SVC with RBF kernel typically performs strongly on high-dimensional sparse data when tuned appropriately.
- Multinomial logistic regression is also a strong baseline for text classification.
- Naive Bayes is simple, fast, and often competitive, but can be outperformed by discriminative models.
- Decision trees tend to overfit if not regularized, especially with many features.
- XGBoost (gradient-boosted trees) usually provides strong performance by capturing non-linear interactions.

Table 1: Comparison of model performance on the test set.

Model	Notes	Test Accuracy
SVC (RBF kernel, $C = 2$)	TF-IDF + #ingredients	0.7778
Custom Multinomial Logistic Regression	TF-IDF + #ingredients	0.7774
Custom Decision Tree	Grid search on depth and min split	0.3970
Random Forest	Bagging	0.7003
Custom Multinomial Naive Bayes	Best $\alpha = 10.0$	0.7484
XGBoost	TF-IDF + #ingredients	0.7243
LightBGM	TF-IDF + #ingredients	0.7573

7.3 Qualitative Observations

From the EDA and modeling pipeline, we can make several qualitative observations:

- Certain cuisines (e.g., *indian*, *thai*, *japanese*) have highly distinctive ingredients (e.g., *garam_masala*, *fish_sauce*, *soy_sauce*), which help models to classify them correctly.
- More generic ingredients (e.g., *salt*, *oil*) contribute little discriminative power and appear across many cuisines.
- Classes with fewer samples (e.g., *brazilian*) are harder to classify, and misclassifications often occur into more common cuisines.

- Models that can exploit non-linear relationships (e.g., SVC with RBF kernel, XGBoost) are expected to outperform simpler models such as Naive Bayes or shallow decision trees, at the cost of more computation.

8 Limitations and Future Work

8.1 Limitations

Several limitations of the current pipeline are:

- **Sparse high-dimensional features:** TF-IDF produces very large feature vectors. Converting to dense arrays (as done before XGBoost) can be memory intensive.
- **Simple preprocessing:** The preprocessing step only lowercases and replaces spaces with underscores. It does not perform stemming/lemmatization, synonym handling, or spelling normalization.
- **Class imbalance:** Imbalanced label distribution may bias the models towards majority cuisines.
- **Single train-test split:** Using a single random split may not fully or robustly evaluate model performance; k -fold cross-validation or multiple runs could provide more stable estimates.

8.2 Future Improvements

Possible extensions include:

- Using character or word n -grams instead of only unigrams.
- Applying lemmatization or ingredient normalization (e.g., mapping `olive_oil` and `extra_virgin_olive_oil` to a shared concept).
- Trying linear models with appropriate regularization (e.g., linear SVM, logistic regression with L2 penalty) directly on sparse TF-IDF.
- Employing class-weighted loss functions or resampling strategies to mitigate class imbalance.
- Exploring embedding-based models (e.g., Word2Vec, fastText) or deep learning architectures for further performance gains.

9 Project Contributions

This project was a collaborative effort. Individual contributions are outlined below.

Anant Gupta (2025AIB2556)

- **Preprocessing Pipeline:** Designed the Custom Tokenizer to handle multi-word ingredients and implemented TF-IDF vectorization.
- **Sklearn Models:** Implemented and trained SVM for high level comparison with scratch implementations.
- **Scratch Implementations:** Developed the core logic for Naive Bayes, and Decision Tree from scratch.
- **Ensemble Modeling:** Advanced models implemented and tuned including Random Forest, XGBoost, and LightGBM.

- **Model Evaluation:** Conducted comparative analysis (Train vs Test accuracy) and generated the final leaderboard.
- **Report Documentation:** Authored the report.

Meghender Pal (2025AIB2563)

- **Exploratory Data Analysis (EDA):** Conducted a detailed analysis of class imbalance and ingredient distributions.
- **Visualizations:** Generated Box Plots for recipe complexity and Network Graphs for ingredient co-occurrence.
- **Scratch Implementations:** Developed the core logic for Logistic Regression from scratch.
- **Ensemble Modeling:** Advanced model Random Forest implemented and tuned
- **Presentation PPT.** Authored the PPT.

10 Conclusion

This project implemented a complete pipeline for cuisine classification using ingredient lists as textual input. Starting from raw JSON data, we performed extensive EDA, engineered a custom TF-IDF representation augmented with a simple numeric feature, and implemented or trained multiple classification models, including custom implementations of logistic regression, decision tree, and multinomial Naive Bayes, as well as SVC and XGBoost with hyperparameter tuning.

The framework demonstrates the full lifecycle of a classical machine learning project:

1. data understanding,
2. feature engineering,
3. model design and training,
4. evaluation and comparison,
5. identification of limitations and future improvements.

With further tuning, richer features, and more advanced models, it is possible to significantly improve performance on this task and better capture the complex structure of culinary styles across cuisines.