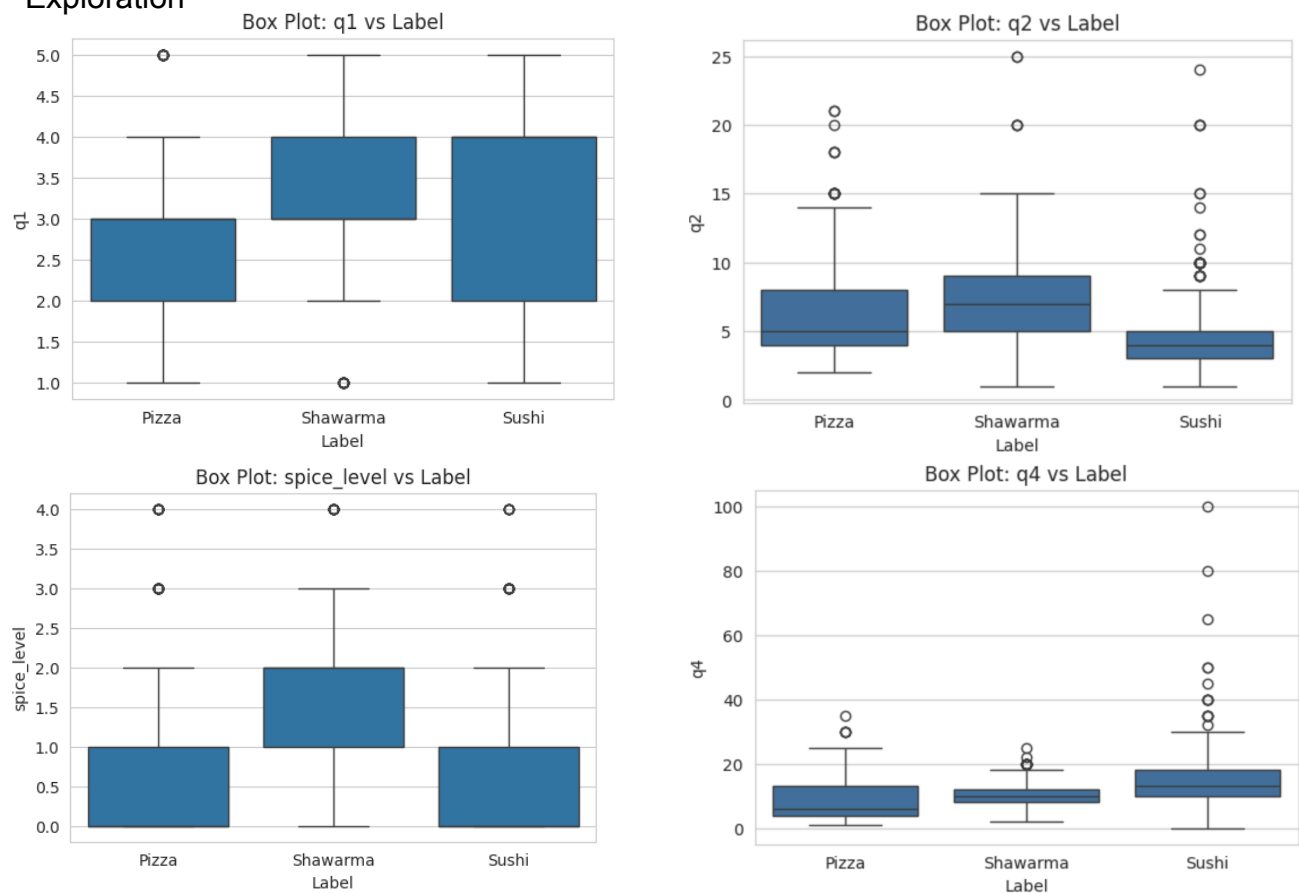


Data

Exploration

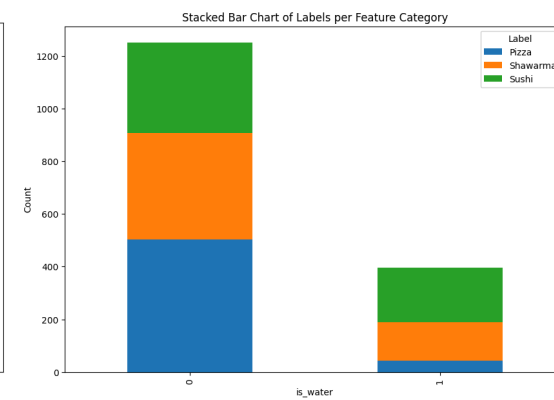
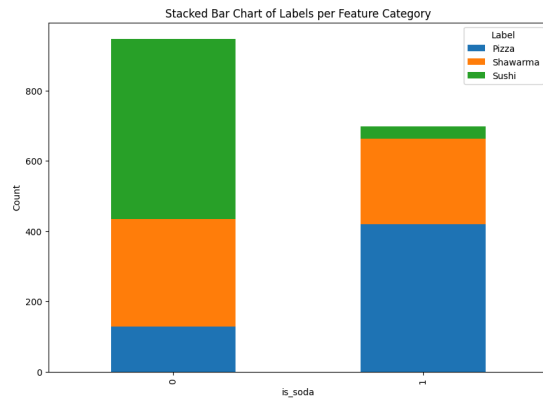
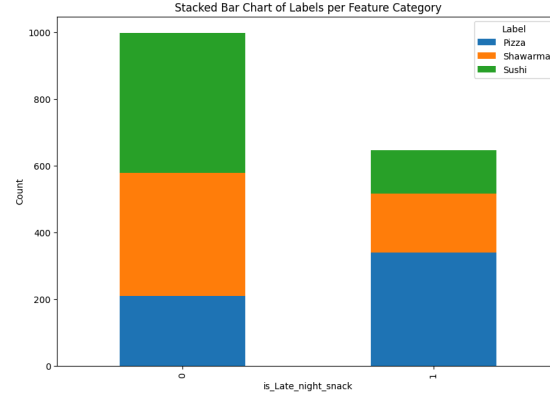
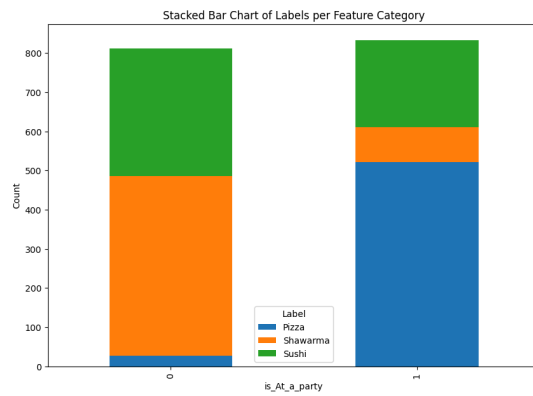
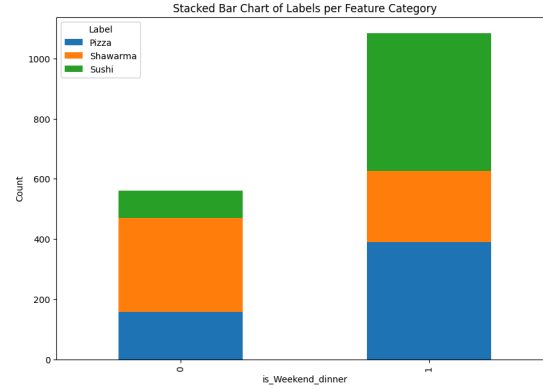
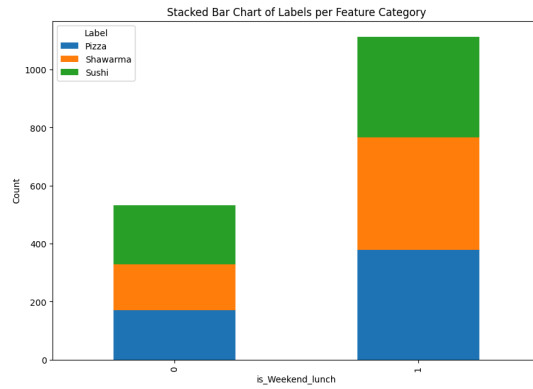
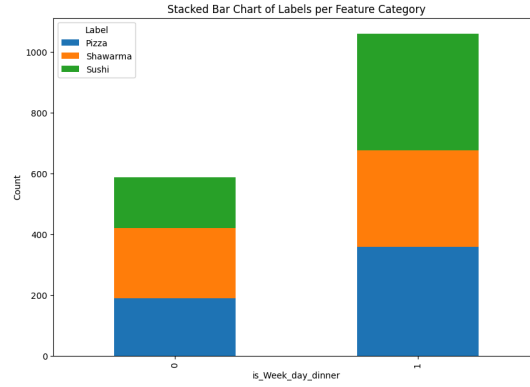
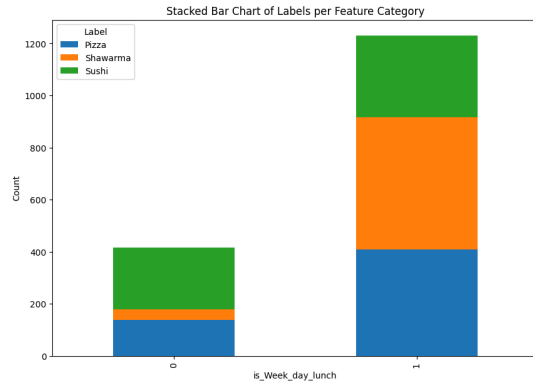


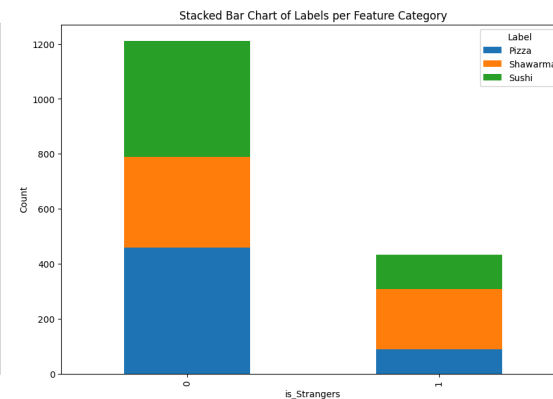
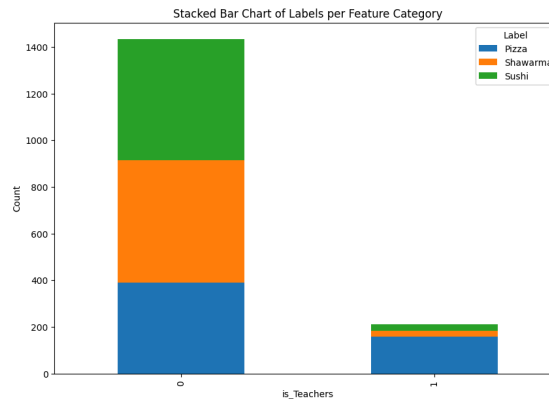
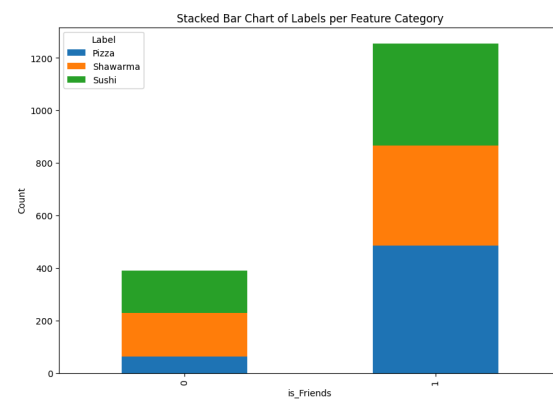
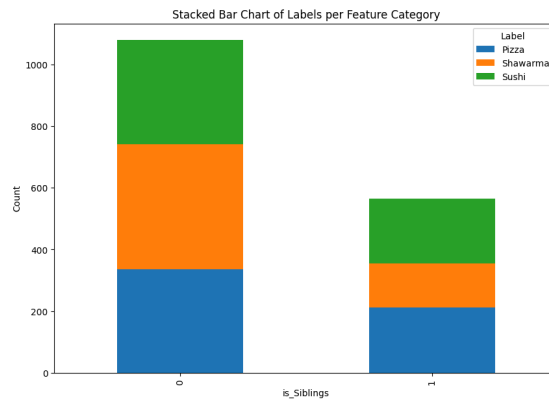
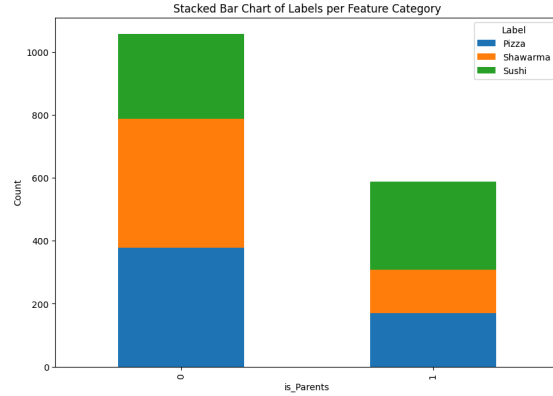
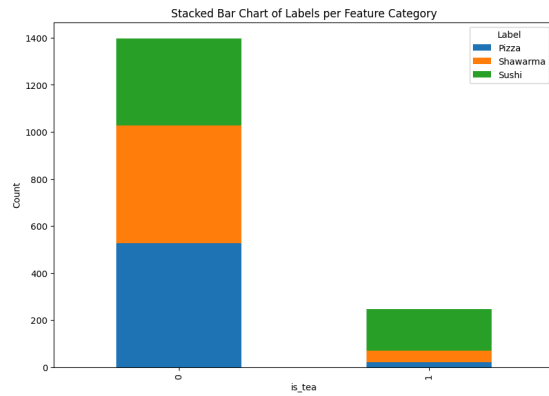
We began by plotting Box plots for each numerical feature to understand their overall distribution across the different food categories (Pizza, Shawarma, Sushi). For instance:

The box plot for Q1 shows that the median difficulty rating for Shawarma is higher than for Pizza, suggesting that respondents generally find Shawarma harder to prepare than Pizza.

The box plot for Q2 shows that the median number of ingredients for Sushi is lower than that for Shawarma, suggesting that people generally associate Sushi with fewer ingredients compared to Swarma. Moreover, Sushi's box plot exhibits more outliers, indicating higher variability in perceptions.

The box plot for spice level shows that the median spice level for Shawarma is higher than for both Pizza and Sushi, suggesting that respondents generally associate Shawarma with a spice.





We also generated Stacked bar charts which allowed us to observe how additional categorical variables (like settings from Q3 or drink pairings from Q6) interact with the food categories. The ratio difference of food categories between bars helped us to identify potential relationships between the context of the food item and its perceived attributes.

Feature Selection and Data Representation

For the given dataset, we noticed that there were many inconsistencies. For example, people would answer Soda or Coke which are the same drinks for question 6. We took initiative to thoroughly clean the dataset to remove these inconsistencies.

This is how we cleaned each column:

- Id column: As id does not give any correlation for the targets, we decided to remove them.
- Q1 (complexity) column: We check to see if the values are in the range between one to five using regex. If any null values exist in the column, we replaced them with the median of the entire column
- Q2 (ingredient count) column: Many of the responses in this column did not contain numbers but a list of ingredients instead or a combination of both. We converted words that contained numbers into digits and we removed any vague or confusing words and replaced it with the median of the entire column.
- Q3 (setting) column: Instead of having all of the responses in one column, we divided each response into different columns with binary data instead. So we have columns for weekday lunch, weekday dinner, weekend lunch, weekend dinner, party, and late night snack where each column has 0s or 1s.
- Q4 (cost) column: This column also contains many inconsistencies such as different symbols, including range instead of a set price and having rough estimates. We removed any words and symbols and if the response was a range we replaced it with the average instead.
- Q5 (movie) column: We decided to convert this column into a bag of words. In the bag of words, we include the most common 350 words. We experimented with many different numbers and 350 words was the best result we received. We believed it was important to retain as much information as possible because there were some correlations between the movie and food. For example, many people who have chosen Avengers would associate it with Shawarma.
- Q6 (drink) column: As there were different responses for the same type of drinks, we decided to group drinks together. So we have columns that grouped drinks that are

soda, juice, energy drinks, water, tea, coffee, and milk. Each new column has a boolean value.

- Q7 (people) column: Similar to the Q3 column, we created new columns that are parents, siblings, friends, teachers, and strangers where they have a boolean value.
- Q8 (hot sauce) column: Depending on the response, we would map it to the specific number. So our hot sauce column is still one column while having values from one to five.

Data Splitting

For Data Splitting, we started by splitting the dataset using an 80/20 stratified split, to allocate 80% of the data for training and validation, and saving 20% of the data for final testing. We chose this split based on the size of the dataset, knowing that we need enough data for training while reserving a relatively large population for testing to evaluate the model's performance properly. To make sure that each subset has the original class distribution, we used stratified sampling using sklearn's `train_test_split()` function with `stratify=y` parameter. Then within the training + validation portion, we applied 5-fold stratified cross-validation using `StratifiedKFold`. This approach helped us preserve label proportions across all the folds and get a more reliable estimate of model performance during hyperparameter tuning. Each configuration was trained and validated across all folds, and then averaged cross-validation accuracy to guide hyperparameter selection. This process of data splitting ensured fair evaluation with balanced classes and minimized overfitting to a specific subset or split.

Model

In order to determine the most suitable model for this task, we evaluated multiple established algorithms: Logistic Regression, Random Forests, Neural Networks and Naive Bayes. The choices of these model families were based on the task requiring models that perform classification and are not overly complex. These models were implemented using Scikit-learn and were assessed by comparing evaluation metrics, however primarily focusing on accuracy since the task does not require specific attention to other metrics, specially for the general choice of the model. However, we did make sure to choose the model that does not overfit and performs relatively consistently across different classes.

For Logistic Regression, after dividing the numerical and categorical features, we have split the train with valid size and test size to 80% and 20%. Then split the train and valid size to 80% and

20%. After normalizing the data and using Sklearn to train the data, we received 95% training accuracy and 82% test accuracy.

With Neural Networks, we received 93% training accuracy and 87% test accuracy, but it introduced additional complexity and was prone to overfitting. Our neural network was built using Scikit-learn's MLPClassifier with a single hidden layer of 40 neurons, L2 regularization ($\alpha=0.0001$), and an initial learning rate of 0.001. Early stopping is enabled with 10% of the training data held out for validation, and the model is allowed up to 2000 iterations with a fixed random state for reproducibility. This configuration helped the network achieve about 93% training accuracy and 87% test accuracy, capturing complex nonlinear relationships, but it also introduced additional complexity and required careful hyperparameter tuning, making it less attractive compared to simpler models like Random Forests.

For Naive Bayes, we fine tuned our hyperparameters via a 5 fold cross validation using 23 candidate values of alpha. For each value of alpha we use GridsearchCV to train the model on 4 folds and validate it on the 5th. We ended up getting 87% training accuracy and 85% test accuracy for $\alpha = 5$. Naives Bayes relies too much on the assumption of independence among features, this is a problem in our case since we have overlapping features. We also tried using decision trees such as XGboost which would get an accuracy of 86%

Random Forest, however, consistently had a better trade-off between capturing details and generalizing predictions. Since Random forest does not introduce as much complexity as some other models, requires minimal preprocessing, and could achieve an 87% percent test accuracy initially, we decided to choose it as our final model and work on tuning the hyperparameters. We did also explore simple Decision Trees but with a low accuracy of 83%, it did not outperform Random Forest.

Model Choice and Hyperparameters

Model Selection:

We considered many models that we could realistically implement by only using NumPy and Pandas. We decided to implement random forests. Out of all the models we have tested using Sklearn, random forest has the highest accuracy. Also, models like neural networks and naives bayes would be difficult to implement so we ultimately decided on random forests.

Evaluation Metrics:

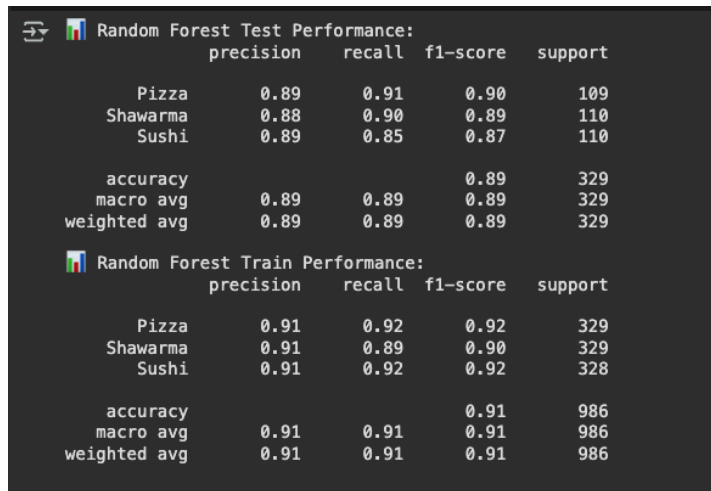
In order to evaluate our model, we calculated the accuracy, precision, recall and F1-score. We first split the data into 80/20 training/validation and test sets. We then ran our model on the different subsets to test the stability across various data samples. The accuracy reflects on the overall percentage of correct predictions across all of our classes, we also calculated the precision (proportion of correct predictions/ all the predictions made for a class) these help us know the model's performance. We also save the recall as it tells us how many actual instances we successfully identified, this helps us reduce cases of false negatives. Finally we calculate the F1-score in order to get an overall score of both false negatives and false positives, since we want our model to treat all categories fairly the F1-score is crucial.

Reporting these various metrics across the data samples helped us check our model's overall performance.

Hyperparameter Training:

To further optimize the performance of our model (Random Forest Classifier) we performed grid search using 5-fold stratified cross-validation as explained earlier in the Data Splitting Section. We did this manually by iterating through different combinations of hyper parameters such as `n_estimators` (number of trees), `max_depth` (max tree depth), `min_samples_split` (min number of samples required to split a node). The search space included values for `n_estimators` = [250, 300, 350], `max_depth` = [7, 8, 9], `min_samples_split` = [10, 15]. We fixed `min_samples_leaf` = 1 based on earlier experiments which resulted in better performance overall. The search space was chosen to minimize overfitting, while allowing the model to capture enough details for accurate predictions. The application of k-fold cross-validation also helped prevent overfitting to one single split and provided a better metric for evaluating the best hyper parameters. We chose the final best parameters as `{'n_estimators': 250, 'max_depth': 7, 'min_samples_split': 10, 'min_samples_leaf': 1, 'max_features='sqrt'}` which had a high average cross validation accuracy. Even though, another combination `{'n_estimators': 300, 'max_depth': 9, 'min_samples_split': 15, 'min_samples_leaf': 1}` had a higher CV accuracy, we decided to go with the first combination since, the model was overfitting with a higher `max_depth` with a test accuracy of 88% and train accuracy of 92%. The first combination, however, resulted in a better balance with test accuracy of 89% and train accuracy of 91% which shows better generalization. Even though CV average is a reliable estimate of performance the difference in CV average was very low and we decided to choose the parameters that show better test results. During this process,

we ensured that while we maintain a high accuracy across all predictions, we also maintain a consistent performance between different classes by looking at the classification reports. Below is a screenshot of the classification report of the improved model on the test and train dataset.



The image shows a screenshot of a Jupyter Notebook interface with two tables of classification reports. The first table is titled 'Random Forest Test Performance:' and the second is 'Random Forest Train Performance:'. Both tables have columns for precision, recall, f1-score, and support. The test report shows performance for Pizza, Shawarma, and Sushi classes, with overall accuracy, macro avg, and weighted avg all at 0.89. The train report shows performance for the same three classes, with overall accuracy, macro avg, and weighted avg all at 0.91.

Random Forest Test Performance:				
	precision	recall	f1-score	support
Pizza	0.89	0.91	0.90	109
Shawarma	0.88	0.90	0.89	110
Sushi	0.89	0.85	0.87	110
accuracy			0.89	329
macro avg	0.89	0.89	0.89	329
weighted avg	0.89	0.89	0.89	329

Random Forest Train Performance:				
	precision	recall	f1-score	support
Pizza	0.91	0.92	0.92	329
Shawarma	0.91	0.89	0.90	329
Sushi	0.91	0.92	0.92	328
accuracy			0.91	986
macro avg	0.91	0.91	0.91	986
weighted avg	0.91	0.91	0.91	986

Final Model:

The final model in `pred.py` is a manual implementation of Random Forest using only libraries like `numpy` and `pandas`. This model is composed of 250 decision trees. After training the model using `Scikit-learn` with the best hyperparameter choices found via cross-validation, we extracted the resulting trees' internal structures (`feature_index`, `threshold`, `children` and `class distribution`) and saved them to a `.npy` file. This was done in a Google Colab Notebook `DT-ML.ipynb`. Then, in `pred.py` we implemented a custom prediction function from scratch without relying on `scikit-learn`. During prediction, `pred.py` loads these trees and reconstructs the forest. Each tree independently predicts a label for the given input by traversing through nodes recursively based on feature thresholds. The final prediction is done by majority vote across all trees which ensures consistency across different predictions and prevents high variance predictions from a single tree. Overall, this implementation satisfies the project's constraints and provides reasonable predictions given the input data without reliance on external libraries.

Prediction

We believe that our model will achieve 89% accuracy, the same as the original dataset. In our prediction script, we have taken into consideration different cases in the dataset by cleaning each column and using different methods like bag of words and regex matching as well as handling missing values silently. This performance estimate of the model was calculated based

on robust evaluations during our training and testing process. We held out a test set to evaluate the model's performance on unseen data. Using 5-fold cross-validation, majority voting in random forest, minimal overfitting (low difference between train and test accuracy) ensure a consistent behavior from the model on unseen data and indicates strong generalization capability. Additionally, our training set was diverse and had a balanced class distribution so we expect the model to treat each class fairly and behave as we expect on the test dataset.

Workload Distribution

Wonjae Lee: Tested logistic regression, helped with data correlation, tested final model, and wrote feature selection/data representation, model and prediction portions of the report while actively participating in group meetings.

Bora Celebi: Tested neural networks, implemented cleaning, helped with data correlation and prediction, tested final model, and wrote data exploration portions of the report while actively participating in group meetings.

Ali: Tested decision trees and random forest, implemented cleaning and prediction, tested final model, and wrote model evaluation and hyperparameters portion of the report while actively coordinating and participating in group meetings.

Najim: Tested Naive Bayes and Decision tree, worked on the final model and prediction, tested final model, wrote report (model and evaluation metrics), participated in meetings.