Food Item Classification

CSC311: Final Project Report

Aamid Mohsin, Imad Syed, Mohsin Muzammil, Mahmoud Zeidan

Date: April 4, 2025

Abstract

In this report, we present our approach to classifying three food items—Pizza, Shawarma, and Sushi—using supervised machine learning. Our dataset consists of survey responses capturing various attributes, including perceived complexity, ingredient count, serving setting, expected price, associated movies, drink pairings, personal associations, and hot sauce preferences. We explore multiple classification models, including k-nearest neighbors, decision trees, logistic regression, multilayer perceptrons, random forests, naive bayes, and extremely randomized trees. Our results show that the ExtraTreesClassifier consistently achieved accuracies above 90%. We discuss dataset composition, data splitting strategies, feature engineering techniques, model exploration and evaluation, and the rationale for selecting Extra Trees as our final model.

0 Contents

1 Introduction

Problem Statement

The objective of this project is to develop a machine learning model that can accurately classify a food item–Pizza, Shawarma, or Sushi–based on survey responses. Our focus is on creating a generalizable model that performs well on unseen data.

Goals

Our goals for this project are:

- Achieve top performance on the teaching team's unseen test set. Our goal is to develop a model that generalizes well and wins the challenge.
- Apply best practices in designing and building machine learning models.
- Maintain simplicity and interpretability. We aim to use clear, effective techniques and models while ensuring a well-structured design architecture that maximizes performance.

2 Dataset and Splitting Strategy

2.1 Dataset Description

Our dataset consists of 1644 survey responses, evenly distributed across three food categories: Pizza, Shawarma, and Sushi (548 responses per category). Each respondent answered eight questions about each food item. These questions captured the perceived complexity, number of ingredients, typical serving setting, expected price, associated movies, drink pairings, personal associations, and hot sauce preferences. So the dataset is diverse in the sense it includes numerical, categorical, and text-based data.

2.2 Data Splitting Strategy

To ensure our model generalizes well to unseen data, we carefully designed our data splitting strategy to prevent data leakage and ensure reliable evaluation. Instead of random splitting, we grouped responses by respondent ID, ensuring that all responses from a single individual remained within the same split. This prevents the model from memorizing individual-specific patterns and better reflects real-world scenarios where it encounters entirely new individuals during testing. Additionally, we maintained class balance across all splits, ensuring equal representation of Pizza, Shawarma, and Sushi. This prevents bias toward any particular category during training and evaluation.

To ensure fair evaluation and reliable hyperparameter tuning, we employed a two-step data splitting strategy consisting of a fixed manual split followed by an automated robust tuning approach. This strategy ensured that all models were trained on the same set of respondents, hyperparameters were tuned on the same validation set, and generalization was tested on the same test set.

Step 1: Manual 70/15/15 ID-Based Split

Initially, we manually divided the dataset into:

- 70% Training Set: Used for training models.
- 15% Validation Set: Used for manual hyperparameter tuning and exploration.
- 15% Test Set: Completely held out during training and validation, used for the final generalization check.

This fixed split allowed us to manually explore feature importance and visualize the impact of different hyperparameters on performance. By analyzing trends on the validation set, we gained insights into parameter sensitivity, which informed our automated hyperparameter tuning strategy.

Step 2: Automated Group 5-Fold Cross-Validation

To obtain a more robust estimate of hyperparameter, we applied Group 5-Fold Cross-Validation (`GroupKFold`) on the combined training + validation sets.
We used `GroupKFold` instead of standard `KFold`, to ensure that all responses from a given individual remained within a single fold. This prevented data leakage and allowed the model to be evaluated on completely unseen individuals, aligning with our manual splitting approach. Each fold served as a validation set while training on the remaining folds, enabling us to average results for more stable hyperparameter selection. We automated this process using `GridSearchCV` with `GroupKFold`, systematically tuning model parameters. Finally, after finding the best configuration, we evaluated it on the held-out test set to confirm its real-world performance.

Final Model Training Before Submission

After selecting the best model and confirming its generalization ability, we retrained it on the entire dataset (train + validation + test sets) before submission. This ensured that the model learned from all available data, maximizing its ability to capture meaningful patterns.

3 Feature Engineering

To transform survey responses into a structured format suitable for machine learning, we developed the `FeatureBuilder` class. This class streamlines the data processing pipeline by applying a consistent set of transformations to any dataset, ensuring reproducibility and fairness across training, validation, and test splits, which is essential for maintaining a fair evaluation.

FeatureBuilder Pipeline

Our feature engineering process consists of four key steps. Standardization ensures consistency by converting all text data to lowercase. Feature extraction involves processing each survey question (Q1–Q8) to generate numerical or categorical representations. Missing value handling is applied appropriately based on the nature of each question. Lastly, feature interactions are created by combining existing features to capture meaningful patterns.

Since some transformations rely on statistical properties (e.g., mean, median, and standard deviation), all computations are performed exclusively on the training set to prevent data leakage. These precomputed values are then applied to validation and test sets to maintain consistency.

Overview of Feature Engineering

Each survey question (Q1–Q8) was processed based on the type of information it captured. In the following sections, we describe:

- The techniques applied to convert raw responses into feature values.
- How missing data was handled.
- Alternative techniques we considered.
- Data distribution analysis to identify patterns that could aid model learning.

3.1 Perceived Complexity Q1

Algorithm

Q1 responses are integer values from 1 to 5, representing perceived food complexity. Since the numbers are already meaningful, we used them directly as features without any transformations.

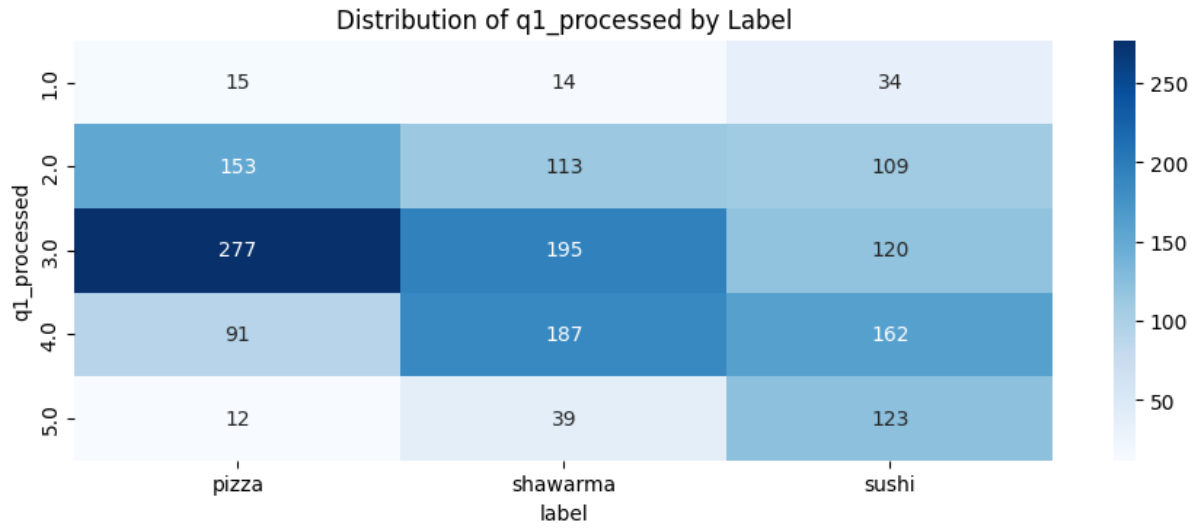
Missing Data

Missing values were replaced with the median complexity score from the training dataset.

Other Approaches

We didn't try any other transformations since using the raw numbers was the most intuitive and effective choice.

Data Distribution



Pizza is generally perceived as less complex (most responses are 2 or 3), shawarma is seen as moderately complex (mainly 3 and 4), while sushi has a wider spread (ranging from 2 to 5), though it is generally considered more complex. These patterns should help the model; for example, lower complexity scores should indicate a higher likelihood of pizza, while higher scores lean towards shawarma or sushi.

3.2 Ingredient Count Q2

Algorithm

A numerical column is extracted by identifying ingredient counts from the response:

- Search for explicit numerical values and take their average.
- If no number is found, look for spelled-out numbers (e.g., twenty-six $\rightarrow$ 26) and their their average.
- If still no number is found, estimate count based on the number of listed ingredients (separated by commas or new lines).
- If still missing, the value is considered missing data.

Additionally, three numerical probability features are derived using a Naive Bayes model (details provided in section 3.9 Naive Bayes Approach).

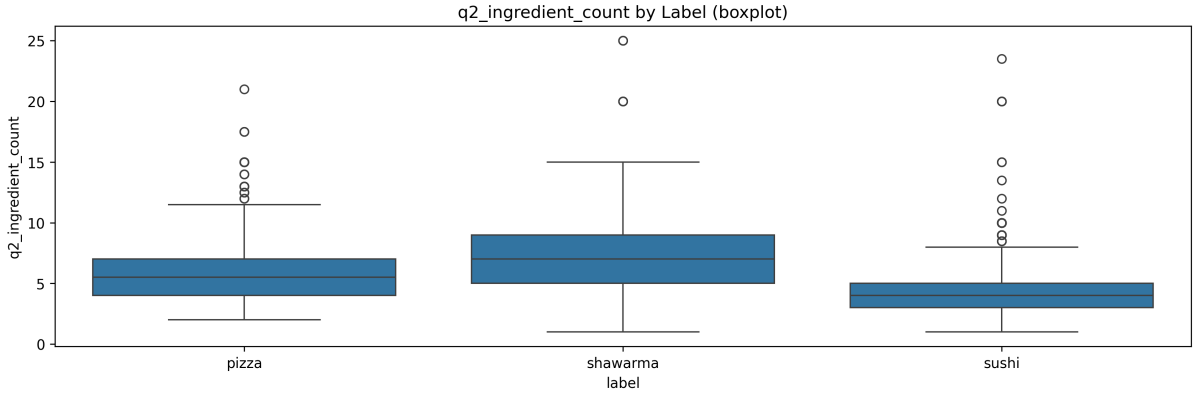
Missing Data

If no number is found, the value is replaced with the median ingredient count from the training set. For the probability-based features, missing values are set to $\frac{1}{3}$ for each class.

Other Approaches

Alternative techniques are discussed later in section 3.9 Naive Bayes Approach, as they are similar to those used for Q5 and Q6.

Data Distribution



mean	median	min	max	std
6.135605665623914	5.5	2.0	21.0	2.634679754612371
7.227980535279806	7.0	1.0	25.0	3.0412693490932
4.767822384428224	4.0	1.0	23.5	2.4038581713031584

The distribution of ingredient counts shows that pizza and sushi tend to have similar ingredient counts, typically around 5, while shawarma has a broader range, often around 7. This is further reflected in summary statistics, where shawarma exhibits higher values across the mean, median, minimum, and maximum. Overall, shawarma tends to have a higher ingredient count, followed by pizza and sushi, albeit by small margins.

3.3 Expected Serving Setting Q3

Algorithm

The raw data consisted of strings listing up to six possible options, separated by commas. We converted these into one-hot encoded vectors, where each of the six binary columns represents whether a specific option was selected in the response:
Options: Weekday Lunch, Weekday Dinner, Weekend Lunch, Weekend Dinner, At a Party, Late Night Snack

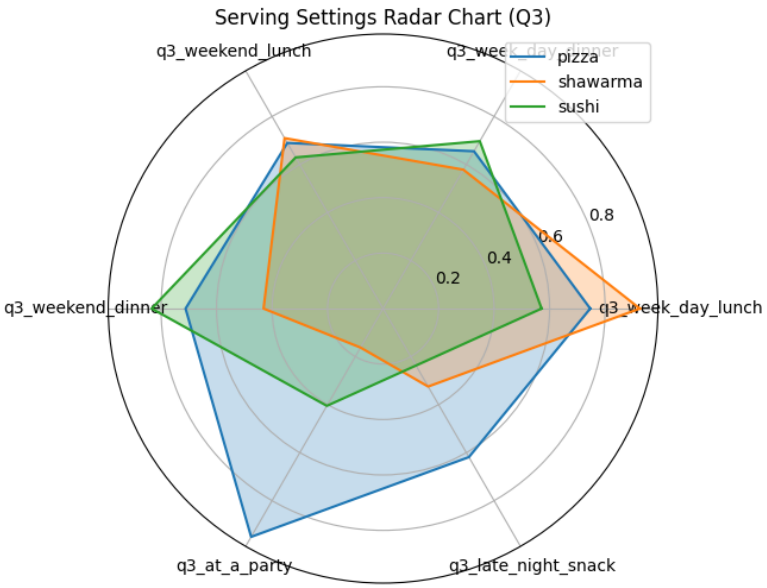
Missing Data

If a response was missing, we assumed that no option was selected, and the one-hot vector was set to all zeros.

Other Approaches

We did not explore alternative methods, as one-hot encoding was the most straightforward and effective way to represent multiple selections.

Data Distribution



The radar chart reveals distinct serving setting preferences for each food item. Pizza is strongly associated with parties and late-night snacks, while shawarma is most linked to weekday lunches. Sushi is predominantly chosen for weekend dinners, though pizza also has a notable presence in this setting. Weekday dinners and weekend lunches show overlap across all three categories, potentially making them less useful for distinguishing between food items. These patterns suggest that serving setting is a meaningful feature for classification, as certain options correspond to specific food labels.

3.4 Cost Q4

Algorithm

The raw data consisted of text responses indicating the perceived cost of the food item. We extracted numerical values using a stepwise approach similar to Q2:

- Search for explicit numerical values and take their average.
- If no number is found, look for spelled-out numbers (e.g., twenty-six $\rightarrow$ 26) and their their average.
- If still missing, the value is considered missing data.

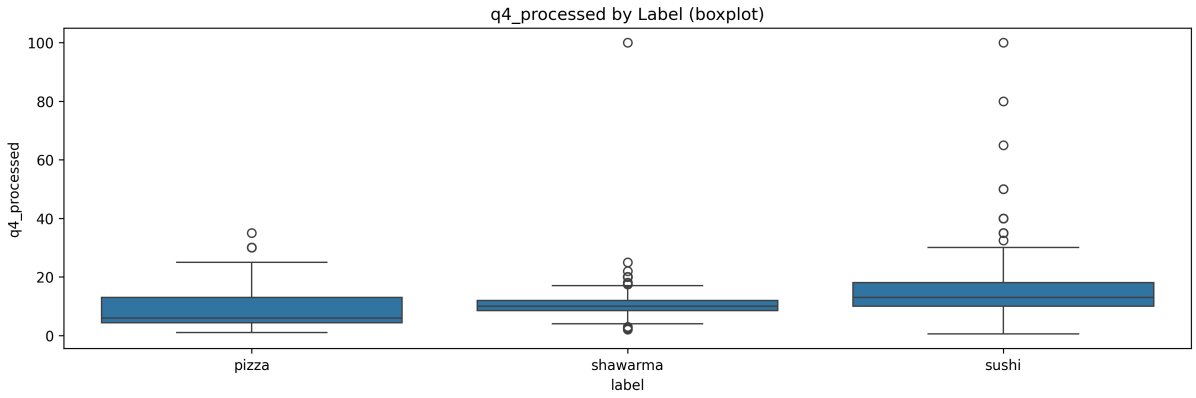
Missing Data

Missing values were replaced with the median perceived cost from the training set.

Other Approaches

We did not explore alternative methods, as extracting a numerical cost was the most intuitive and effective approach.

Data distribution



mean	median	min	max	std
8.838123479318735	6.0	1.0	35.0	6.1558986929460255
10.860742092457421	10.0	2.0	100.0	4.976849588537204
14.44404197080292	13.0	0.5	100.0	8.873155638831243

The box plot and summary statistics highlight differences in expected price (Q4) across the three food labels. Pizza has the lowest average price (≈ 8.84) and a relatively tight distribution. Shawarma has a slightly higher average price (≈ 10.86) with an even more concentrated distribution, though it includes an extreme outlier at 100. Sushi has the highest average price (≈ 14.44) and median (13), along with the largest standard deviation and the most widely spread outliers. These differences in expected price could serve as an informative feature for classification, particularly in distinguishing sushi from the other two categories.

3.5 Associated Movie Q5

Algorithm

The raw data consisted of free-text responses. We transformed these into three numerical probability features $P(\text{class}|\text{response})$, computed using a Naive Bayes-based text model (details provided later).

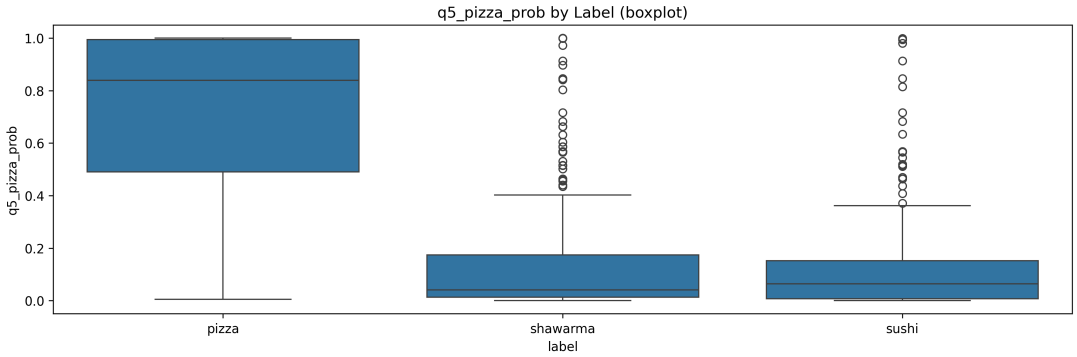
Missing Values

If the response was missing, we assigned equal probabilities of $\frac{1}{3}$ for each class.

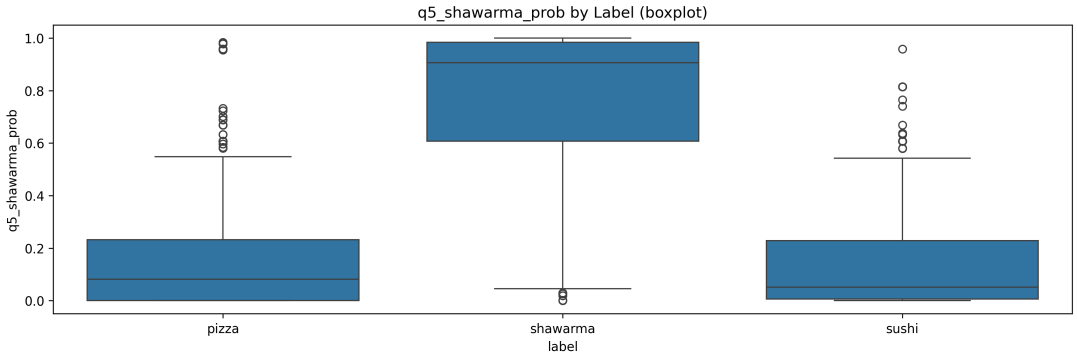
Other Approaches

Alternative methods, including different text processing techniques, are discussed in the Naive Bayes section, as they are closely related to our approach for Q2 and Q6.

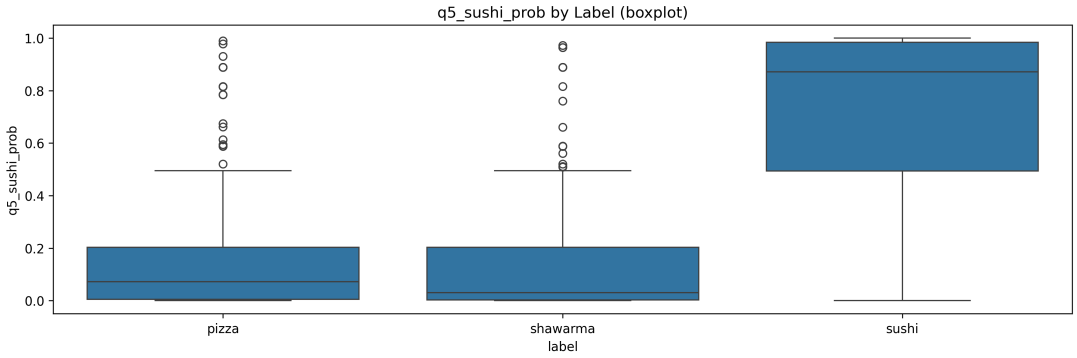
Data Distribution



mean	median	min	max	std
0.7080911281709232	0.8387502612440461	0.004548620475721289	0.9999999998478444	0.29937370140649494
0.12478337413885984	0.040545730471236524	6.158164786858829e-08	0.9997906166077919	0.17215153360269628
0.11094110864628991	0.06452931285789831	1.0530402222401134e-07	0.9996201541149362	0.1552183469264345



mean	median	min	max	std
0.1711251916360937	0.08151264293398822	1.3863143524301033e-10	0.9841594826842798	0.21141760683006033
0.7613021331813961	0.9067714876494648	4.371968383740858e-06	0.9999999383614143	0.27027817387955116
0.15142396133859895	0.05182628678264904	2.347686024146312e-08	0.9582367645743708	0.19527703366305393



mean	median	min	max	std
0.12078368019298329	0.07234867734255315	1.3523505169440009e-11	0.9898861277670123	0.16351694328297006
0.11391449267974421	0.030258536332596904	5.6927823265480446e-11	0.9729583879197118	0.16185146552886132
0.7376349300151114	0.8715996860807593	0.00037547391667754656	0.9999998703034279	0.2879082183158207

The box plots for Q5 indicate that the Naive Bayes-based probability features derived from movie names are highly effective in distinguishing between the three food categories. For instance, `q5_pizza_prob_boxplot` has a significantly higher mean for pizza (≈ 0.83) compared to shawarma (≈ 0.04) and sushi (≈ 0.06), with little to no overlap between distributions. A similar pattern is observed for `q5_shawarma_prob_boxplot` and `q5_sushi_prob_boxplot`, where each probability feature is distinctly aligned with its respective food category. This suggests that movies serve as strong predictors, likely because certain films are more uniquely associated with specific foods, reinforcing their relevance as a classification feature.

3.6 Associated Drink Q6

Algorithm

The raw data consisted of free-text responses. We transformed these into three numerical probability features $P(\text{class}|\text{response})$, computed using a Naive Bayes-based text model (details provided later).

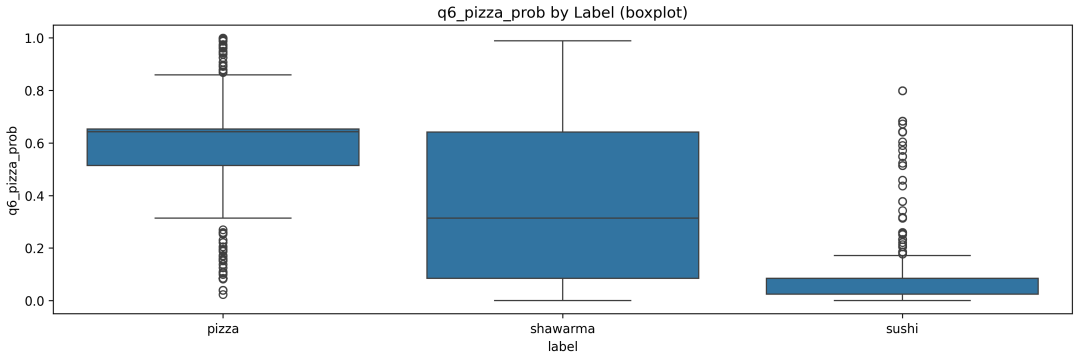
Missing Values

If the response was missing, we assigned equal probabilities of $\frac{1}{3}$ for each class.

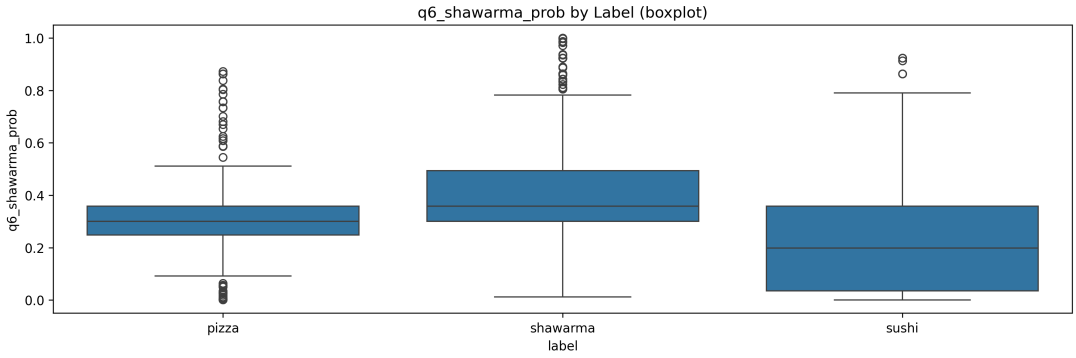
Other Approaches

Alternative methods, including different text processing techniques, are discussed in the Naive Bayes section, as they are closely related to our approach for Q2 and Q5.

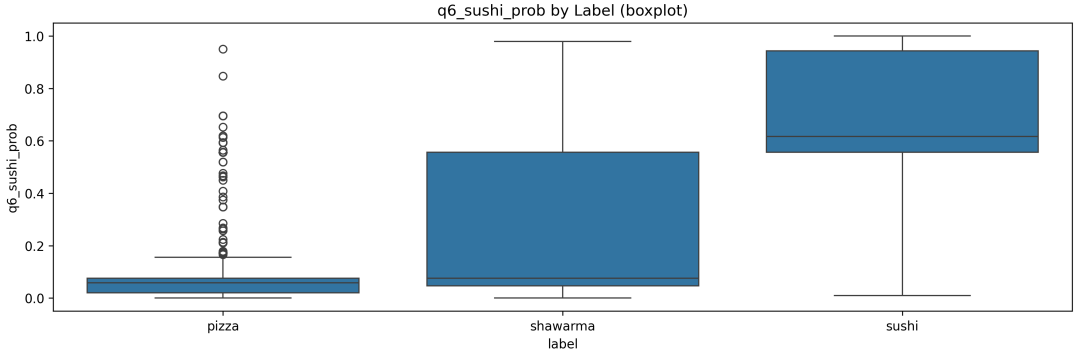
Data Distribution



mean	median	min	max	std
0.5792373347535023	0.6421703605884906	0.0238922483149313	0.9989504802426868	0.21349556651304838
0.3472262687468923	0.31441833996835755	2.0669618324510966e-06	0.9885391304220005	0.26518949466470315
0.11047031320438662	0.08486515542145075	5.269358750855672e-08	0.7993798823658379	0.15274539512502847



mean	median	min	max	std
0.31362873581707124	0.29993244518654233	0.0009024559269300866	0.8727287930043991	0.1356711900020608
0.4212114839927311	0.3584045253025227	0.011458375784464132	0.999997650179726	0.1911909835575506
0.2204599813780067	0.1987247169639918	3.3412046495048326e-06	0.9235366763270115	0.1778338074780872



mean	median	min	max	std
0.10713392942942658	0.05789719422496746	4.307326618235858e-08	0.9499463913103463	0.163292214858885537
0.23156224726037666	0.07515410307082251	1.6856916538136054e-07	0.9789973683729177	0.243077697873847
0.6690697054176067	0.6166580727903075	0.009115698078475988	0.9999966061017641	0.26372341172059144

In contrast to movies, the drink-based probability features exhibit greater overlap across food categories, making them less distinctive for classification. For instance, in `q6_pizza_prob_boxplot`, while pizza has the highest mean probability (≈ 0.58), shawarma is not far behind (≈ 0.35), creating overlap between the two categories. Similarly, in `q6_shawarma_prob_boxplot`, all three categories have moderate probability values (means of ≈ 0.31 for pizza, ≈ 0.42 for shawarma, and ≈ 0.22 for sushi), further indicating that drink preferences are less exclusive to a single food category. The clearest distinction occurs in `q6_sushi_prob_boxplot`, where sushi has a much higher mean probability (≈ 0.67) compared to pizza (≈ 0.11) and shawarma (≈ 0.23), suggesting that drinks may still help identify sushi to some extent.

These results indicate that drinks, while somewhat informative, do not provide as strong of a signal for classification as movies. This is likely because many drinks, such as soda, tea, and wine, can commonly be paired with multiple foods, whereas movies tend to have more distinct cultural or thematic associations with specific cuisines.

3.7 Personal Association Q7

Algorithm

The raw data consisted of strings listing up to five possible options, separated by commas. We converted these into one-hot encoded vectors, where each of the six binary columns represents whether a specific option was selected in the response:
Options: Parents, Siblings, Friends, Teachers, Strangers

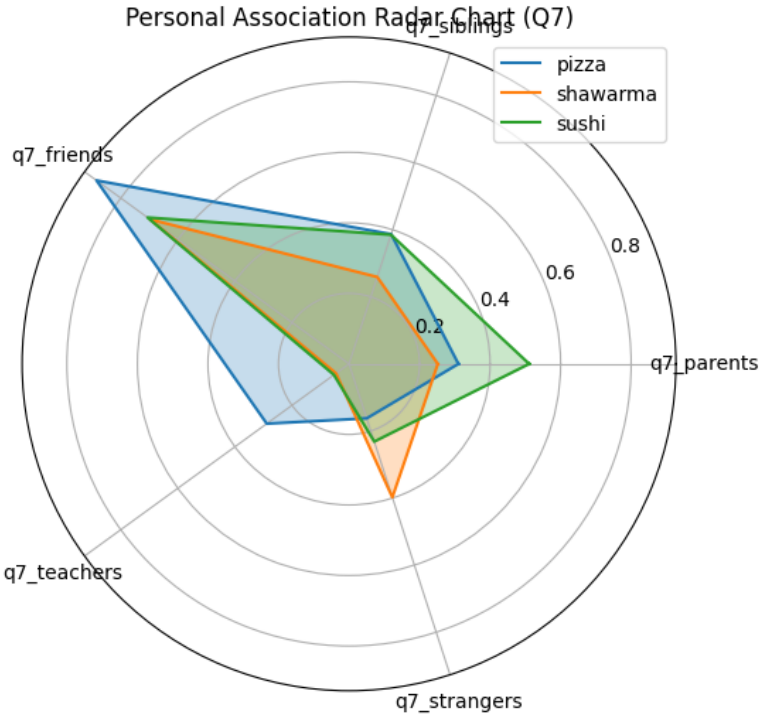
Missing Data

If a response was missing, we assumed that no option was selected, and the one-hot vector was set to all zeros.

Other Approaches

We did not explore alternative methods, as one-hot encoding was the most straightforward and effective way to represent multiple selections.

Data Distribution



According to the above figure, every food item has a large association to friends. Shawarma has a slightly stronger association to strangers, and sushi has a slightly stronger associ-

ation to parents. This is somewhat problematic, as we don't have stronger patterns in association to help our model distinguish between food items.

3.8 Hot Sauce Preference Q8

Algorithm

The raw data consisted of categorical responses representing different levels of hot sauce preference. We mapped these categories onto an ordinal numerical scale, preserving the natural ordering:

```
option_mapping = {
    "none": 0,
    "a little (mild)": 1,
    "a moderate amount (medium)": 2,
    "a lot (hot)": 3,
    "i will have some of this food item with my hot sauce": 4
}
```

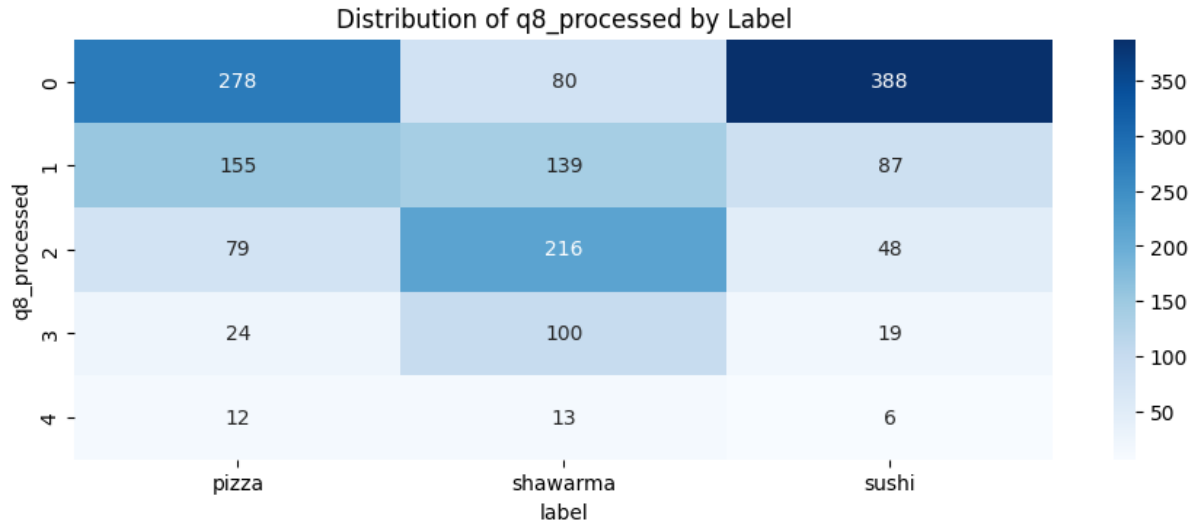
Missing Data

Missing values were replaced with 0 (indicating no hot sauce).

Other Approaches

We considered one-hot encoding, similar to Q3 and Q7, where each option would be represented as a separate binary column. However, since the responses had a clear ordinal structure, a single numerical feature was more appropriate. This also reduced the number of features.

Data Distribution



From the above figure, we observe that sushi has a strong association with a score of 0, indicating that most people do not prefer hot sauce with it. Pizza shows a broader distribution, with responses spread across different levels, though it still has a noticeable presence at lower scores. Shawarma, on the other hand, has a higher distribution of scores, suggesting that people tend to prefer more hot sauce with it.
In terms of model learning, this pattern suggests that lower scores are likely indicative of sushi or pizza, while higher scores are more strongly associated with shawarma. This distinction could help the model in recognizing shawarma.

Summary of Feature Transformations

Question	Original Type	Final Features	Handling Missing Values
Q1	Numerical	1 numerical	Replace with median
Q2	Text	4 numerical	Replace number with median, Set probabilities to 1/3 each
Q3	Options	6 categorical (one-hot)	Set all to 0
Q4	Text	1 numerical	Replace with median
Q5	Text	3 numerical	Set probabilities to 1/3 each
Q6	Text	3 numerical	Set probabilities to 1/3 each
Q7	Options	5 categorical (one-hot)	Set all to 0
Q8	Options	1 numerical	Replace with 0

3.9 Naive Bayes Approach

For text-based responses such as Q2, Q5, and Q6, we employed a Naive Bayes-based transformation to convert free-text responses into three numerical features:

$$P(\text{class}|\text{response})$$

This transformation replaces each text response with a probability distribution over the three target classes. By doing so, we integrate textual features into our model without relying on manual categorization or bag-of-words approaches, which often introduce sparsity.

Training Phase

For each text-based column, we precompute and store the following based on the training dataset:

- Vocabulary (V): A set of unique words appearing in the training responses.
- Priors $P(\text{class})$: Since the dataset is balanced, all priors are $P(\text{class}) = \frac{1}{3}$.
- Likelihoods $P(\text{word}|\text{class})$: Estimated using Laplace smoothing to prevent zero probabilities:

$$P(\text{word}|\text{class}) = \frac{\#\text{ of documents in class containing word} + 1}{\#\text{ of documents in class} + |V|}$$

  where $|V|$ is the vocabulary size.

  (We also experimented with Maximum A Posteriori (MAP) estimation using hyperparameters α, β, but the results were similar to Laplace smoothing while introducing an extra parameter to tune. Thus, we opted for the simpler approach.)

Each text column maintains its own vocabulary, priors, and likelihoods, ensuring the model correctly captures the distribution of words relevant to that specific feature.
In summary, the training phase produces the following for each text-based column:

- V: A list of words from the training data.
- A dictionary of priors: $\{\text{class} : P(\text{class})\}$.
- A dictionary of likelihoods: $\{\text{class} : \{\text{word} : P(\text{word}|\text{class})\}\}$.

Inference Phase

Given a new response, we compute:

$$P(\text{class}|\text{response}) = \frac{P(\text{response}|\text{class})P(\text{class})}{P(\text{response})} = \frac{P(\text{response}|\text{class})P(\text{class})}{\sum_{\text{class}} P(\text{response}|\text{class})P(\text{class})}$$

where

$$P(\text{response}|\text{class}) = \prod_{\text{word} \in \text{response}} P(\text{word}|\text{class})$$

where

- Unseen words (not in V) are ignored.
- For words present in the response, we use $P(\text{word}|\text{class})$.
- For words from the vocabulary that are absent, we use $1 - P(\text{word}|\text{class})$ to model their absence.

Alternative Approaches Considered

We explored several alternative techniques for handling text-based columns:

Manual Categorization: We initially created predefined categories (e.g., based on the top N most common movies or drinks or based on TF-IDF score). This approach led to rigid feature definitions that were heavily dependent on the training data, limiting generalization. Given these drawbacks, we wanted to move away from manual categorization as soon as possible, regardless of the results we were getting.

Bag-of-Words (BoW): This method produced results very similar to our Naive Bayes approach but introduced many additional features, resulting in a high-dimensional and sparse feature matrix. The increased dimensionality made the model harder to interpret. Although Naive Bayes is more complex, we chose it over BoW to reduce feature count and make it easier to generate and analyze graphs for interpretation.

Separate Naive Bayes Model for Text Features: Another approach we thought about was training a Naive Bayes classifier on text features separately, while using a different model (e.g., a random forest) for the other features. The outputs could then be combined using voting. However, this approach is not only more complicated but also limits interactions between textual and non-textual features, as the models are trained independently. Instead of separating the text-based features into a distinct classifier, we fed the transformed probabilities directly into the main model. This allowed our model to leverage textual features alongside numerical and categorical inputs, increasing predictive power.

3.10 Interaction Features

Machine learning models often assume that features contribute independently to the target variable, but real-world relationships are rarely that simple. By incorporating interaction features, we allow the model to capture more complex dependencies between variables.

We suspected that factors such as ingredient complexity, cost, and hot sauce usage might have multiplicative or ratio-based relationships. For example,

- *Does a more complex dish tend to have more ingredients?*
- *Does the cost of a dish increase with complexity?*
- *Does hot sauce usage scale with ingredient count or price?*

Selecting Interaction Features

To explore these relationships, we generated all pairwise combinations of four features: Q1 (Complexity), Q2 (Ingredient Count), Q4 (Cost), and Q8 (Hot Sauce Usage).

For each pair of features, we applied both multiplication and division operations to create interaction terms. We then evaluated the impact of these interaction terms on the ExtraTreesClassifier by appending them one at a time to our base feature set.

To identify the most valuable interactions, we considered different values of k—the number of interaction terms added—ranging from 1 to the maximum number of possible interaction terms (i.e, using all of them). In total, the ExtraTreesClassifier was trained on over 32,000 different interaction combinations.

Through this process, we identified that adding the following 8 interaction features ($k = 8$) provided the highest accuracy,

Table 3.1: Interpretation of Key Feature Interactions

Feature	**Interpretation**
Q1 × Q2	Does higher complexity imply more ingredients?
Q1 × Q4	Does higher complexity imply higher cost?
Q4 ÷ Q1	Cost per complexity
Q2 × Q4	Do dishes with more ingredients tend to be more expensive?
Q2 ÷ Q4	Ingredient count per dollar—how much does each ingredient contribute to cost?
Q4 ÷ Q2	Cost per ingredient—normalizes cost by ingredient count.
Q8 ÷ Q2	Hot sauce per ingredient—do dishes with more ingredients use more hot sauce?
Q8 ÷ Q4	Hot sauce per dollar—how much hot sauce is used relative to dish cost?

To further validate these extra features, we compared the ExtraTreesClassifier's performance before and after adding the 8 interaction features across 91 different train-validation-test splits. This approach helps ensure the robustness and generalizability of the model.

Table 3.2: Model Performance Comparison With/Without Feature Interactions

Metric	Baseline	With Interactions
Validation Accuracy		
Lowest	84.96%	85.77%
Average	90.01%	**90.66%**
Highest	94.31%	94.31%
Test Accuracy		
Lowest	83.53%	83.94%
Average	89.96%	**90.82%**
Highest	94.38%	94.78%
90%+ Splits		
Validation	46/91	**56/91**
Test	42/91	**60/91**

Model Configuration

Parameter	Value
bootstrap	True
criterion	gini
max_depth	None
max_features	sqrt
min_samples_leaf	1
min_samples_split	2
n_estimators	180
random_state	42

(For more details, refer to the files `ExtraTreesClassifier_NoInteractionFeat.txt` and `ExtraTreesClassifier_WithInteractionFeat.txt`.)

Adding these eight interaction features consistently improved performance across unseen data. Specifically, we observed higher average accuracy and more instances of accuracy exceeding 90%, indicating genuine performance gains rather than overfitting or random chance.

Overall, the interaction features allowed the model to capture more complex relationships between ingredients, cost, complexity, and hot sauce usage, leading to higher predictive accuracy. Furthermore, evaluating across multiple data splits reinforces the robustness of our results, showing that the improvements are not due to overfitting.

4 Model Exploration

Having established the features we will use and the transformations needed to extract them from raw data (via the `FeatureBuilder` class), we now turn our focus to evaluating the performance of various models.

As outlined in section 2.2 Data Splitting Strategy, we used a two-step strategy to split the data, which guided both our model exploration and evaluation. First, we performed a manual 70/15/15 ID-based split to gain initial insights and visualize how different hyperparameters affected validation accuracy. This allowed us to manually tune and experiment with different model parameters.

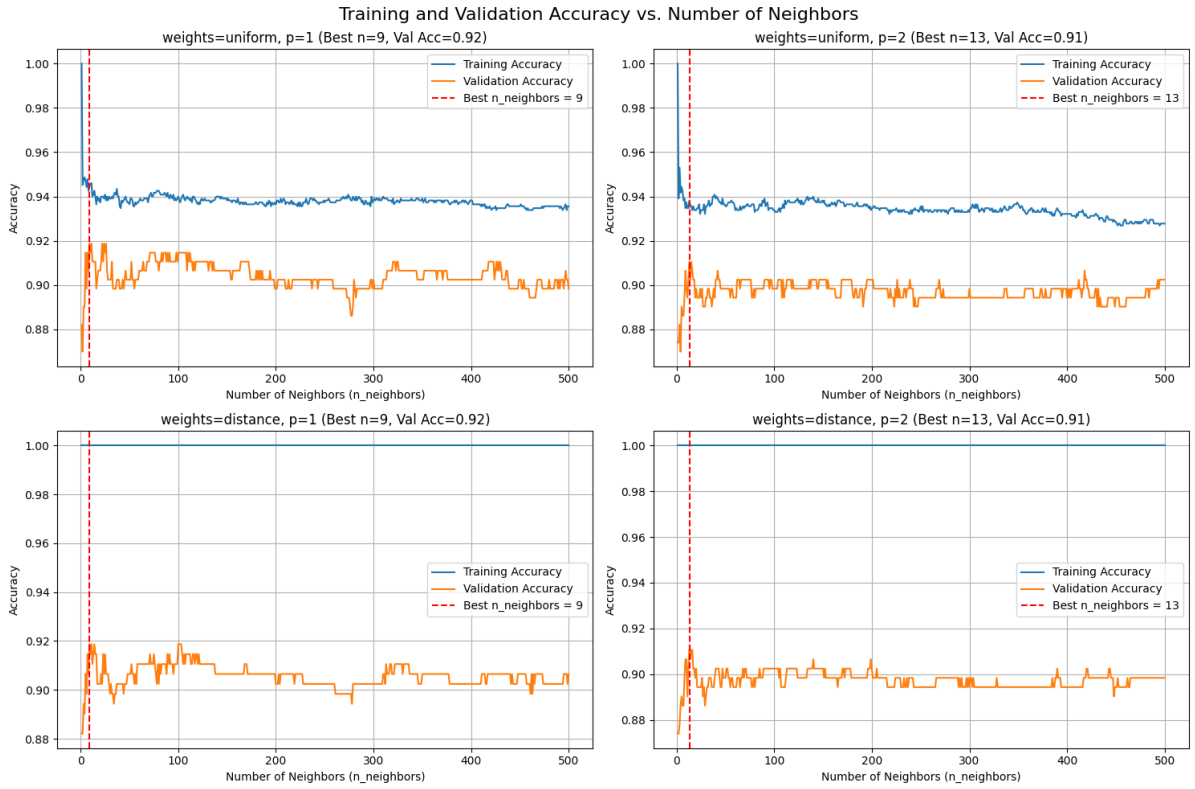
To ensure a more robust estimate of model performance and avoid overfitting, we then employed Group 5-Fold Cross-Validation in combination with GridSearch to automate hyperparameter tuning. This approach provided a more reliable estimate of generalization by averaging performance across multiple folds, with the additional benefit of preventing data leakage by ensuring all responses from a given individual remained within the same fold.

4.1 k-Nearest Neighbors (kNN)

The code accompanying the following analysis can be found in `explore_knn.ipynb`.

k-Nearest Neighbors (kNN) is a simple, non-parametric method that classifies a sample based on the majority class among its nearest neighbors. Since kNN is scale-sensitive, we normalized all features based on training set statistics to ensure equal weighting in distance calculations.

We investigated the impact of three key parameters on accuracy: the number of neighbors ($1 \leq k \leq 500$), distance metric (Manhattan vs. Euclidean), and weighting scheme (uniform vs. distance-based).



Our manual exploration indicates that Manhattan distance ($p = 1$) slightly outperforms Euclidean distance ($p = 2$) in validation accuracy. The weighting scheme showed distinct training behaviors–distance weighting maintained perfect training accuracy regardless of k due to the dominance of the nearest (identical) neighbor, while uniform weighting exhibited the expected decrease in training accuracy with larger k values. However, both schemes yielded comparable validation performance, with accuracy typically hovering around 90% for most values of k.

To obtain more reliable hyperparameter estimates, we conducted a comprehensive grid search using 5-fold GroupKFold cross-validation. The search space and the best-found configuration are presented below:

Table 4.1: Grid Search Parameters and kNN Configuration

```
param_grid = {
    "n_neighbors": list(range(1, 501)),
    "weights": ["uniform", "distance"],
    "p": [1, 2]
}
```

Parameter	Value
n_neighbors	29
p	1
weights	distance

This model achieved 93.62% cross-validation accuracy. On the held-out test set, it maintained, strong generalization with 91.16% accuracy.

Model Performance

Next, we analyze the classification report and confusion matrix generated from the test set.

Class	Precision	Recall	F1-score
pizza	0.90	0.98	0.94
shawarma	0.88	0.89	0.89
sushi	0.96	0.87	0.91
Accuracy			91.16%

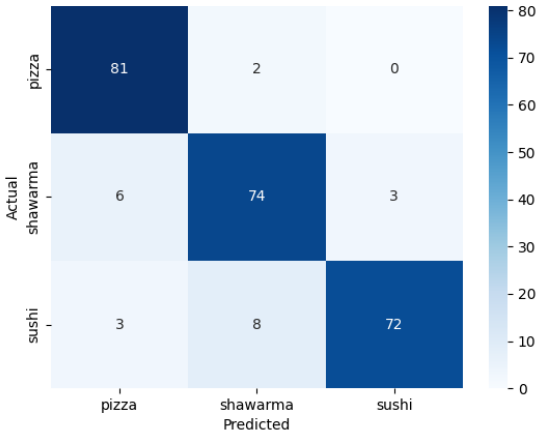


Figure 4.1: Test Set Classification Report (left) & Confusion Matrix (right)

The kNN classifier demonstrates strong performance across all three food categories. Pizza achieves the highest recall (98%), meaning that nearly all actual pizza samples are correctly identified. Sushi, on the other hand, has the highest precision (96%), indicating that when the model predicts sushi, it does so with high confidence. Shawarma exhibits a balanced performance, with both precision and recall around 90%, ensuring consistent classification accuracy. This pattern suggests pizza may have the most distinctive feature signature, while sushi, though identifiable when its characteristic pattern appears, shares more features with other categories in some instances. The F1-scores for all classes are

around 90%, highlighting the model's ability to maintain a strong trade-off between precision and recall across categories.

While kNN demonstrated strong and consistent performance, several practical limitations warrant consideration. As a lazy learning algorithm, kNN requires storing the entire training set, making inference computationally expensive—particularly for large datasets. Additionally, with our dataset's 20+ features and relatively small sample size, we must consider the potential impact of the curse of dimensionality, which can diminish the meaningfulness of distance metrics in high-dimensional spaces. Despite these considerations, kNN achieved exceptional classification performance (90%+ accuracy), demonstrating that simple distance relationships effectively capture the distinguishing patterns between our food categories

4.2 Decision Trees

The code for the following analysis can be found in `explore_decision_tree.ipynb`.

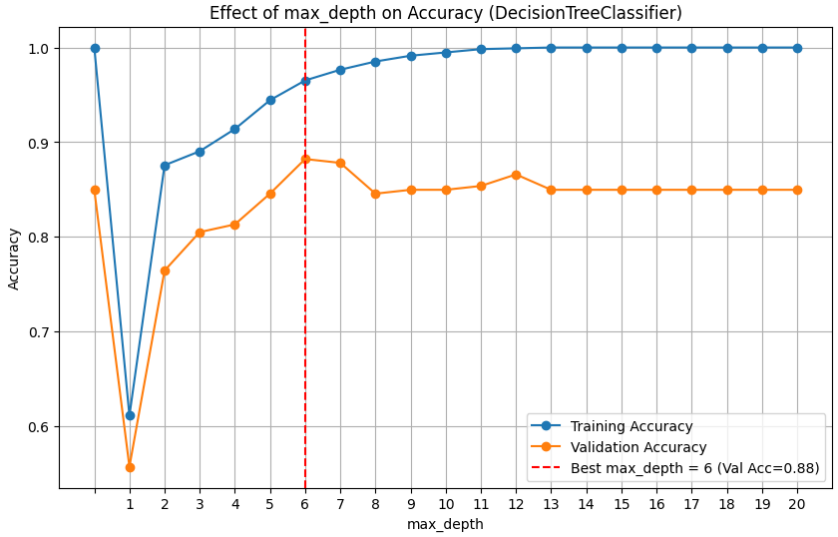
Decision trees are known as "universal function approximators", which are systems that are able to model any function to an arbitrary degree of accuracy. We believed that this property of decision trees would allow them to be able to interpret complex patterns, and make predictions to a high degree of accuracy. It was for this reason that we opted to include all features as part of training sets, as we wanted to provide the decision trees as much information as possible to extract and generalize patterns for predictions.

Hyperparameter Exploration

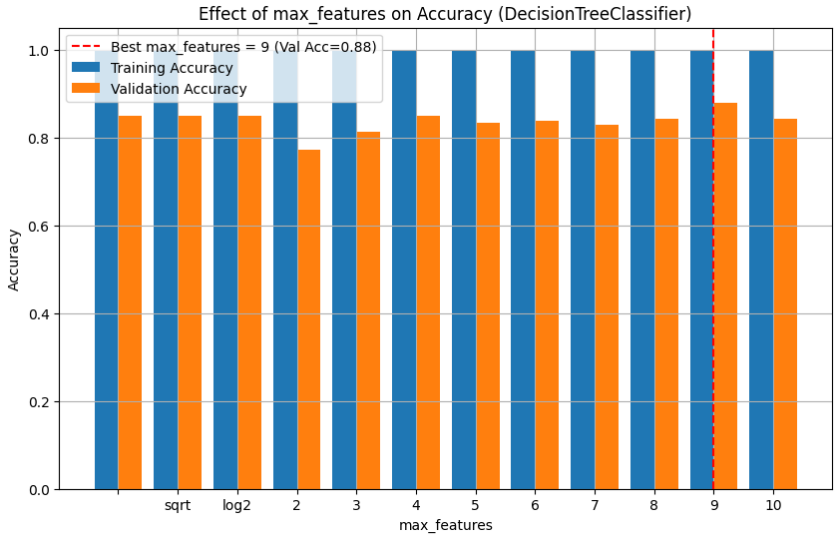
We first wanted to analyze the impacts of the following six hyperparameters:

```
"max_depth": max height of any tree
"max_features": number of features to consider when looking for the best split
"max_leaf_nodes": max number of leaf nodes of any tree
"min_samples_split": min number of samples required to split a non-leaf node
"min_samples_leaf": min number of samples required to be a leaf node
"criterion": function to measure the quality of a split
```

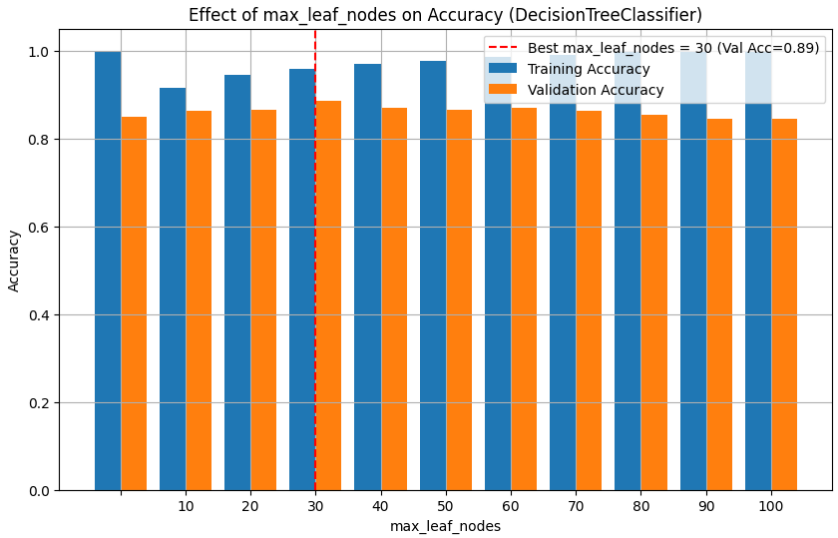
We tested these hyperparameters by running a series of tests where we set of all but one hyperparameter, x, to their default values. Allowing x to be variable, we trained a decision tree classifier using this set of hyperparameters, and then plotted the training and validation scores as the values of x changed. The goal is to find the best performing values of these hyperparameters to feed into the GridSearch we conduct later. Here are the results:



With `max_depth`, we can see that the validation score reaches its absolute max of 0.88 at a value of 6, followed by a smaller peak of 0.86 at 12, before converging to a score of 0.85, beginning at a `max_depth` of 13. Note that a `max_depth` of `None` (at the far left of the graph), also attains a peak of about 0.85. We can conclude that the `max_depth` values of `None`, 6, and 12, provide peaks in model performance, while values beginning from 13 provide stable model performance. We will consider these values in our eventual GridSearch.

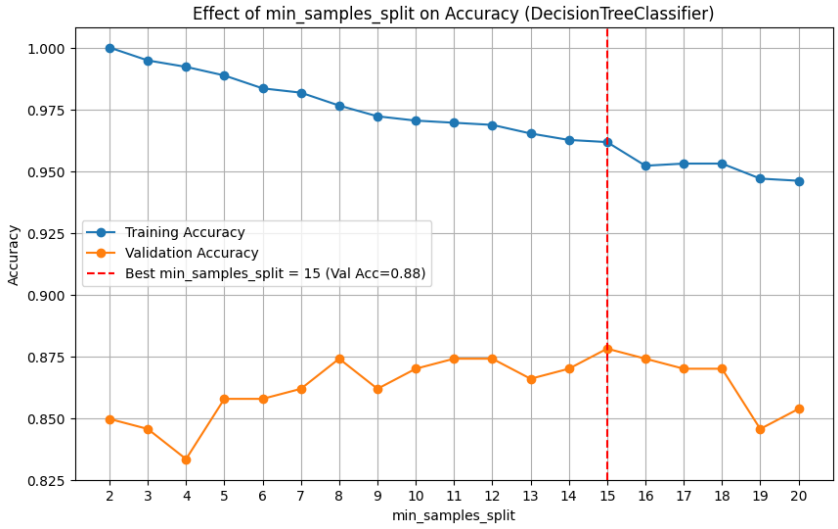


With `max_features`, we can see that the validation score reaches an absolute max of 0.88 at a value of 9. Before that, a smaller peak of approximately 0.85 occurs at 4, with similar validation scores occurring at values of `None`, `sqrt`, and `log2`. We can conclude that `max_features` values of `None`, `sqrt`, `log2`, 4, and 9 will theoretically improve model performance, and we will consider these values in our GridSearch.

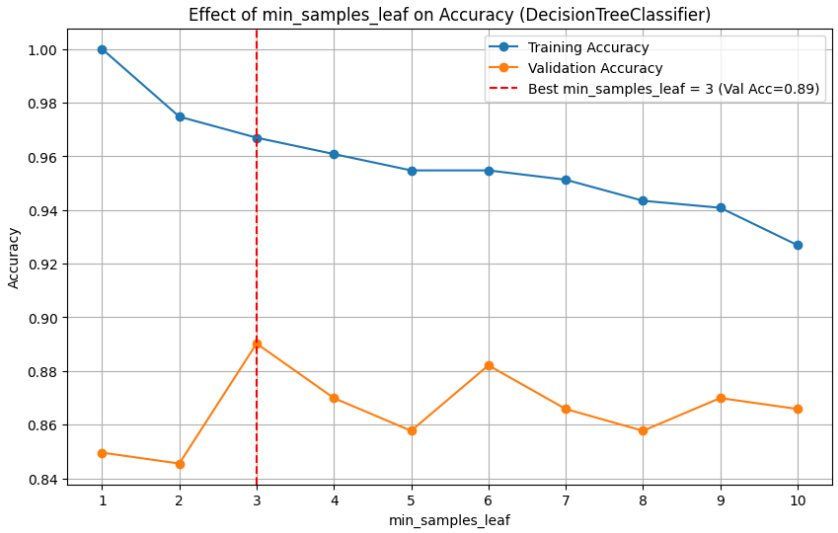


With `max_leaf_nodes`, we can see that the validation score reaches an absolute max

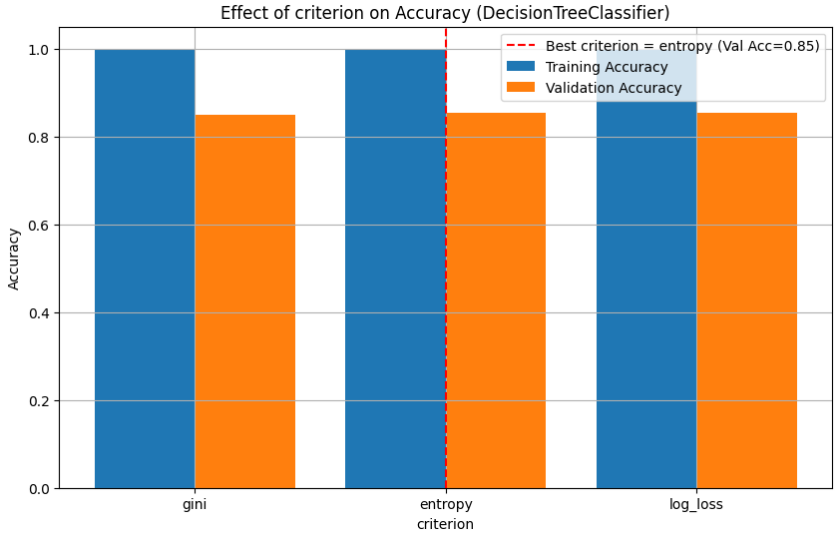
of 0.89 at a value of 30. Other values of `max_leaf_nodes` offer similar, yet lower validation scores. However, we can notice a subtle, small peak of approximately 0.87 at a `max_leaf_nodes` value of 60. We can conclude that `max_leaf_nodes` values of 30 and 60 will theoretically improve model performance, and we will consider these values in our GridSearch. Additionally, we will consider the value of `None`, to see if uncapped `max_leaf_nodes` will be beneficial in GridSearch.



With `min_samples_split`, we can see that the validation score achieves an absolute max of 0.88 at a value of 15, with smaller peaks of 0.875 occurring at values of 8, 11 and 12. We can also notice that the training accuracy steadily decreases as `min_samples_split` increases. It appears concerning, but note that the difference between training and validation accuracy also decreases as `min_samples_split` increases, which implies that the model is overfitting less and less, which is a good sign. We can conclude that `min_samples_split` values of 8, 11, and 15 maximize model performance, and we will consider these values for GridSearch. Note that we are not considering a `min_samples_split` value of 12 because its performance is too similar to values of 8 and 11, it would be redundant to add it.



With `min_samples_leaf`, we can again see that the training accuracy steadily decreases as `min_samples_leaf` increases. As previously stated, this indicates that the model is overfitting less and less, which is a good sign. With regards to validation score, we can see that an absolute max of 0.89 occurs at a `min_samples_leaf` value of 3, followed by smaller peaks of 0.88 and 0.87 at values of 6 and 3, respectively. As a result, we will include `min_samples_leaf` values of 3, 6, and 9 as parameters for GridSearch.



With `criterion`, we can see that all possible values result in extremely similar, almost equal training and validation accuracies, with `entropy` performing slightly better with a validation accuracy of 0.85. Because of the extreme similarity in training and validation scores, we will consider all values of `criterion` in our GridSearch.

Our final parameters for our GridSearch and best parameter values were:

Table 4.2: Grid Search Parameters & Best Decision Tree Configuration

```
param_grid = {
    "max_depth": [None, 6, 12, 13],
    "max_features": [None, sqrt, log2, 4, 9],
    "max_leaf_nodes": [None, 30, 60],
    "min_samples_split": [8, 11, 15],
    "min_samples_leaf": [3, 6, 9],
    "criterion": ["log_loss", "entropy", "gini"],
}
```

Parameter	Value
criterion	gini
max_depth	None
max_featyres	9
max_leaf_nodes	30
min_samples_leaf	3
min_samples_split	11

The performance of the model with the best parameters is outlined in the next section.

Model Performance

The best parameters achieved a validation accuracy of 91% and a test accuracy of 86%. The 5% gap suggests slight overfitting. Below are the test classification report and confusion matrix.

Class	Precision	Recall	F1-score
pizza	0.88	0.92	0.90
shawarma	0.83	0.83	0.83
sushi	0.86	0.83	0.85

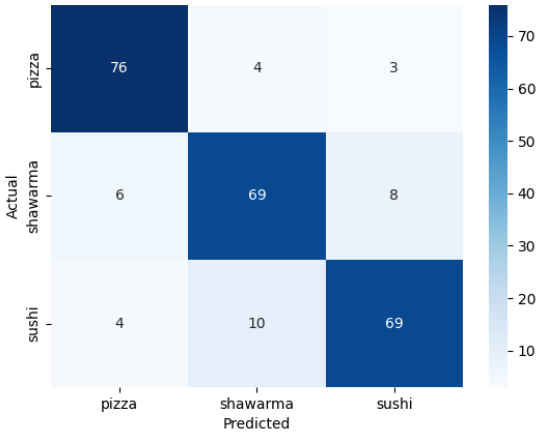


Figure 4.2: Test Set Classification Report (left) & Confusion Matrix (right)

The classification report shows that the model performs best on pizza, achieving an F1-score of 0.90, while its performance on sushi and shawarma is weaker. The confusion matrix confirms this, indicating that the model tends to misclassify shawarma and sushi. As expected, decision trees excel at capturing complex patterns across features, making them more effective than linear models. However, this flexibility also increases the risk of overfitting, and our results suggest some signs of it. To enhance generalization, careful tuning of hyperparameters such as pruning, max depth, or ensembling (e.g., random forests) is necessary to prevent the model from memorizing training data instead of learning broader patterns.

4.3 Logistic Regression

The code accompanying the following analysis can be found in `explore_logistic_reg.ipynb`.

Next, we focus on the Logistic Regression classifier. We began by investigating the influence of each hyperparameter on model performance. This involved a series of experiments in which we varied one hyperparameter at a time, keeping all other hyperparameters fixed.

Hyperparameter Exploration

Initially, we investigated the effect of the `max_iter` hyperparameter. For this test, we kept the following hyperparameter values constant:

```
"C": 1, # represents a moderate level of regularization
"l1_ratio": 0.5, # balances L1 and L2 penalty strength
"multi_class": "auto", # let scikit-learn choose the best strategy for multiclass
"penalty": "elasticnet", # combines benefits of both L1 and L2 penalties
"solver": "saga" # solver to handle our penalty value
```

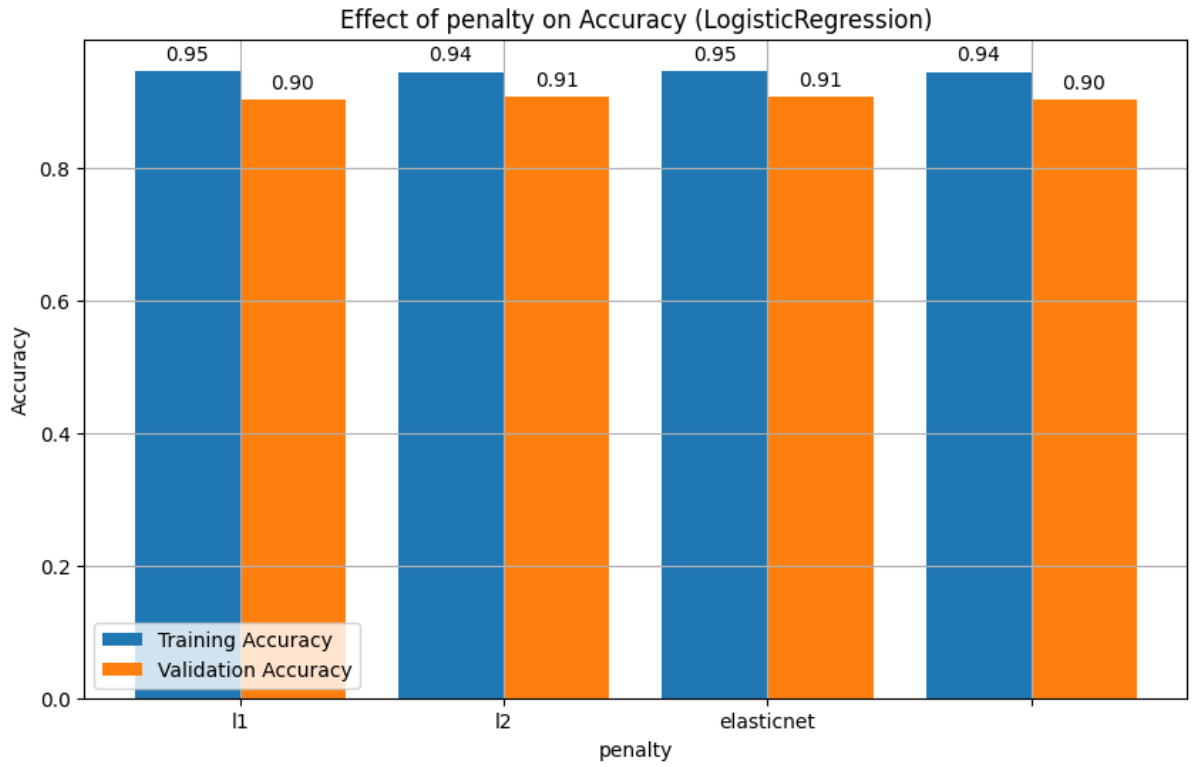
The results of this investigation are shown in the following table:

We can see the impact of the `max_iter` hyperparameter on the model keeping other hyperparameters constant. Training and validation accuracies initially exhibit a rapid increase, plateauing after 150 iterations, pointing to diminishing returns. The slight gap

between training and validation accuracy may suggest some overfitting, but it does not worsen significantly with a higher number of iterations. From analyzing the graph, we can see that we quickly reach the optimal range of iterations between 100-200.
We also observed the effect of individual `penalty` types being applied on our model. For his test we tested for L1, L2, Elastic Net (combination of L1 and L2), and no penalty applied. We kept the following hyperparameters constant:

```
"C": 1, "l1_ratio": 0.5, "max_iter": 800, "multi_class": "auto",
"solver": "saga"
```

The following graph demonstrates our findings:

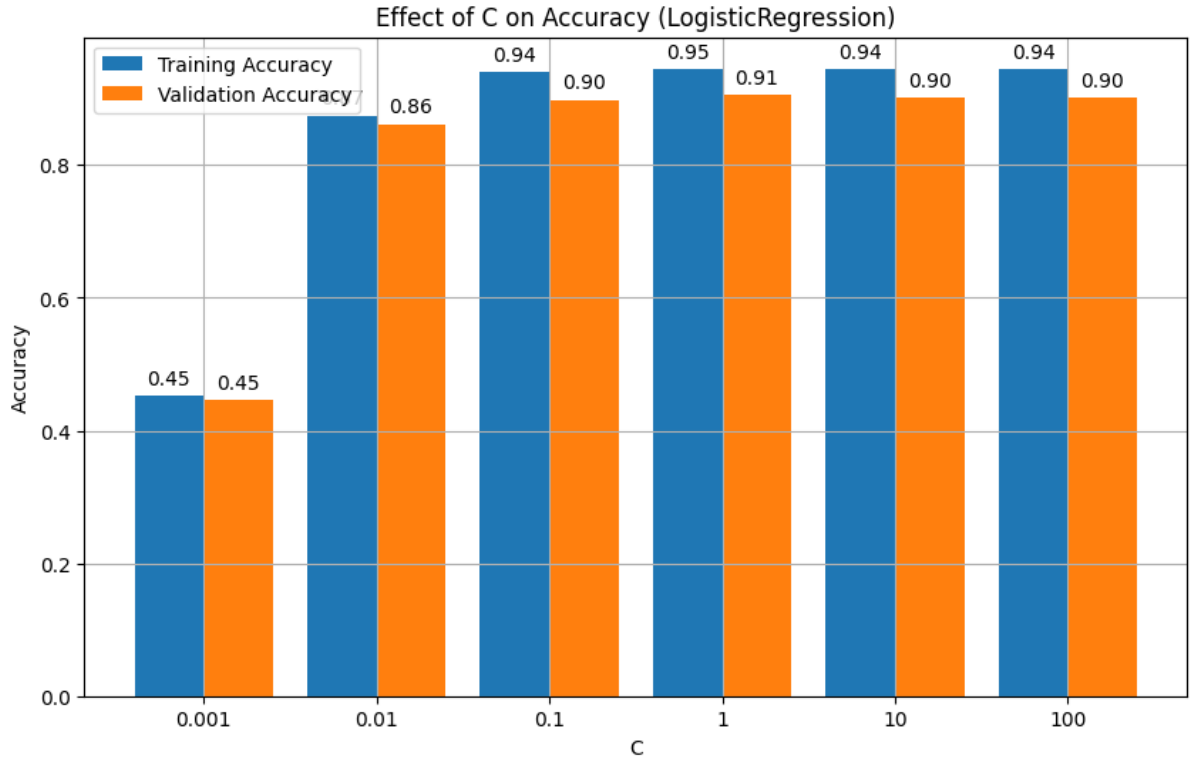


We notice that training and validation accuracy remain consistently high across all penalty conditions, varying minimally, indicating a strong fit of the model to the training data regardless of penalty. A slight accuracy gain is observed for the Elastic Net when looking at both the training and validation accuracy. Again, the consistent gap between training and validation accuracy across all penalties may suggest little overfitting, but the choice of penalty did not substantially alter this.

Furthermore, we investigated the effect of the `C` hyperparameter, which controls the inverse of regularization strength on the model. We tested `C` for a range of values (0.001, 0.01, 0.1, 1, 10, 100) and kept the following hyperparameter values constant:

```
"l1_ratio": 0.5, "max_iter": 800, "multi_class": "auto",
"penalty": "elasticnet", "solver": "saga"
```

The following graph demonstrates our findings on a range of values for C:

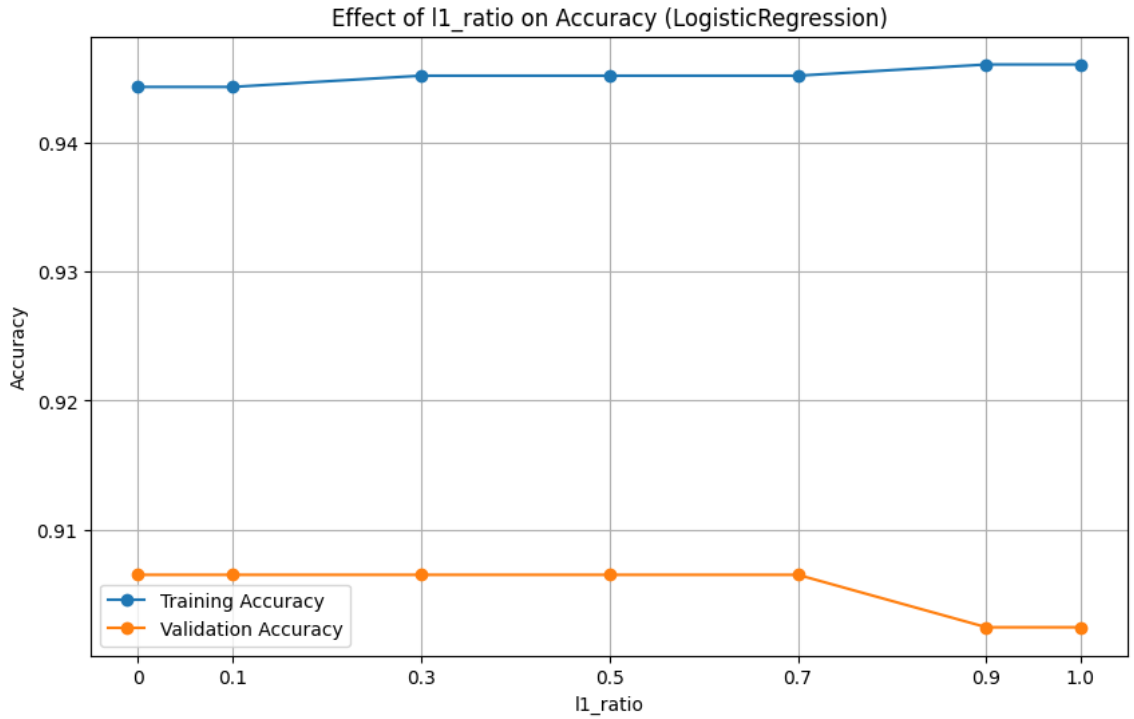


We notice that at a very low C value, the model accuracy is notably poor, suggesting strong regularization leads to significant underfitting. For C values after 0.001, the training and validation accuracy remain consistently similar, indicating the model's ability to fit the training data plateaus after a certain point. The accuracy of model training and validation is topped at a C value of 1, suggesting that a moderate regularization strength is optimal for the generalization of our model. Higher C values may lead to an overly flexible model, increasing the chance of overfitting.

Finally, we tested the `l1_ratio` hyperparameter, controlling the balance between L1 and L2 regularization in Elastic Net. For this test, we tested the hyperparamter for a range of values (0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0) and kept the following hyperparameter values constant:

```
"C": 1, "max_iter": 800, "multi_class": "auto", "penalty": "elasticnet",
"solver": "saga"
```

The following graph demonstrates our findings on a range of L1 ratio values:



We notice that the training and validation accuracy remain consistently high across most tested values from 0 to 0.7, indicating the model's fit to the training data is relatively insensitive to variations in the L1/L2 balance within this range. However, validation accuracy experiences a noticeable decline as `l1_ratio` approaches 1.0. This drop suggests that relying heavily on L1 regularization negatively affects the model's ability to generalize to unseen data, highlighting the importance of balancing L1 and L2 regularization for optimal performance.

Using our findings from observing the effect of each hyperparameter individually, we constructed the following GridSearch `param_grid`. The evaluation metric used during grid search was 5-fold cross-validation accuracy. The best combination of hyperparameters were as follows:

Table 4.3: Grid Search Parameters & Best Logistic Regression Configuration

```
param_grid = {
    "C": [1],
    "max_iter": [100, 125, 150, 175, 200],
    "solver": ["saga"],
    "penalty": ["elasticnet"],
    "l1_ratio": [0.1, 0.3, 0.5, 0.7],
    "multi_class": ["auto"]
}
```

Parameter	Value
C	1
l1_ratio	0.1
max_iter	175
multi_class	auto
penality	elasticnet
solver	saga

Model Performance

With our best hyperparameter values, we achieved a validation accuracy of 90% and a test accuracy of 92%.
To observe our results in greater detail, we generated the following classification report and confusion matrix:

Class	Precision	Recall	F1-score
pizza	0.90	0.93	0.92
shawarma	0.87	0.87	0.87
sushi	0.94	0.91	0.93

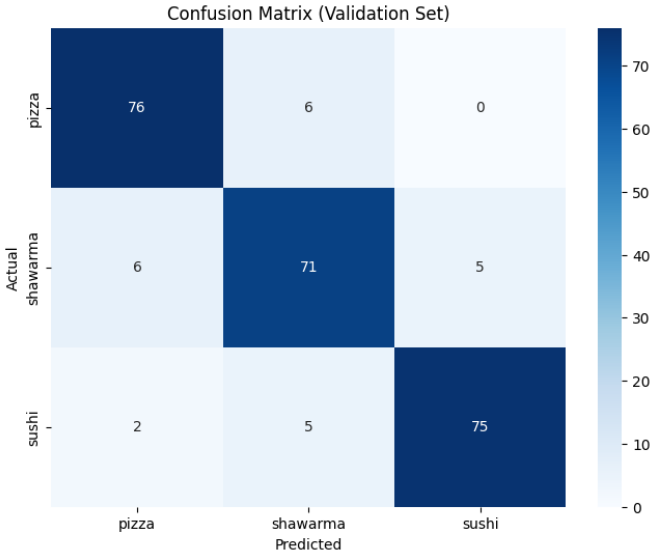


Figure 4.3: Test Set Classification Report (left) & Confusion Matrix (right)

From the classification report, we can see that the model demonstrates varying levels of effectiveness across the different food categories. Notably, sushi achieves the highest precision and recall, whereas shawarma achieves relatively lower precision and recall. This observation is further corroborated by the confusion matrix, which shows the degree of confusion between pizza and shawarma, with 6 instances of each being misclassified as the other. While the model maintains a strong ability to correctly classify pizza, the combined evidence from the classification report and confusion matrix point to shawarma as the category where the model struggled more predicting accurately.

Furthermore, investigating the most important features, the model has determined the following levels of feature importance:

Feature	Importance
q5_sushi_prob	1.029498
q5_shawarma_prob	0.995039
q5_pizza_prob	0.929739
q3_at_a_party	0.875122
q6_sushi_prob	0.763292
q6_pizza_prob	0.577093
q3_week_day_lunch	0.347140
q7_teachers	0.320352
q3_weekend_dinner	0.299439
q8_processed	0.278060
q2_pizza_prob	0.274820
q2_sushi_prob	0.266970
q2_shawarma_prob	0.263807
q6_shawarma_prob	0.244171
q3_late_night_snack	0.230868
q7_friends	0.219973
q7_strangers	0.196251
q7_parents	0.179831
q1_processed	0.107726
q2_ingredient_count	0.068118
q7_siblings	0.061960
q3_weekend_lunch	0.049474
q3_week_day_dinner	0.039112
q4_processed	0.027880

Table 4.4: Feature Importance Rankings

The feature importance rankings confirm that question 5 (associated movie) is the strongest predictor, aligning with our analysis in Section 3.5, where Q5 probability distributions were highly distinctive across food categories. In contrast, question 6 (associated drink) features rank lower, reflecting the overlap observed in Section 3.6. This reinforces our conclusion that movies provide more exclusive food associations, making them stronger classification features than drinks.

Since we used elastic net regularization, the relatively low L1 ratio (0.1) implies a stronger emphasis on shrinking less important features towards zero rather than outright elimination. The model reduced the magnitude of all coefficients, with the most important feature retaining their relatively higher influence.

In summary, we have seen that the Logistic Regression model demonstrates promising classification performance, achieving high accuracies on validation and test sets. The model offers several advantages including its interpretability, computational efficiency, and ability to provide insights into feature importance. However, we must acknowledge the limits this model has. It assumes a linear relationship between the features, which can limit its performance when dealing with complex, non-linear relationships in the data. Notably, our analysis was conducted without feature normalization, as experiments with normalization did not yield significant improvements in model performance. Additionally, while the use of regularization can reduce the chance of overfitting, the model's success is heavily dependent on the appropriate selection of hyperparameters, and data quality. Overall this model provides a solid benchmark to compare to other models tested.

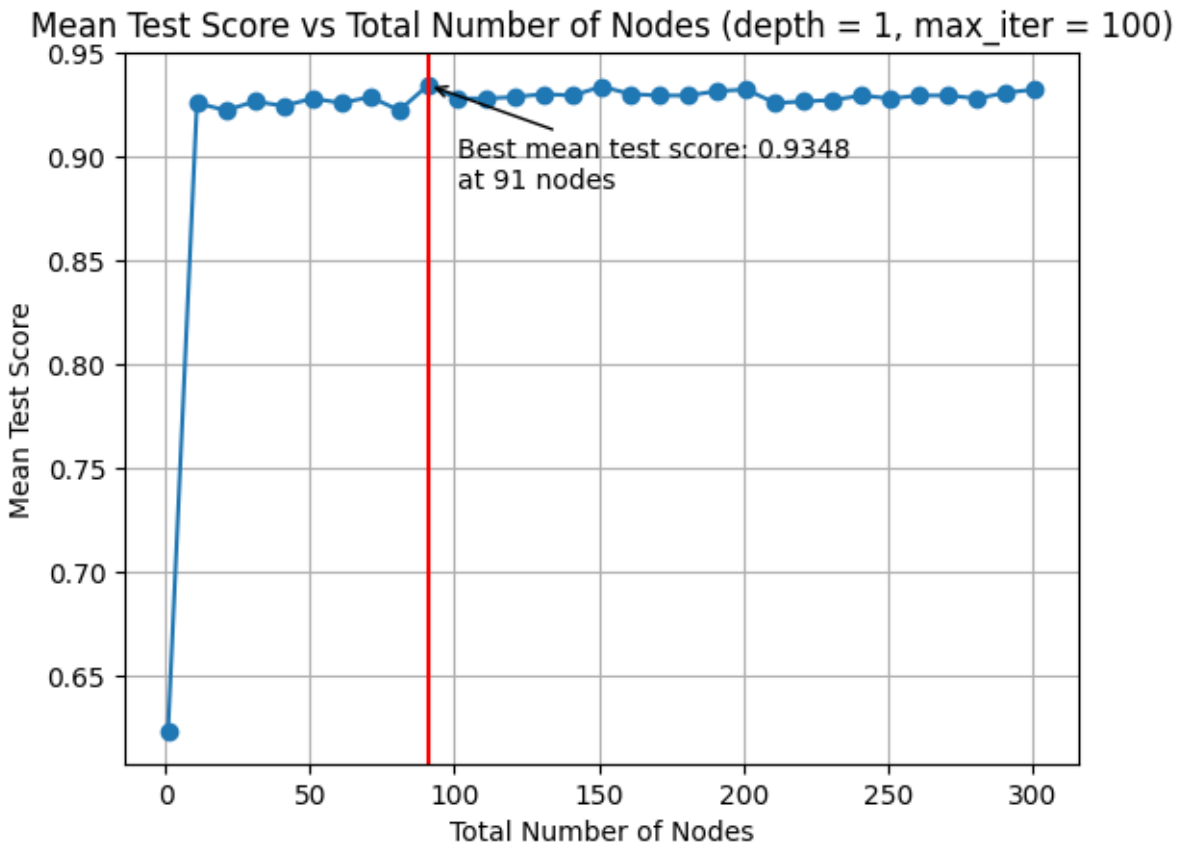
4.4 Multi-layer Perceptrons (MLP)

The code accompanying the following analysis can be found in `explore_mlp.ipynb`.

MLP models are difficult to train due to the large amount of hyperparameters that can be tuned. These include the depth of the network, the amount of nodes in each layer, and the activation function for each layer. Executing a grid search across all these parameters would take an unreasonably long time. To avoid this, We want to narrow down the range of hyperparameters that give the best results. Then, we can perform grid search on this smaller domain. We will describe our approach.

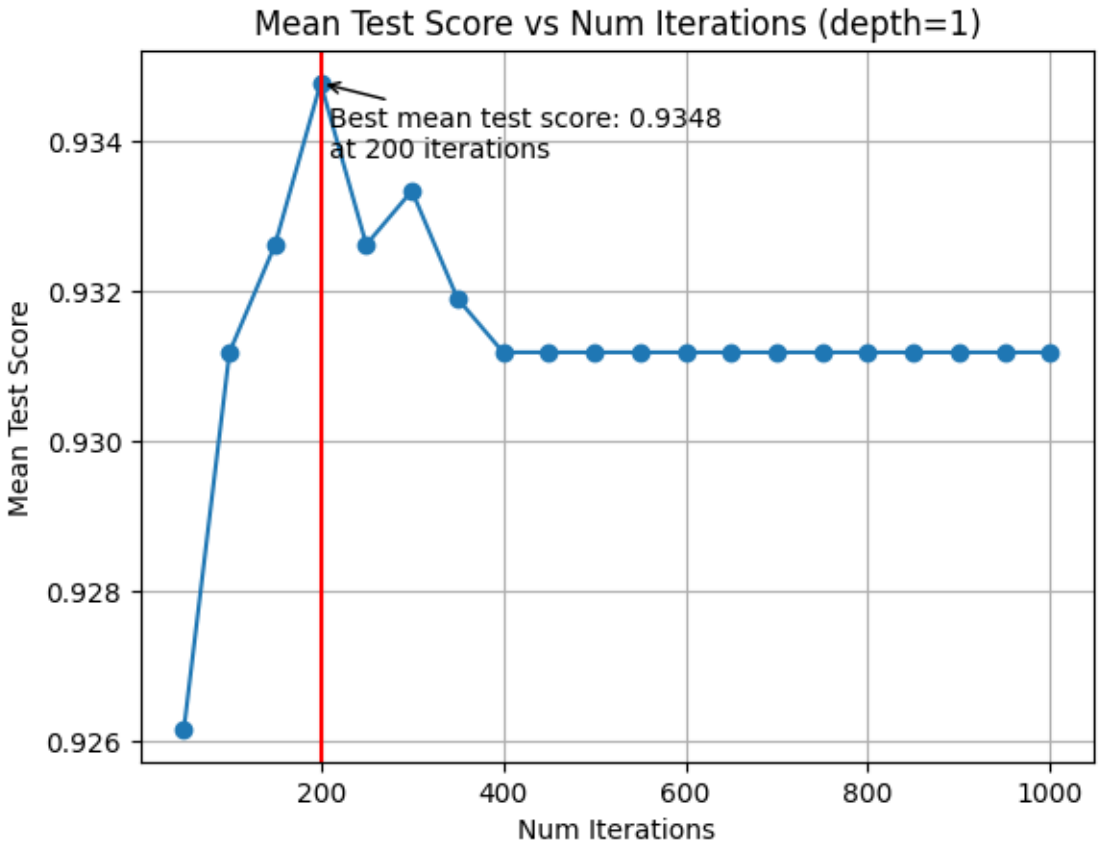
Training Iterations

The first hyperparameter we wanted to narrow down is the amount of training iterations. Determining this early will allow us to pick the smallest amount of iterations where the model converges, ensuring that we don't waste time training the model in our future tests. To do this, we created a well performing base model to test on. We tested all the one deep neural networks ranging from 1 to 302 nodes. The solver was Stochastic Gradient Descent, and the activation function was RELU. We found that at 91 nodes the model performs the best.



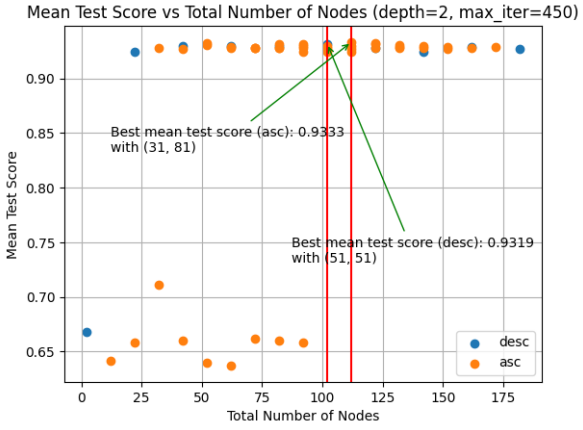
Throughout our tests, mean test score refers to the mean score the model had on 5 random iterations of training and testing on a validation set.

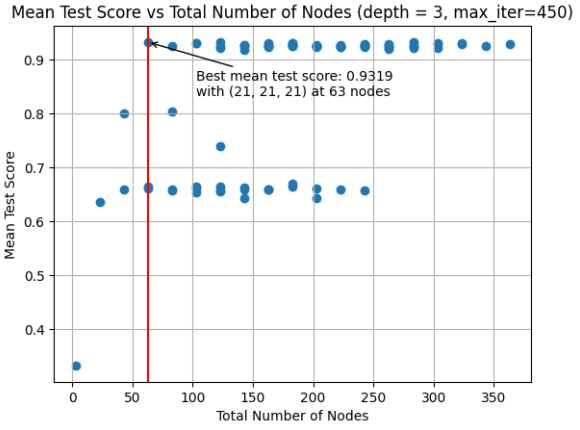
Now that we have a well performing base network, we can test it on training iterations ranging from 50 to 1000. We found that after 400 iterations, the model converges. So, for the rest of testing we stuck with 450 iterations.

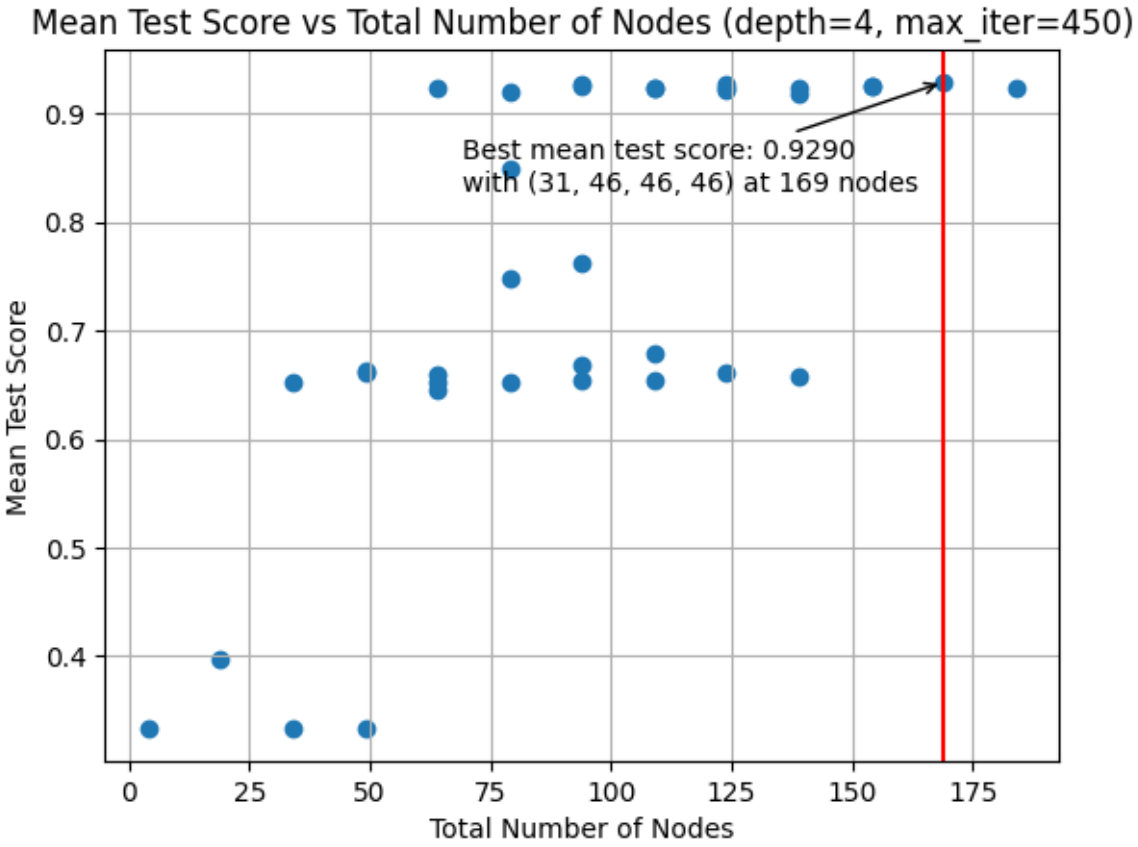


Depth

The next hyperparameter to tune is the depth of the network. We already tested how a one deep network performs. Now we test how a 2, 3, and 4 layer network performs.





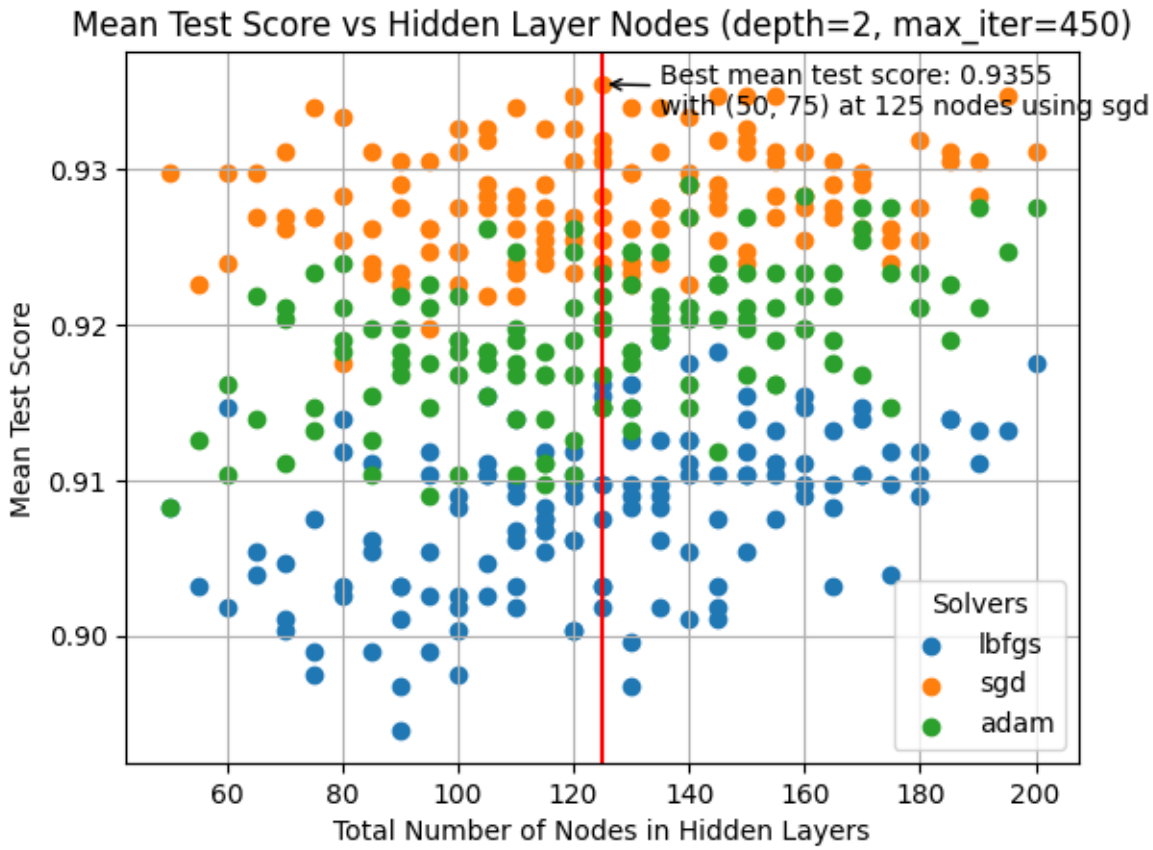


We found that the one layer network performs the best, with the mean test score decreasing as we increase the amount of layers. This indicates that the data is very simple, and adding more layers causes the model to overfit. Do note however that the difference in performance between all the networks is negligible, and it could be argued that if the grid search was finer for each test, we could have found better scores. In summary, we concluded that it is best to use a two layer network because it should be simple enough to not overfit, but allows room for complexities in the data.

When testing the two deep network, we wanted to see if a strictly increasing amount of nodes per layer has any performance difference than a balanced or strictly decreasing amount of nodes per layer. The results are graphed above, and there is no significant difference, so we did not continue exploring this.

Grid Search

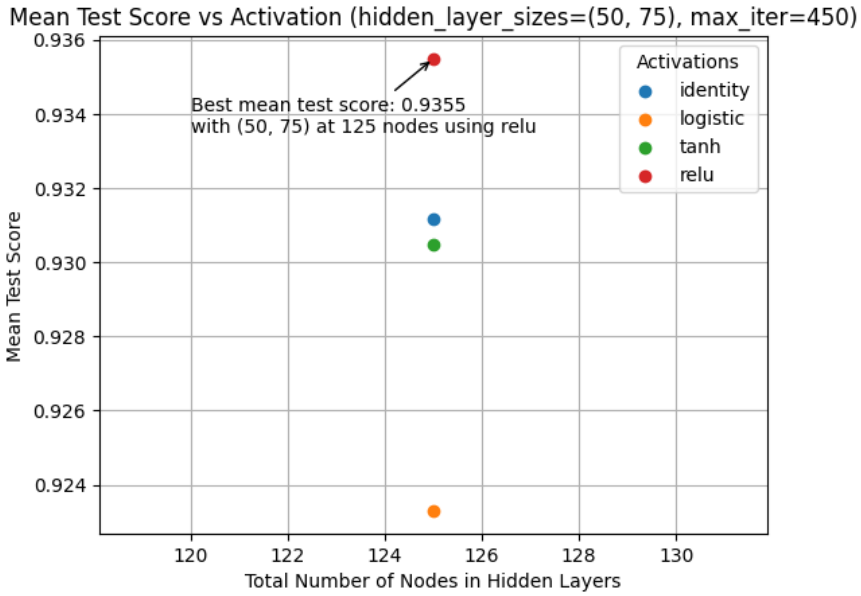
Now that we have narrowed down the well performing range of hyperparameters, we performed a finer grid search. We tested every possible two layer network with the amount of nodes in each layer ranging from 25 to 100 with a step size of 5. Each network was trained using LBFGS, SGD, and ADAM in order to also find the best solver.



The finer grid search was able to score higher than the one layer neural network that we tested in the beginning. This confirms that it was not incorrect for us to go with the two layer network even though the one layer network scored higher in our initial testing. Since the differences in test scores are so small, the finer grid search was able to find a slightly better choice of hyperparameters.

Activation Function

The last hyperparamater to tune is the activation function. Using the best model we just found, we tested it using identity, logistic, tanh, and RELU activation functions.



The best activation is RELU.

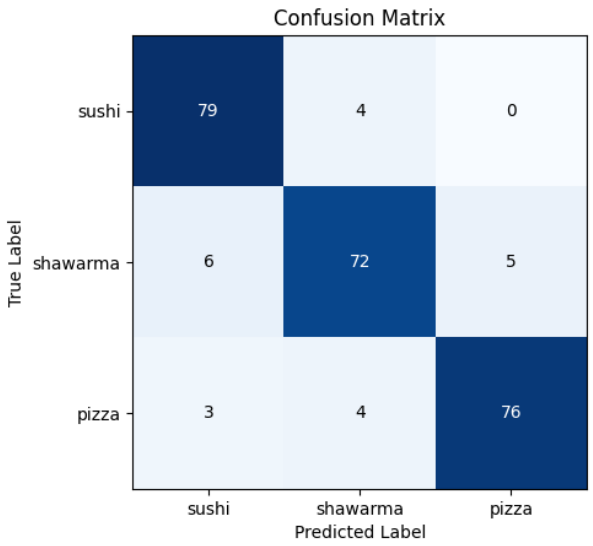
Results

Our final hyperparameters are

Parameter	Value
solver	sgd
hidden_layer_sizes	(50, 75)
activation	relu
max_iters	450

Here is how the model performs on a test set:

Class	Precision	Recall	F1-score
pizza	0.90	0.95	0.92
shawarma	0.90	0.87	0.88
sushi	0.94	0.92	0.93



Our model demonstrates strong performance across all three food classes. Shawarma scores abnormally low recall relative to the other classes. Meaning the model mistakes shawarma for another food class more than it makes the same misclassification for sushi and pizza.

In conclusion, we were able to build a strong MLP model by systematically narrowing down the best hyperparameters, then performing a fine grid search. We found that in general, no matter the hyperparameters, the model performs very well, with differences being negligible. We doubt that a significantly better MLP model can be built without overfitting the dataset.

4.5 Random Forests

The code for the following analysis can be found in `explore_random_forest.ipynb`.

With the Decision Tree performing reasonably well, we wanted to see if we could squeeze out more performance by using multiple of them. So, we decided to use the Random Forest, which will train multiple decision trees on random subsets of the data and features. Each decision tree will then make a prediction, with the final prediction being the popular prediction.

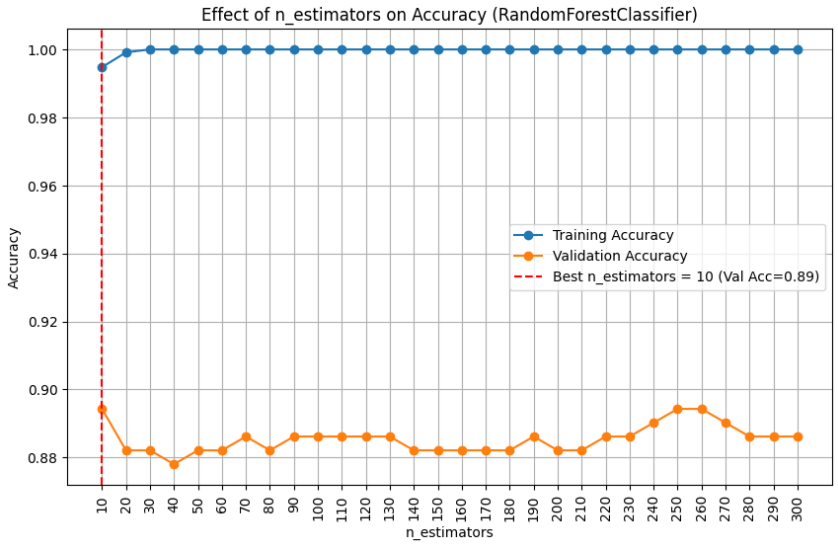
Hyperparameter Exploration

Just like with decision trees, we want to first analyze the impacts on accuracy of the following hyperparameters:

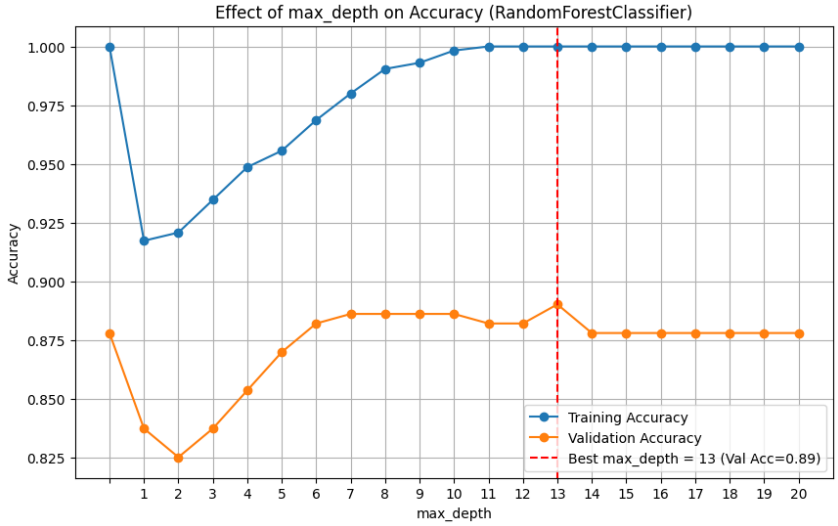
```
"n_estimators": number of trees in the forest
"max_depth": max height of any tree
"max_features": number of features to consider when looking for the best split
"min_samples_split": min number of samples required to split a non-leaf node
"min_samples_leaf": min number of samples required to be a leaf node
"criterion": the function to measure the quality of a split
```

This time, we wanted to be more conscious about the more important hyperparameters, and provided GridSearch with more values for those particular hyperparameters. As a result, GridSearch is provided with one or two values of the less important hyperparameters. This is to reduce the number of unnecessary hyperparameter combinations.

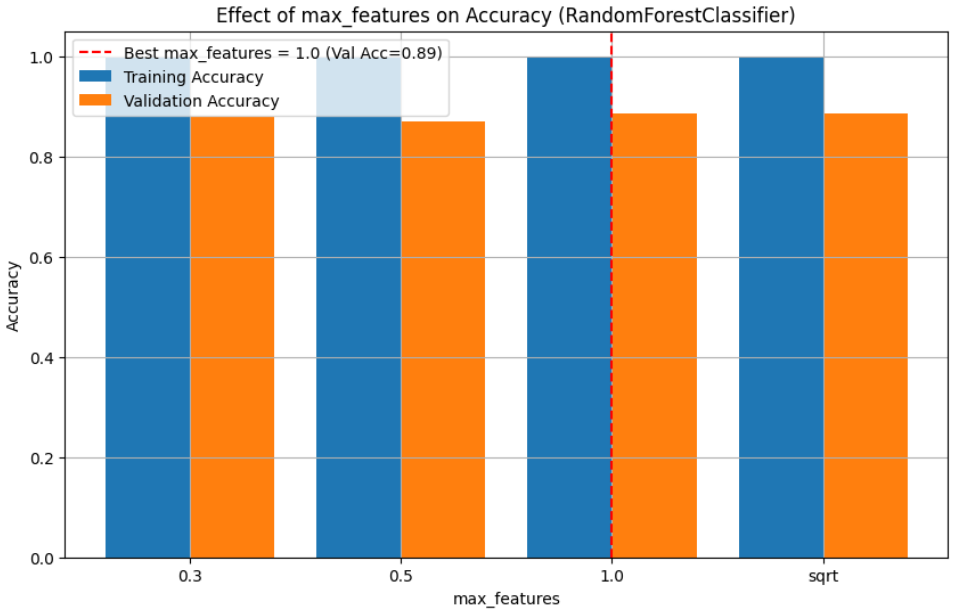
Similar to decision trees, we trained our random forest models on a training set with all features, in order to provide our models with as much information when learning. This helps especially with random forests, which will need a large variety of subsets of data and features when training the decision trees within them. The results of our hyperparameter analysis are as as follows:



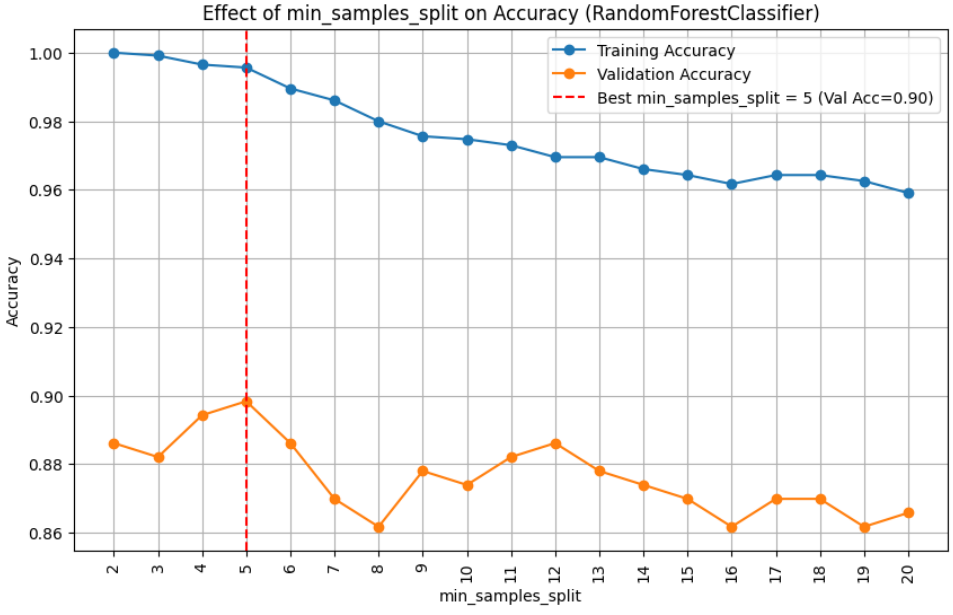
With `n_estimators` we can see that training accuracy is staying at a near consistent 1.0. Validation accuracy is fairly consistent, with accuracies tending between 0.88 and 0.89. Validation accuracy achieves an absolute max of 0.89 at a `n_estimators` values of 10 and 250, with a slightly smaller peaks occurring at 70, and 190. The difference in validation accuracy across all of these values is somewhat small, but since `n_estimators` is an vital parameter, we may want to consider a large range of value for GridSearch. We will consider `n_estimators` values from 100 (the default) to 300 for GridSearch.



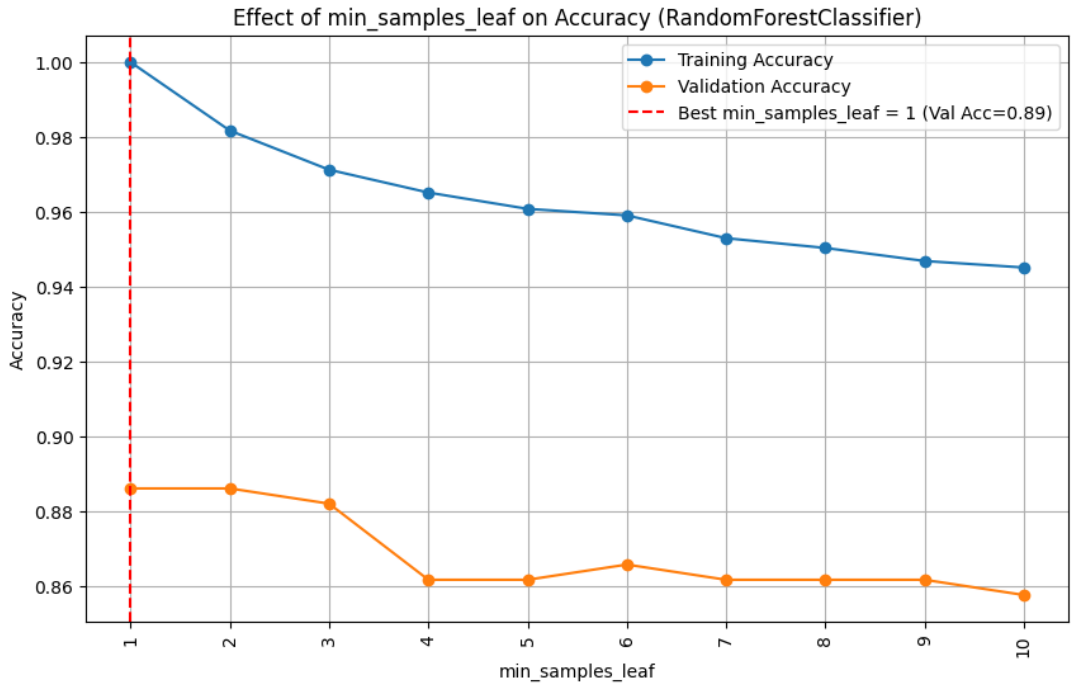
With `max_depth` we can see that both the training and validation accuracies follow a similar trend, steadily increasing, then asymptotically approaching 1.0 and 0.875 respectively, as `max_depth` increases. The validation accuracy achieves a rather sudden absolute max of 0.89 at a `max_depth` value of 13. The only other smaller peak is of about 0.875, when `max_depth` is `None`. For GridSearch, we set `max_depth = None` to allow individual trees to grow fully and overfit. This leverages the ensemble effect of Random Forests, where multiple overfitted trees collectively improve generalization through averaging.



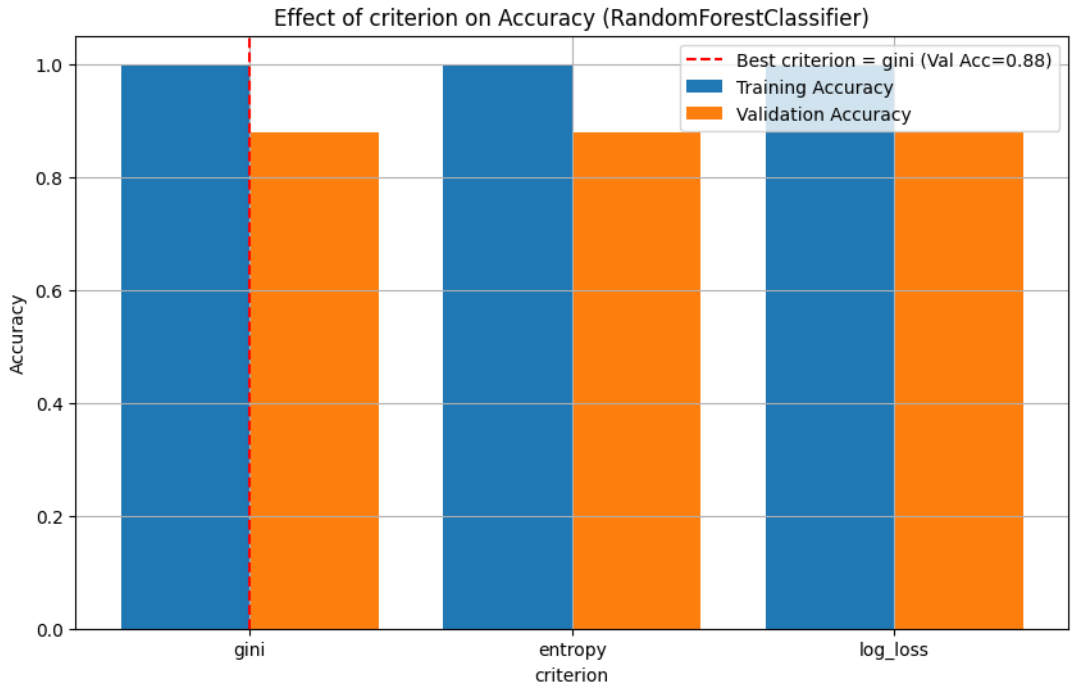
For `max_features`, we can see that the training accuracy remains a constant 1.0, and the validation accuracies tend to be around 0.89, with the absolute max of 0.89 occurring at a `max_features` value of 1.0. Since `max_features` is a particularly important hyper parameter, and all values of `max_features` produce very similar validation accuracies, we will consider all of these values for GridSearch.



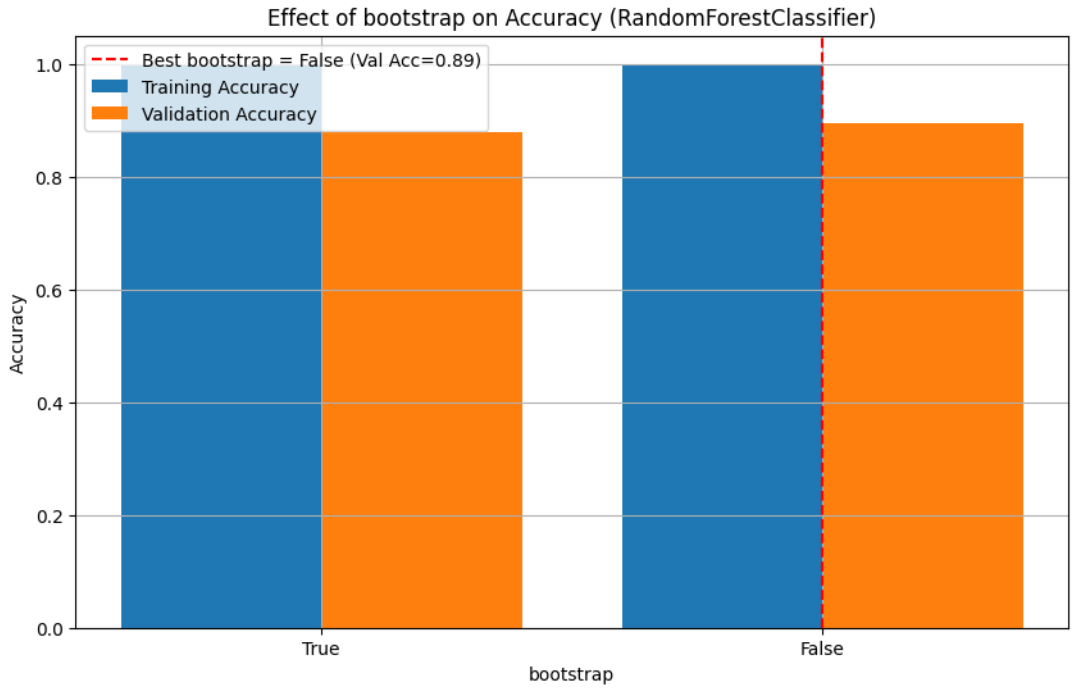
For `min_samples_split`, we can see that the training accuracy steadily decreases, asymptotically approaching 0.96. This behaviour seems to show the model slowly stabilizing. Validation accuracy tends oscillate and approach 0.87, with an absolute max of 0.9 occurring at a `min_samples_split` value of 5, and smaller peaks of 0.88 and 0.89 occur at 9 and 12. However, we will only consider `min_samples_split=2` for GridSearch, as we want to fully develop the trees in the forest.



For `min_samples_leaf`, we can see that both the training accuracy steadily decrease as `min_samples_leaf` increases. Validation accuracy achieves an absolute max of 0.89 at 1. Given that Random Forests benefits from deep and fully overfitted trees, we will consider `min_samples_leaf = 1` for GridSearch.



For `criterion`, we can see that all three values have very similar training and validation accuracies, with `gini` just about achieving the absolute max of 0.88. Since all three values have very similar training and validation scores, we will consider them all for GridSearch.



For `bootstrap`, we can see that both values have very similar training and validation accuracies, with `False` just about achieving the absolute max of 0.89. Because of the similarity in accuracies, we will be considering both `True` and `False` for GridSearch.

To obtain more reliable hyperparameter estimates, we conducted a grid search using 5-fold GroupKFold cross-validation, motivated by the discussion of our previous analysis. The search space and the best-found configuration are presented below:

Table 4.5: Grid Search Parameters and Best Random Forest Configuration

```
param_grid = {
# IMPORTANT
"n_estimators": list(range(100, 301, 10)),
"max_features": [0.3, 0.5, "sqrt", 1.0],
"max_depth": [None],
"min_samples_split": [2],
"min_samples_leaf": [1],
"bootstrap": [True, False],
"criterion": ["gini", "entropy", "log_loss"],
"random_state": [42]
}
```

Parameter	Value
bootstrap	True
criterion	gini
max_depth	None
max_features	sqrt
min_samples_leaf	1
min_samples_split	2
n_estimators	100
random_state	42

This model achieved 93.62% cross-validation accuracy. On the held-out test set, it maintained, strong generalization with 89.95% accuracy.

Model Performance

Random Forests supports Out-of-Bag (OOB) score, which provides an internal cross-validation score without needing a separate validation set. This is achieved by evaluating each tree on samples not included in its training subset. Our model's OOB score: 93.47%, closely matching the cross-validation accuracy, further confirming its reliability.

Class	Precision	Recall	F1-score
pizza	0.92	0.93	0.92
shawarma	0.83	0.90	0.87
sushi	0.96	0.87	0.91
Accuracy			89.95%

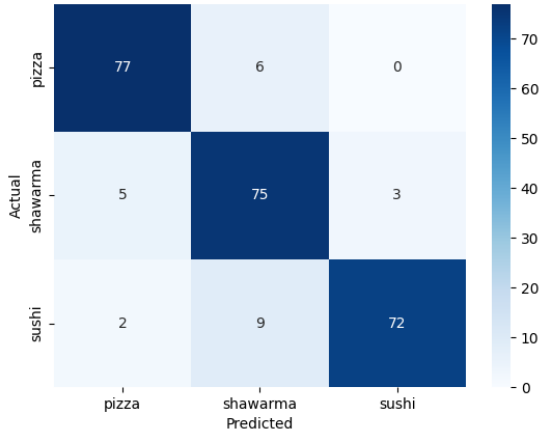


Figure 4.4: Test Set classification report (left) and confusion matrix (right)

The Random Forest classifier delivers strong performance across all three food categories. Pizza achieves the highest recall at 93%, meaning the model correctly identifies nearly all actual pizza samples. Sushi, on the other hand, has the highest precision at 96%, indicating that when the model predicts sushi, it does so with high confidence and minimal misclassification. Shawarma demonstrates solid recall at 90%, correctly identifying most shawarma samples, but its precision is lower at 83%, suggesting that some non-shawarma samples are misclassified as shawarma. Despite this, the F1-scores for all classes are around 90%, highlighting the model's ability to balance precision and recall effectively across categories.

Overall, the RandomForestClassifier demonstrated excellent performance on our food classification task and is a great candidate for our final model.

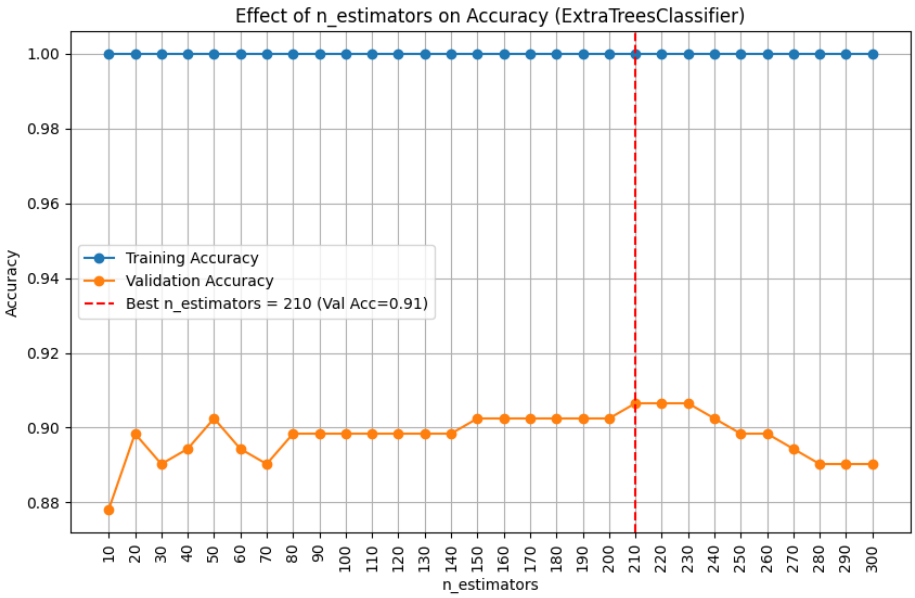
4.6 Extra Trees Classifier

The code accompanying the following analysis can be found in `explore_extra_trees.ipynb`.

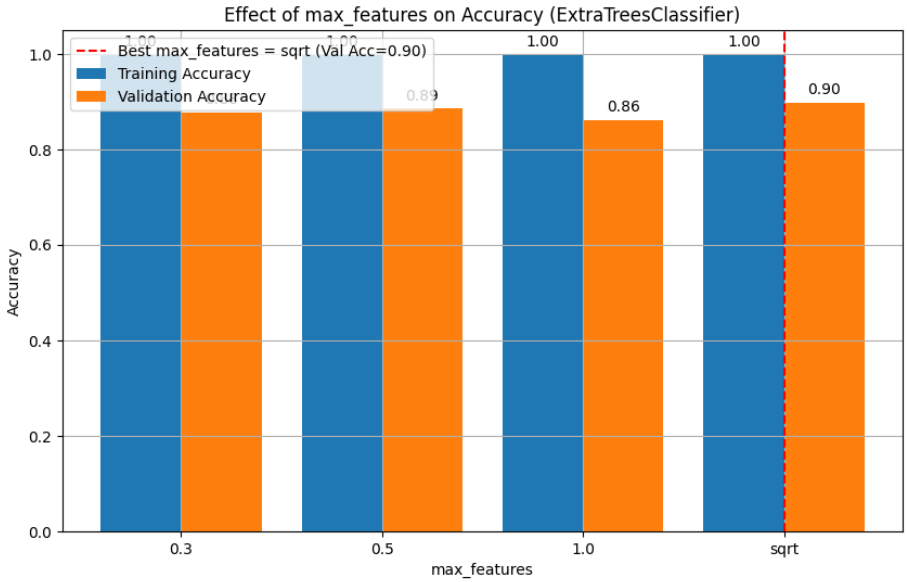
The ExtraTreesClassifier is an ensemble method that builds upon the RandomForestClassifier by introducing additional randomization during the split selection process. While random forests search for the optimal split point among a subset of features, Extra Trees selects split points completely at random from the feature's empirical range. This extra level of randomness can help reduce variance and improve generalization.
As non-parametric universal function approximators, Extra Trees can model complex relationships without strong distributional assumptions, given sufficient data and trees. The ensemble combines many deliberately overfit trees (weak learners) to create a strong, robust predictor through averaging effects.

Hyperparameter Exploration

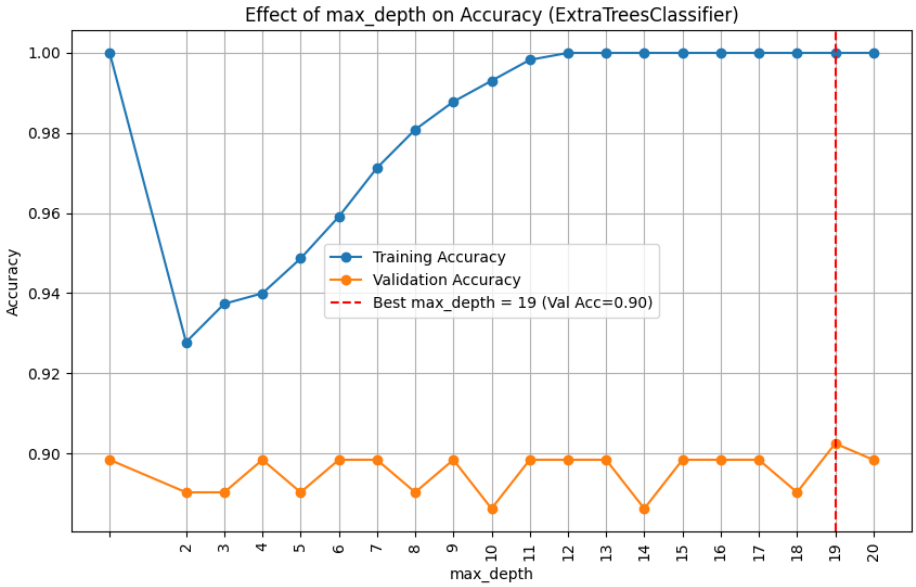
We first explored the effect of different hyperparameters by varying one at a time and analyzing their impact on validation accuracy.



`n_estimators`: Controls the number of trees in the ensemble which is a fundamental hyperparameter. A larger number of trees reduces variance but increases computational cost. From our results, we observe that validation accuracy is best between 150-250 estimators. However, performance remains consistently high even outside this range, indicating robustness to this parameter.

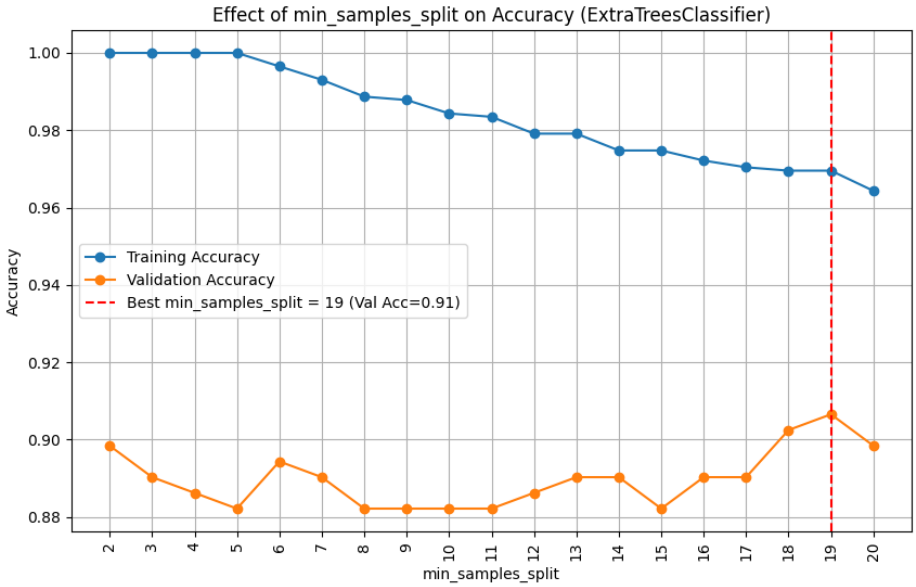


`max_features`: Controls the number of randomly selected features considered for splitting at each node. A smaller value increases randomness, reducing variance but potentially increasing bias, while a higher value allows for better feature exploitation. The default recommendation for classification tasks is max_features = $\sqrt{\text{n\_features}}$. Our results confirm this, showing that the best validation accuracy is achieved with max_features=sqrt.

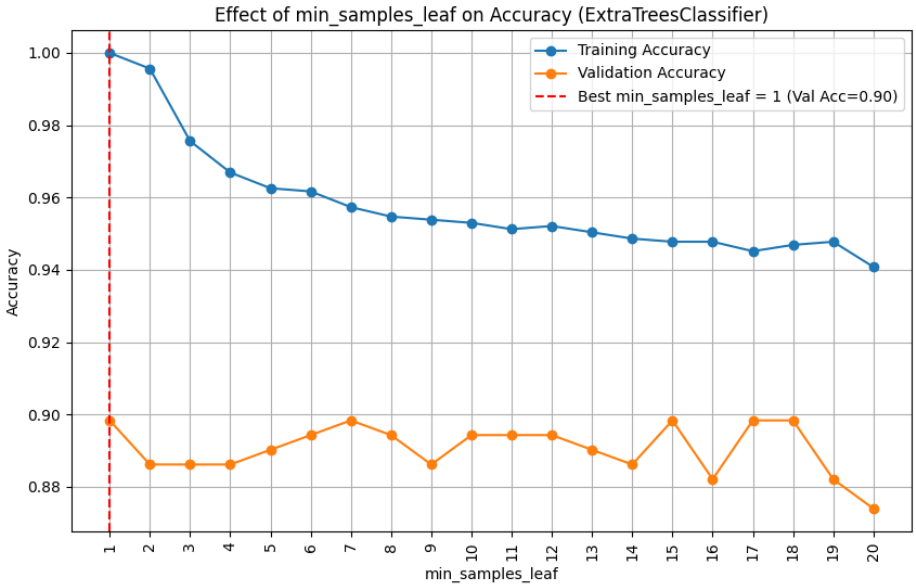


`max_depth`: Controls the depth of individual trees affects the bias-variance tradeoff. Smaller depths may lead to underfitting, while deeper trees tend to overfit. The first value in the plot corresponds to unrestricted depth (None), where trees grow until they reach pure leaf nodes. Since Extra Trees relies on an ensemble effect, we would want to

set max_depth=None to allow trees to overfit individually, leveraging the strength of the overall ensemble.

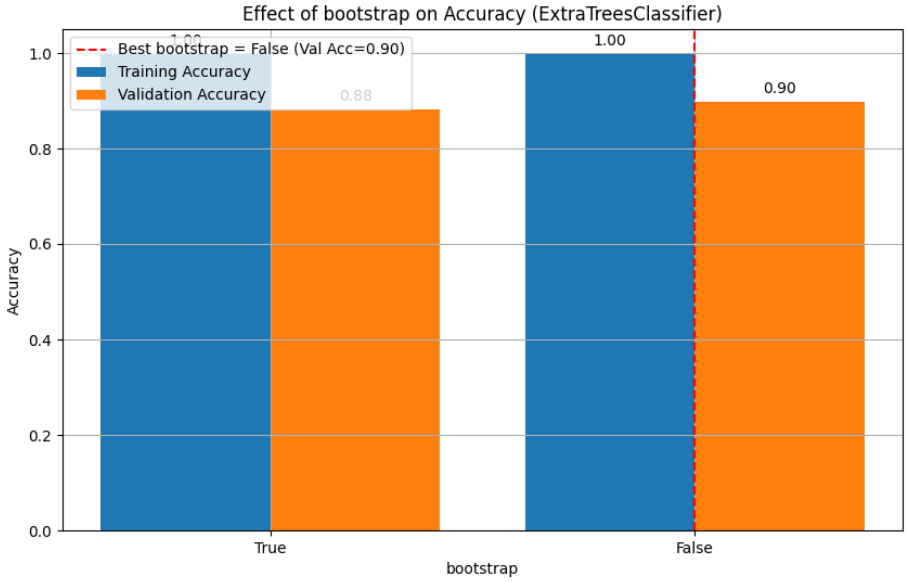


`min_samples_split`: Controls the minimum number of samples required to split a node. A lower value leads to deeper trees, while a higher value restricts growth. Similar to max_depth, we would want to set min_samples_split=2 to allow maximum tree growth, ensuring that trees are fully developed before aggregation.



`min_samples_leaf`: Controls the minimum number of samples required to form a leaf node. A lower value allows finer splits, while a higher value forces leaf nodes to contain

more samples, which can help reduce overfitting. Given that Extra Trees benefits from deep and fully overfitted trees, we would want to set min_samples_leaf=1.



`bootstrap`: Unlike Random Forest, which typically uses bootstrapping (sampling with replacement), Extra Trees does not bootstrap by default. The plot also suggests that disabling bootstrapping seems to be better choice for our dataset. However, the automated GroupKFold cross-validation search later found that enabling bootstrapping improved performance slightly, likely due to additional randomness helping generalization.

To obtain more reliable hyperparameter estimates, we conducted a grid search using 5-fold GroupKFold cross-validation, motivated by the discussion of our previous analysis. The search space and the best-found configuration are presented below:

Table 4.6: Grid Search Parameters and Best Extra Trees Configuration

```
param_grid = {
# IMPORTANT
"n_estimators": list(range(100, 251, 10)),
"max_features": [0.3, 0.5, 1.0, "sqrt"],

"max_depth": [None],
"min_samples_split": [2],
"min_samples_leaf": [1],
"bootstrap": [True, False],
"criterion": ["gini"],
"random_state": [42]
}
```

Parameter	Value
bootstrap	True
criterion	gini
max_depth	None
max_features	sqrt
min_samples_leaf	1
min_samples_split	2
n_estimators	150
random_state	42

This model achieved 93.62% cross-validation accuracy. On the held-out test set, it maintained, strong generalization with 91.97% accuracy.

Model Performance

Extra Trees supports Out-of-Bag (OOB) score, which provides an internal cross-validation score without needing a separate validation set. This is achieved by evaluating each tree on samples not included in its training subset. Our model's OOB score: 93.91%, closely matching the cross-validation accuracy, further confirming its reliability.

Class	Precision	Recall	F1-score
pizza	0.91	0.96	0.94
shawarma	0.89	0.90	0.90
sushi	0.96	0.89	0.93
Accuracy			91.96%

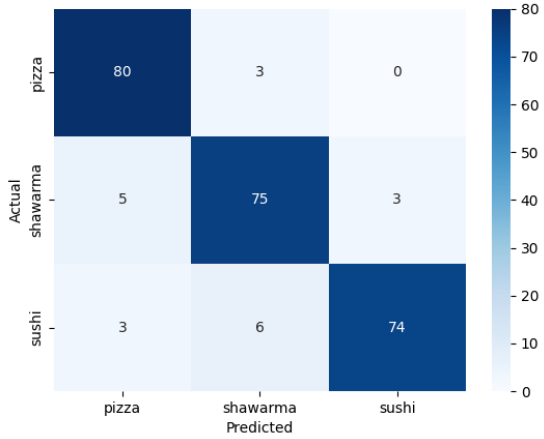


Figure 4.5: Test Set Classification Report (left) & Confusion Matrix (right)

The Extra Trees classifier demonstrates strong performance across all three food categories. Pizza achieves the highest recall (96%), meaning that nearly all actual pizza samples are correctly identified. Sushi, on the other hand, has the highest precision (96%), indicating that when the model predicts sushi, it does so with high confidence. Shawarma exhibits a balanced performance, with both precision and recall around 90%, ensuring consistent classification accuracy. The F1-scores for all classes exceed 90%, highlighting the model's ability to maintain a strong trade-off between precision and recall across categories.

Overall, the ExtraTreesClassifier demonstrated excellent performance on our food classification task and is a great candidate for our final model.

5 Final Model Selection

After exploring several models, including k-Nearest Neighbors (kNN), Decision Trees, Logistic Regression, Multi-Layer Perceptron (MLP), Random Forest, and Extra Trees, we narrowed our focus to Random Forest and Extra Trees as the most promising candidates.

Tree-based models are universal function approximators, capable of capturing complex relationships without making strong assumptions about the underlying data distribution-unlike linear models like Logistic Regression. Additionally, ensemble methods like Random Forest and Extra Trees leverage multiple decision trees, making them more robust and reliable compared to individual models like kNN or MLP. While it is possible to create an ensemble of kNNs or even a heterogeneous ensemble combining kNN, logistic regression, and MLP, one of our goals was to maximize performance while maintaining simplicity and interpretability.

Between Random Forest and Extra Trees, the latter demonstrated slightly better test accuracy (91.97% vs. 89.95%) and a more balanced performance across classes, particularly in Shawarma classification, where Random Forest struggled with precision. Additionally, the higher Out-of-Bag (OOB) score further confirmed Extra Trees' generalization ability.

Although our analysis seems to indicate that Extra Trees outperforms Random Forest, we conducted an additional test to further validate our decision. We evaluated the best configurations for both models across 91 random data splits, training each model on the training set and testing on both the validation and test sets, which, in this scenario, were treated equally as unseen data. The following table summarizes the results:

Table 5.1: Performance Across Multiple Splits: Extra Trees vs. Random Forest

Metric	**Val. (ET)**	**Test (ET)**	**Val. (RF)**	**Test (RF)**
Best Accuracy	95.12%	95.18%	93.50%	93.98%
Average Accuracy	90.69%	90.76%	88.90%	88.96%
Worst Accuracy	85.77%	85.14%	84.55%	83.13%
90%+ Count	59/91	60/91	23/91	22/91

(For more details, refer to the files `ExtraTreesClassifier_BestParams.txt` and `RandomForestClassifier_BestParams.txt`.)

Extra Trees consistently outperformed Random Forest in all areas, with a higher best accuracy, a higher average accuracy, and a significantly greater number of runs exceeding 90% accuracy. Given its consistent performance and robustness across multiple data splits, we confidently select ExtraTreesClassifier as our final model, using the best configuration identified in Table 4.6.

5.1 Teaching Team Test Set Prediction

Based on our extensive evaluation of the Extra Trees Classifier, we anticipate the model will achieve an accuracy of approximately 90% on the teaching team test set. This expectation is supported by the model's consistent performance across 91 random data splits, as detailed in the previous table. Specifically, the Extra Trees model demonstrated an average test accuracy of around 90%, with 60 out of 91 runs (roughly 66%) achieving accuracies exceeding 90%. Given this robust performance across multiple splits, we expect that the model will perform similarly on the teaching team test set.