Student names:
Xin Lei Lin (1009988523),
Kai Lo (1009965940),
Robert Li (1009802331),
Ihor Chovpan (1009765238)

Data Cleaning

The first step towards parsing our dataset involved splitting it into training, validation, and testing subsets. This split was consistent across all the models we tested to ensure fair comparison.

We then attempted to apply various cleaning algorithms to the feature values for all the data. While most of the questions were open-ended, many contained responses that could be cleaned and transformed into categorical or quantitative formats. Our philosophy for each feature's cleaning approach was determined based on its anticipated impact on the analysis and on how effectively its information could be retained once converted to categorical or quantitative data.

We split the data set by question: for most of our models, each question will directly correlate with at least one feature. Some questions may be split into multiple features (as explained below), and some may be left out entirely intentionally for reasons we will detail. The focus of this section is on converting open-ended answers, containing many unusual responses, into clean, categorical data that our models can process. This process also involves many opportunities for improving the effectiveness of our models when deciding how to process the data.

Numerical Cleaning

For Questions 2 and 4, participants were asked to estimate both the number of ingredients and the expected cost of the food item. In converting these open-ended responses into numerical values, several challenges arose:

- What if the user inputs a range (or several numbers), such as "5-10 ingredients"
- What if the user adds additional information next to their input, such as "5$CAD"
- What if the user writes out their answer as a word, like "five ingredients"

Our numerical cleaning procedure employs the following steps:

1. Attempts to parse the input as a date, and returns the mean of the months/days (this is necessary in cases where the data parser mistakes ranges like 4-5 for a date)
2. Removes all dollar signs and extra whitespace
3. Parses any ranges separated with dashes or slashes as individual numbers
4. Parses both numeric and written-out numbers (e.g., 5, five, sixty-seven)

5. Returns the average of all identified numbers
6. If no numbers were identified, find all comma-separated items in the response, and return the number of elements in this list (this is assumed to be a list of ingredients).

This approach provides a more robust method for representing the numerical data, ensuring all unusual input formats are handled systematically.

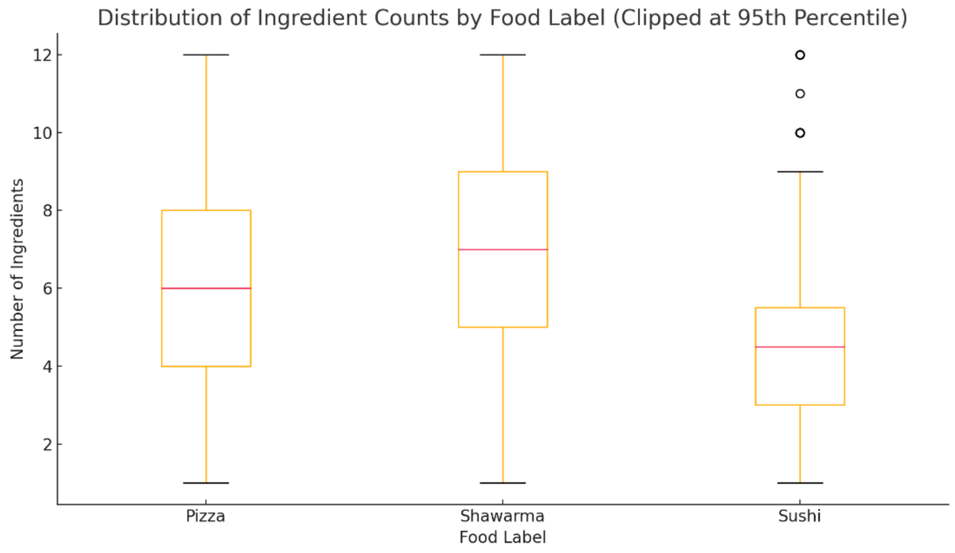

Figure 1: Boxplots of the correlation between # of ingredients and food label

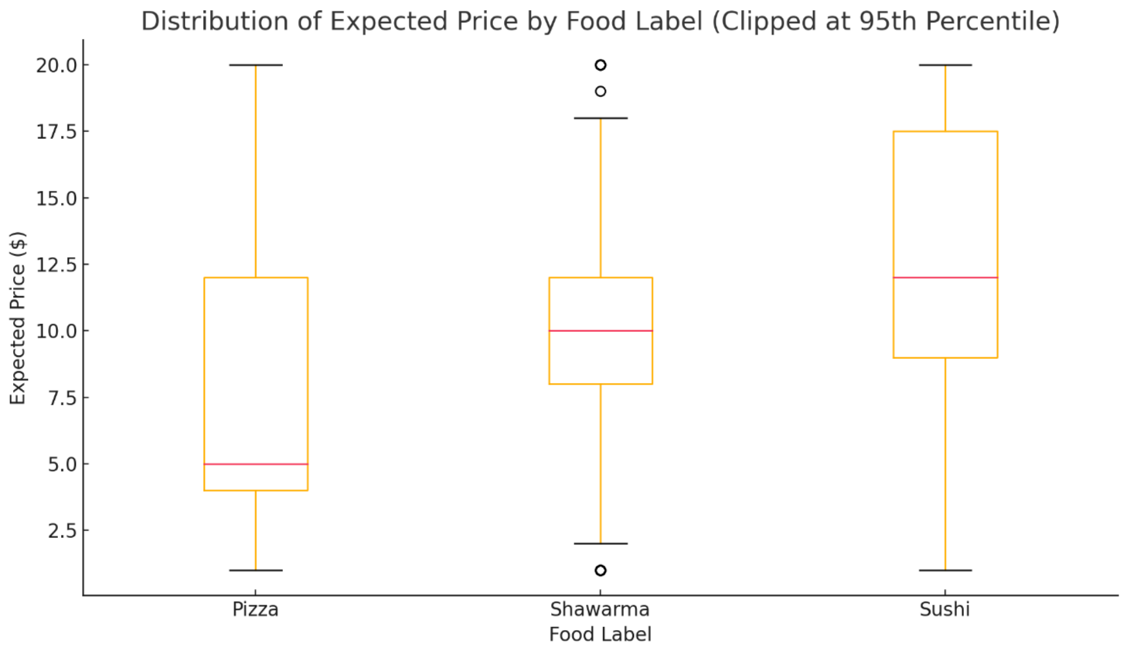

Figure 2: Boxplots of the correlation between expected cost and food label

In Figures 1 and 2, we can see how the distributions of cleaned answers to questions 2 and 4 are correlated with the food label. As predicted, these appear to at least be decently correlated, and this processing will be useful in our models.

The script for performing this cleaning can be found in `clean/number_cleaning.py`.

One-hot and Multi-hot Cleaning

Several questions in the dataset allowed for multiple selections per response:

- Question 3 asked respondents to indicate the settings in which they would typically eat the food, offering choices such as “Week day lunch,” “Week day dinner,” “Weekend lunch,” “Weekend dinner,” “At a party,” and “Late night snack.”
- Question 7 asked which people the food reminded them of, with options including “Friends,” “Teachers,” “Siblings,” “Parents,” “None,” and “Strangers.”

Additionally, Question 8 was a single-select categorical question asking how much hot sauce the respondent would add to the food, with options like “None,” “A little,” “A moderate amount,” “A lot,” and “I will have some of this food with my hot sauce.”

To ensure this data could be effectively used in machine learning models, we transformed the multi-select responses into a set of binary features, with one for each possible choice. This multi-hot encoding was applied to Questions 3 and 7, resulting in six separate features per question. For Question 8, a one-hot encoding was applied to represent the categorical value as a single vector.

This strategy ensures that each category is treated independently by the model, avoiding any unintended ordinal relationships between choices and preserving the full expressiveness of the original responses.

The script for performing this cleaning can be found in `clean/onehot_cleaning.py`.

Drink Cleaning

Question 6 played a particularly important role in shaping our data cleaning strategy, since we observed a strong correlation between the drink a respondent would pair with the food and the food label itself.

In our initial analysis, it became clear that certain drinks were highly indicative of specific food items. For example, responses mentioning tea, green tea, or traditional Japanese beverages showed a strong association with sushi. Similarly, entries referencing ayran, laban, or other yogurt-based/Middle Eastern drinks were frequently paired with shawarma. Soda or Coke had a weaker, but still noticeable, correlation with pizza.

Given this correlation, we made the decision to convert the open-ended responses in Question 6 into a strictly categorical feature. This required standardizing and categorizing a wide variety of textual inputs into a manageable and meaningful set of drink categories. To accomplish this, we created a lookup file called `clean/common_drinks.simple`, which maps similar drink terms (including common misspellings or variations) to a unified category.
An excerpt from this file is shown:

```
coke=cola,coke,coca cola,cola,coca-cola,cocacola,cococola,coke-cola,coke cola
sparkling=sparkling,soda water,soda-water,tonic,tonic-water,club-soda,club
tea=green tea,greentea,green-
tea,tea,matcha,ocha,manzanilla,chamomile,peppermint
juice=juice,apple,cider,orange,grape,cranberry,canberry,fruit,pineapple,pomeg
ranate,pomegranite,coconut,aguas frescas,aguas-frescas,fruit punch,fruit-
punch,hawaiian punch,sherry,xeres,schnapps,schnaps,lemonade
soda=soda,pop,soft,pops,sodapop,soda-pop
```

Each line groups semantically or culturally similar drinks into a common label. This both simplifies the model's input space and captures valuable cultural context.

Our drink-cleaning algorithm processes each answer through the following steps:

1. The input is stripped of excess whitespace, punctuation, and symbols, then lowercased for normalization.
2. It checks whether any token (from the right-hand side of the `common_drinks.simple` mapping) appears directly in the string. The first match is mapped to its associated category.
3. If no direct match is found, the string is split into words and searched again token by token.
4. If there is still no match, we compute a Jaccard similarity score between the input and each known token, returning the best match above a certain threshold (which is a tuned parameter).
5. If all attempts fail, the drink is assigned to a default "No drink" category.

There are several key benefits to this approach:

- By grouping culturally similar drinks, we increase the chance of the model learning associations based on cultural context or common consumption patterns.

- This data cleaning erases some of the specificity of the feature data (like when green tea is categorized as "tea", or aguas frescas as "juice"). This can actually prevent overfitting for the drink data and help us recognize the underlying patterns.
- The ordered structure of the `common_drinks.simple` file lets us prioritize specific categories while reserving more general ones as fallbacks.
- The inclusion of Jaccard similarity ensures greater resilience to misspellings or unusual formatting, which was common in our dataset.

This ensures that the drink data is fully standardized into a discrete set of categories, making it directly usable by any classification model. It also enhances the feature's predictive power by preserving cultural and semantic signals from the raw input.

One concern with this approach was the potential for overfitting, since the drink dictionary was initially derived from training data. To mitigate this, we manually supplemented the file with entries for hundreds of the most referenced drinks found online. While some overfitting may remain, the tradeoff is acceptable in return for converting an open-text field into a powerful categorical feature.

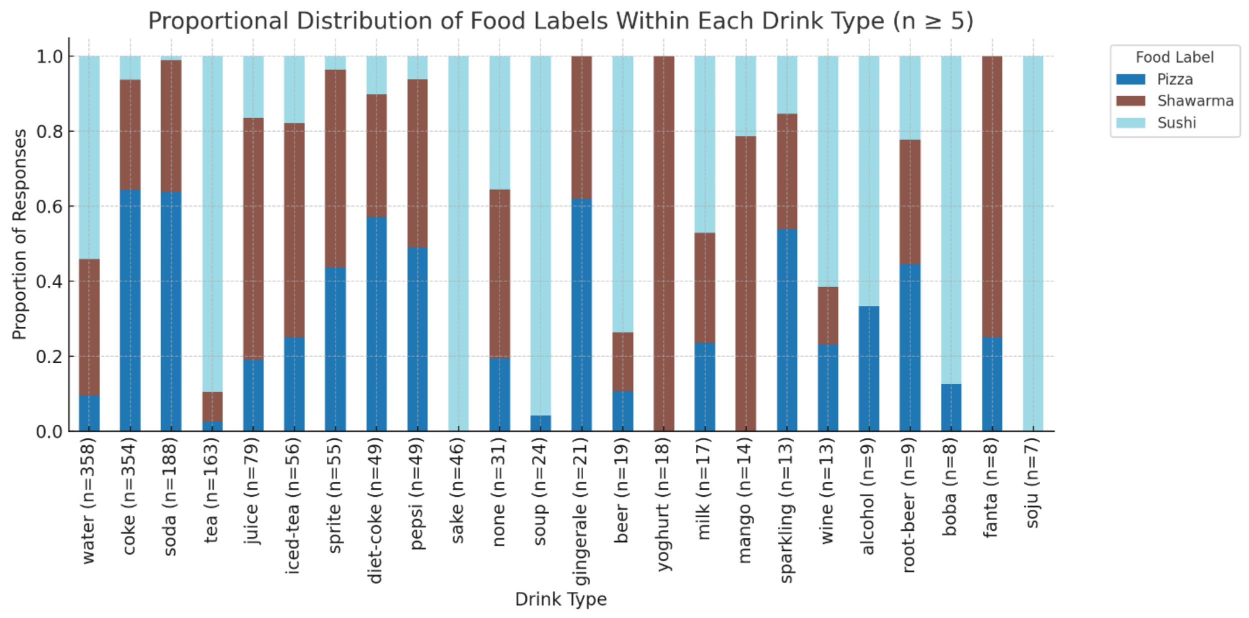

Figure 3: Splits between target labels for each identified drink type

In Figure 3, we can see how many of the commonly identified drink types (the bars furthest to the left) are quite strongly correlated with their truth labels (water/tea with sushi, coke/soda with pizza, iced tea with shawarma, etc.). This confirms our original assumption that drinks could be a strong indicator of the food label, so this drink cleaning should prove to be quite helpful.

The script for performing this cleaning can be found in `clean/drink_cleaning.py`.

No Cleaning

Question 1 asked the respondents to rate the complexity of the food from 1 to 5. Since this data was already purely categorical, no cleaning was required.

Question 7 asked the respondents which movies they would associate with the target label. While this initially seemed like a similar case to the previous question regarding drink pairings, a preliminary examination and classification of the data proved that there were far too many movies to classify, and weak correlations between the types of movies and the truth labels (the script written to prove this can be found in our datasets/clean folder). Therefore, we didn’t apply any cleaning to question 7. Some of our models simply omitted data from Question 7 to prevent overfitting, and some used its data with little-to-no cleaning despite the risk of overfitting.

(It should be noted that there was one exception to the minimal correlation in Question 7 that we saw: responses including “The Avengers” as their movie pairing correlated extremely well with shawarma as the target label. However, we opted to not include this information as we were concerned it may lead to overfitting the training set.)

Predicted Impact of Our Cleaning

Our overall cleaning strategy aimed to balance expressiveness with generalizability. By standardizing numerical inputs, one-hot encoding categorical and multi-select responses, and intelligently categorizing open-ended fields like drink pairings, we enabled our models to work with a structured, noise-reduced dataset while preserving as much original information as possible.

We expect these transformations to have a significant positive impact on model performance, particularly in improving accuracy and consistency across model types. Features like drink pairings and serving context, once cleaned, likely carry strong predictive signals that would have been lost or diluted in their raw form. Conversely, we intentionally left some features (like movie associations) mostly untouched or excluded when their impact was minimal or when cleaning risked introducing overfitting.

CSC311H1
Final Report Food Classifier

First model: Naive Bayes

Motivation:

We first noticed that a lot of the columns are text-based. This made it initially harder to use models that are based on numerical values, such as neural networks and decision trees. As a first baseline, we therefore opted for a naive bayes approach, which, through the bag-of-words tokenization is capable of modelling vocabulary to establish predictions.

Naïve Bayes Original Data Processing:

As a simple baseline for Naïve bayes, we considered the entirety of the data (including numerical values) as strings. We then concatenated all the answers from each row into each datapoint and created our bag-of-words dictionary as such. We have also experimented with only using subsets of questions that contain textual data, instead of numerical values.

Baseline model:

Relevant files: [~/models/naive_bayes.py, ~/ train_naive_bayes_sklearn.py]

We initially trained a scikit-learn implementation of naive bayes. Scikit-learn was invaluable in this case for quick development, as a proof-of-concept, with reasonably tuned accuracy. We chose the MultinomialNB() model as a baseline, and employed the default vectorizer, CountVectorizer(). As shown in Table 1, this was our baseline model.

Numpy-only Training and Prediction Script:

Relevant files: [~/train_naive_bayes_np_only.py, ~/models/np_naive_bayes_utils.py, ~/inference_naive_bayes.py]

Following the promising results from the baseline model, we proceeded by writing the inference script for the training and the inference of the Naïve Bayes using only Numpy. We then proceeded to create a common vocabulary of all possible words that we could find in the testing script, and saved this vocabulary, as well as the conditional probabilities for each word, within Numpy files. We can notice that in the *inference_naive_bayes.py* script, we are now capable of loading the Naïve Baye’s learned priors and conditional probabilities for each word in the common vocabulary.

Hyperparameter Tuning:

Relevant files: [~/train_naive_bayes_np_only.py, ~/models/np_naive_bayes_utils.py]

Like what we learned in the labs, we implemented the MAP Naïve Bayes, after we noticed that MLE Naïve Bayes was outputting random values (with an accuracy of 0.33). We performed grid_search, for the parameters of a and b, present in the MAP computations of the conditional probabilities. We then determined that the best values for (a,b) = (2, 0), by running our grid_search across the validation set, as shown in **Figure 1**.

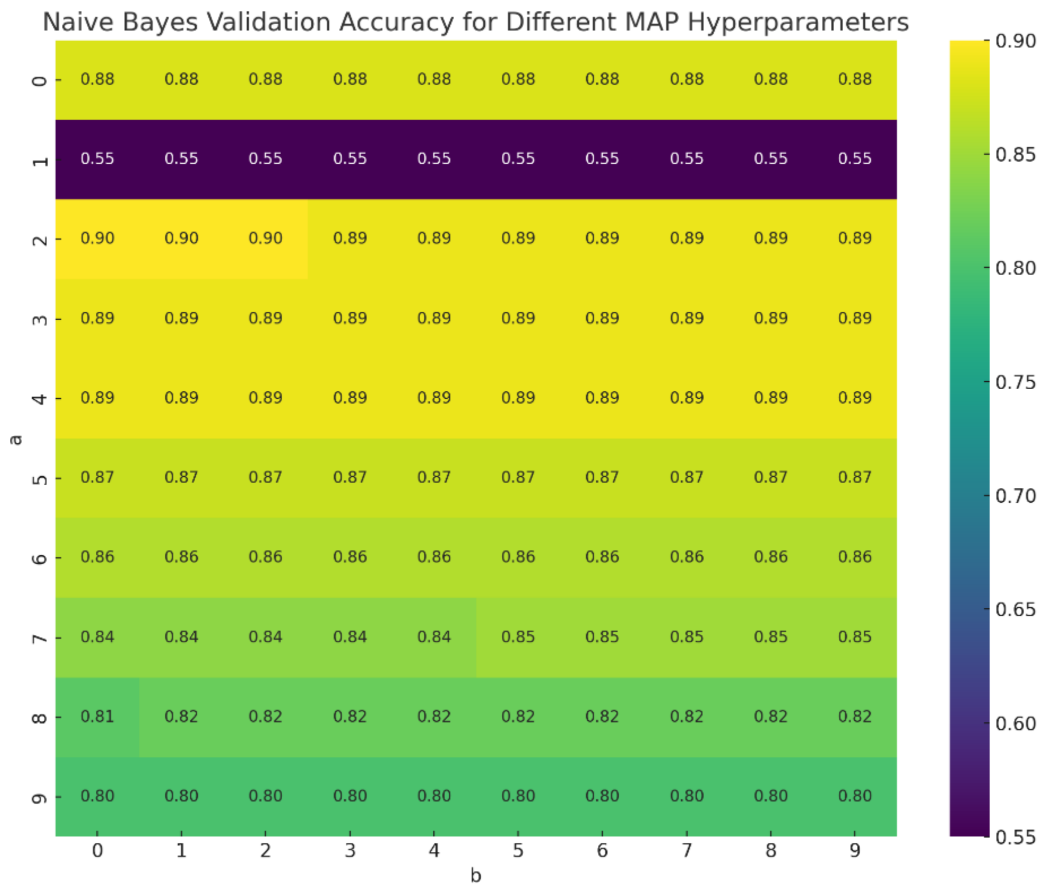

Figure 4: Naive Bayes Validation Accuracy Across Different MAP Hyperparameters (a,b)

Data Preprocessing:

Relevant files: [~/inference_naive_bayes.py]

Following the relative success of our baseline model, we decided to spend a bit more time working on the data preprocessing. First, we applied a crucial transformation, which was to remove the labels from the datapoints. (i.e. if a message wrote: “I love eating Pizza with friends” => “I love eating [LABEL] with friends”). Furthermore, we included the data preprocessing for the drinks, and the movies, as described above. This yielded an increase in validation accuracy, (see **Table 1**). This updated data processing was reflected in the *predict_smart* function in the *inference_naive_bayes.py* file.

Final Accuracies and Confusion Matrix:

Table 1: Different Naïve Bayes Models

Naïve Bayes Model	Training Accuracy	Validation Accuracy	Testing Accuracy
Scikit-learn Naïve Bayes (MultinomialNB())	93.652%	86.235 %	85.830%
Numpy Naïve Bayes	92.348%	89.244%	84.274 %
Numpy Naïve Bayes (+ data preprocessing)	91.826%	89.837%	86.290%

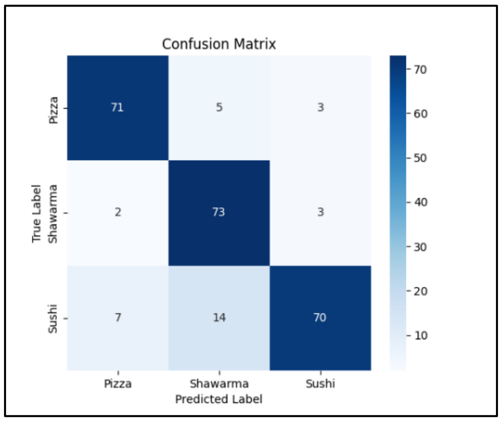

Figure **51:** Confusion Matrix of the Naive Bayes written in Numpy with data preprocessing

The model has very high testing accuracy but has difficulty between Sushi and Shawarma. By observing the confusion matrix, we can determine the presence (or absence of) recurrent mistakes to be able to either scrutinize our data more carefully, or to tune our models. We were therefore pleased to observe the desired highlights in the diagonal, indicating strong accuracy for the correct class.

Second Model: Decision-tree based classifiers

Data cleaning

Decision-tree-based models usually work on numerical data, comparing values for different features. Data cleaning techniques helped us encode the survey answers for textual answers (i.e for pairing drinks) and find the numerical answers for questions that ask for price or several ingredients. These data cleaning methods were introduced above, in the Data cleaning section, and therefore helped remove a lot of the inconstencies in the writing of “number of ingredients”, price of the food item, and other supposedly numerical values which were not properly written by some students. Without cleaning the data, the decision tree would be relying on a lexicographic string comparison, which doesn’t carry much meaning in the context of our dataset, reducing the potential accuracy of predictions.

Models explored

Relevant files: [~/models/tree.py]

We explored **decision tree, random forest and gradient boosting classifiers** because these models can efficiently consider both categorical and numerical information from our dataset, assigning larger priority to some questions compared to others.

For each classifier, we used grid search on 'max_depth', 'min_samples_split' and 'criterion' parameters, compared validation accuracies and chose the best hyperparameters. The following is a table of best hyperparameters for each model and corresponding validation accuracy.

Table 2: Best validation accuracies for decision tree models

Classifier	Criterion	max_depth	min_samples_split	Validation accuracy
Decision tree	gini	10	16	0.825
Random Forest	log_loss	30	2	0.865
Gradient Boosting	squared_error	25	128	0.869

Additional Model Combination Explored: Naïve Bayes + Decision Tree

We also tried chaining Decision Trees with a Naive Bayes classifier, feeding the class logits obtained from the outputs of the Naïve bayes classifier, into the decision tree as three additional parameters. The outputs of the Naïve bayes could therefore be thought of as a latent representation of the same data that is being fed in the decision trees. Since decision trees perform well on numerical data, and naïve bayes on textual data, we believed such a combination could improve the validation accuracy.

Table 3: Best validation accuracies for decision tree models combined with Naive Bayes classifier

Classifier	Criterion	max_depth	min_samples_split	Validation accuracy
Naive Bayes + Decision tree	gini	100	2	0.821
Naive Bayes + Random Forest	log_loss	15	16	0.873
Naive Bayes + Gradient Boosting	squared_error	5	2	0.882

Analyzing the results from the tables, the validation accuracy slightly improved when taking into consideration the Naive Bayes class predictions as extra features for the trees.

Tuning hyperparameters for Decision Trees, Random Forest & Gradient Boosting classifiers

Relevant files: [~/log/*.csv, ~/log/display.py, ~/models/tree.py]

For decision-tree based classifiers, we used grid search on several parameters: max_depth, min_samples_split and criterion, and picked the parameters with the best validation accuracy.

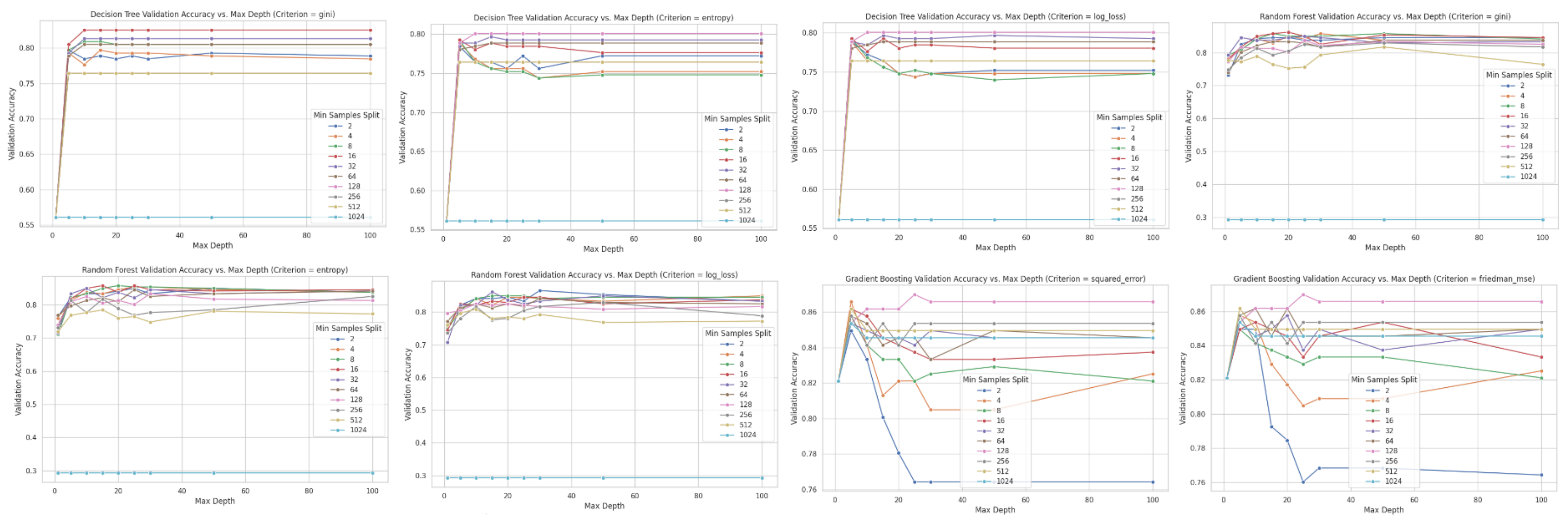

Figure 6: Validation accuracies for different models and criterion choices

Analyzing the graphs, we can see that for any combination of a classifier, criterion and min_samples_split, the optimal max_depth is less than 30 and is less than 10 for the “gradient boost + squared_error” combination. Shallow trees performed the best in every case of parameter optimization since they are usually less prone to overfitting and have smaller variance.

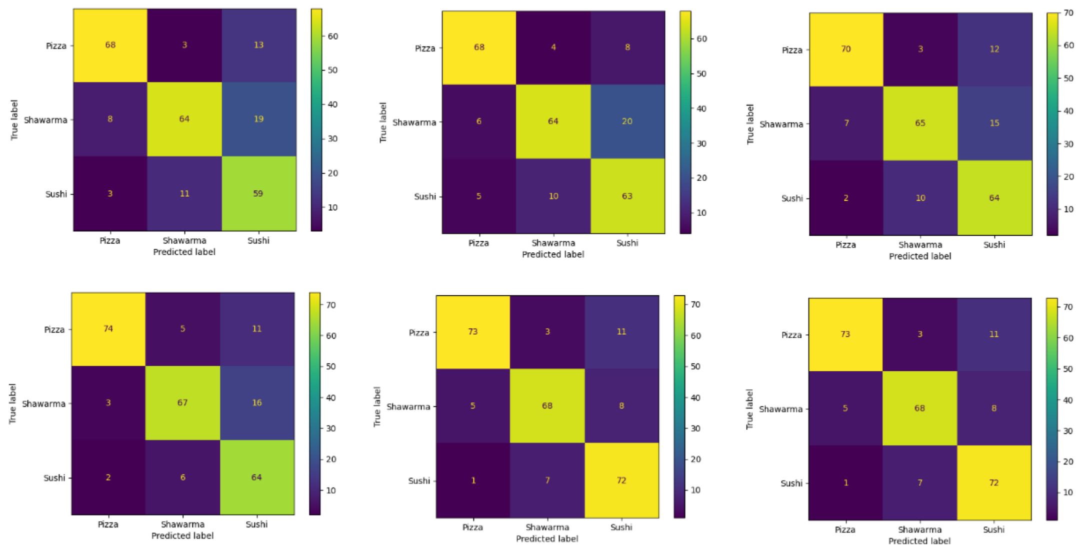

*Figure **7:** Confusion Matrices*

Upper row: Decision Tree, Random Forest, and Gradient Boost (without chaining)

Lower row: Decision Tree, Random Forest, and Gradient Boost (with chaining)

Observing our confusion matrices is a great way to distinguish certain features that are overly mistaken. This helped us detect any anomalies (notably the Avengers phenomenon above), and we were therefore able to adjust our preprocessing and models accordingly.

Implementation details

Relevant files/folders: [~/models/tree.py, ~/decision_tree_package, ~/gradient_boost_package (older commits)]

The training script is contained in 'models/tree.py' file. By selecting various flags on top of the file, one can train one the models: decision tree, random forest, gradient boosting as well as save decision tree and gradient boosting models.

After improving the data processing, we achieved 0.911 testing accuracy on the Naive Bayes + Gradient Boosting combination. Unfortunately, we couldn't import it to the prediction script due to the complexity of the Gradient Boosting implementation inside the scikit library.

Third Model: Neural Networks

Baseline Neural Network Trained from Scratch:

Relevant files: [~/neural_network_package]

Neural Networks yield similar advantages to Trees, in that they are capable of learning the features representation of numerical inputs. We therefore wanted to utilize this capacity, in addition to our data preprocessing, which turned columns such as price, and drinks, into numerical features. The data preprocessing pipeline was therefore exactly the same as what was described in the Data Cleaning section. We have also made sure to normalize the input variables between [0, 1] to make sure that the neural network could learn these representations.

For our baseline Neural Network, our general model architecture includes 2 hidden layers (heavily inspired from our implementation in the labs of a neural network from scratch). Note that the forward pass, backward pass, and the entirety of the implementation was written in Numpy. Additionally, we trained with regularization (L2), and weight clipping (to avoid weights that were larger than a certain threshold). By choosing to implement the neural network's backpropagation entirely using Numpy, we wanted to facilitate the transfer into a prediction file later on Markus.

Chained Neural Network with Naïve Bayes:

Relevant files:[~/ neural_network_chain_naive_bayes.py]

Similar to how we chained the Decision trees after the Naive Bayes classifier, we have attempted the same and observed marginal improvements. A few differences with the chaining that happened with decision trees is that we had to normalize the values (we simply took the ratio of the logit over the sum of all logits), and inputted those values into the neural network.

Hyperparameter Tuning:

Relevant files: [~/neural_network_grid_search_vis.py]

For neural_network based classifiers, we used grid search on several parameters: learning rate, batch size, l2 lambda, hidden sizes, patience and weight-clip threshold. Then, we picked the parameters with the best validation accuracy.

For weight-clip threshold, this is a method that we use to avoid overfitting for our neural network model. We restrict the weights of all neurons to be below the weight-clip threshold to avoid any feature taking too much weight due to potential data imbalance.

Table 4: Validation accuracy of different neural networks

Classifier	Training accuracy	Validation accuracy
Neural Network (without weight clipping)	0.8525	0.8480
Neural Network (with weight clipping)	0.9009	0.8455
Neural Network (with logits from naive bayes model)	0.9498	0.869

Following are the graphs of validation accuracies for different models and criterion choices.

For neural network only:

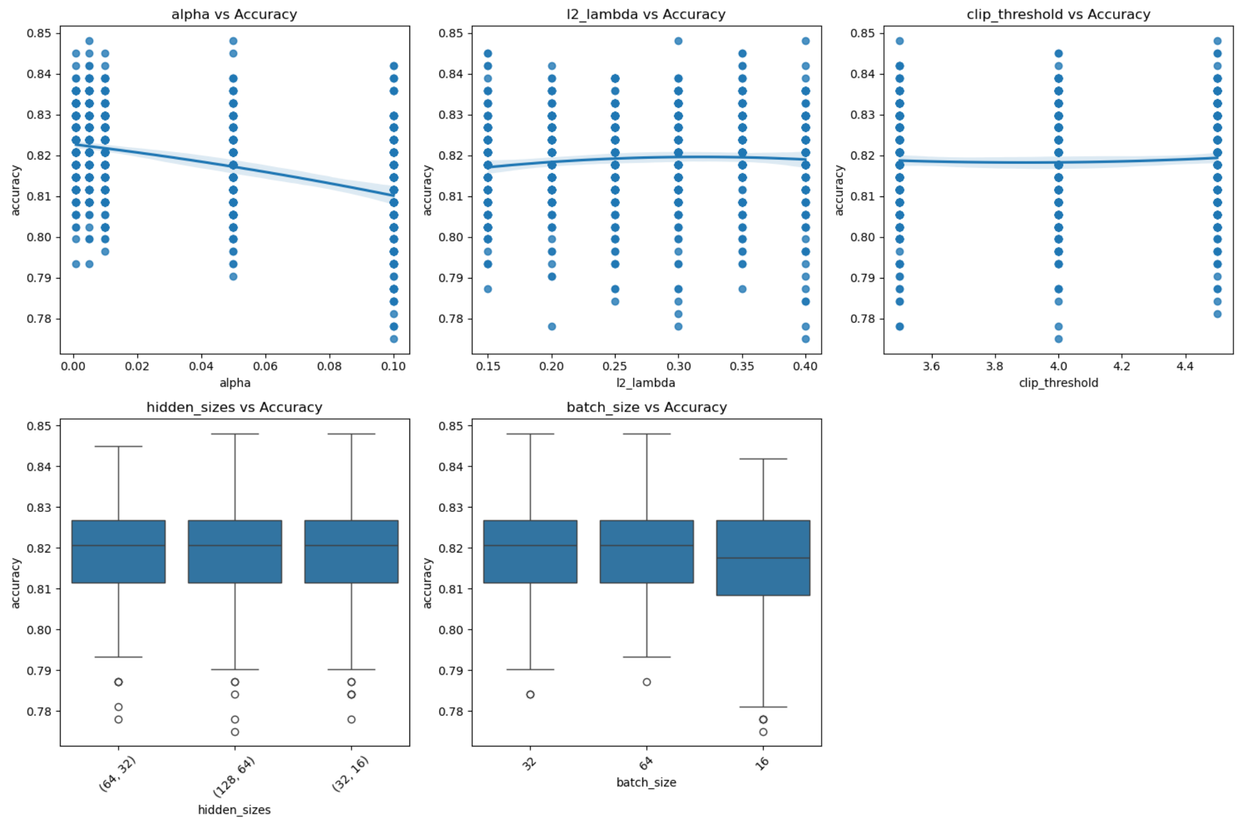

Figure 8: Relationship between accuracy and hyperparameters, where alpha is sampled from 0.001 to 0.1

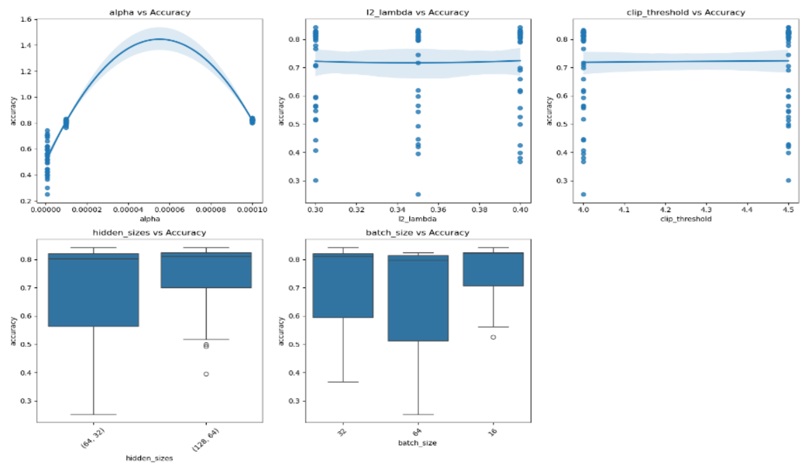

Figure 9: Relationship between accuracy and hyperparameters, where alpha is sampled from 0.00001 to 0.0001

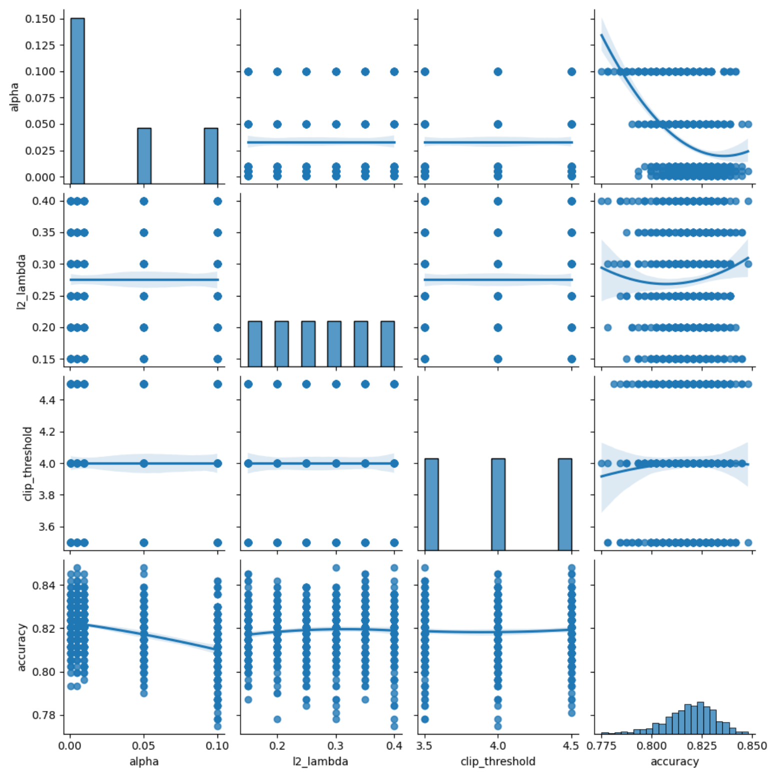

Figure 10: Relationship between hyperparameters and accuracy

From the plot we can see that l2_lamda and clip threshold does not have obvious impact on the accuracy within this range. However, different batch sizes have almost the same median but different variance in accuracies. The smaller the batch size tends to have smaller variance. Also, bigger hidden size has smaller variance as well.

Confusion Matrix on Validation and Testing:

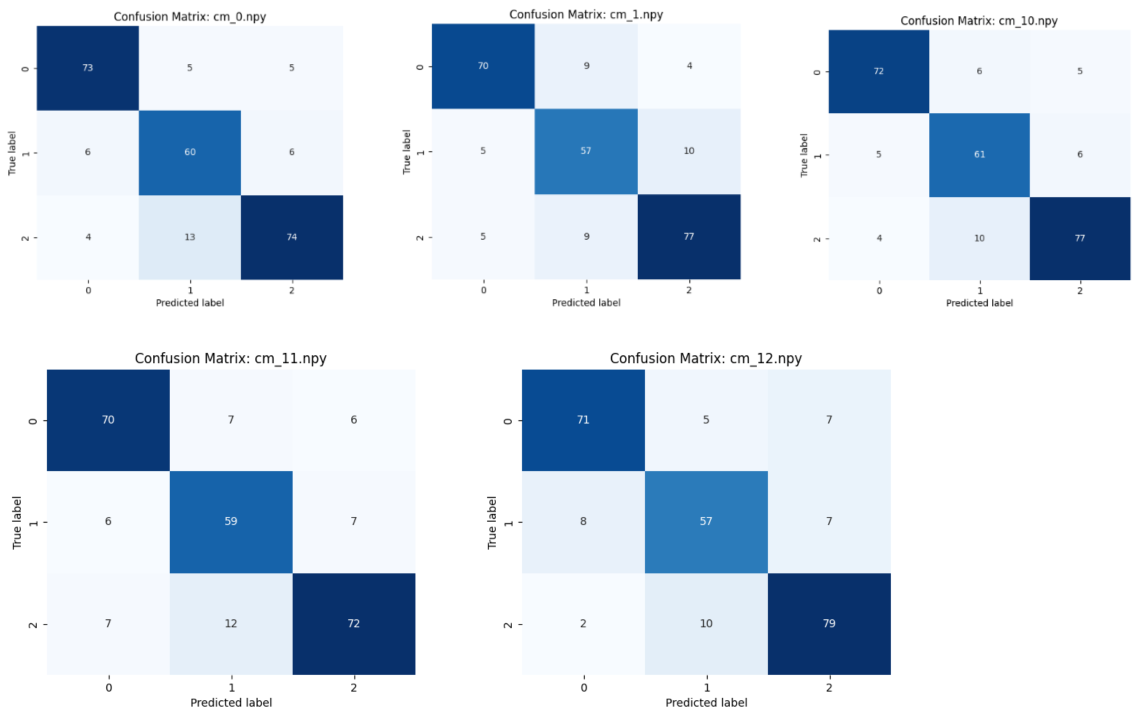

Figure 11: Confusion matrices from 5 models

Once again, observing the confusion matrices was a great way for us to notice any anomalies in the prediction script of our model (i.e. if our model was performing particularly poorly on a class or not distinguishing between two labels.

Implementation details and Running the Script

The training script is contained in *neural_network_chain_naive_bayes.py* file under folders “model”. Just simply run it and you can get the validation accuracy on the given train and validation dataset. To do a grid search for hyperparameters tuning, just run *neural_network grid_search_vis.py* in the *code.zip* folder.

Final Model Choice:

To choose the final model, we first split the data set into a training set, a validation set, and a testing set, in a consistent deterministic split which was preserved across the models. The proportions of the data split are 70:15:15. We compiled the following values for each model and determined that Naive Bayes has the best testing accuracy and is our final model. This comes despite our best efforts to make models which combine either decision trees and naive bayes, or neural networks, with naive bayes. Yet, these models, despite showing better training accuracy, consistently output lower validation and testing accuracies. Therefore, we have decided to opt for our simple Naive Bayes classifier, due to the better test accuracy.

The expected test accuracy for Naïve Bayes is **0.8629.**

Note: we know that it is counter intuitive that the simplest model is the best performing. Yet, through our extensive testing, and attempts at improving the Neural Network and the Decision Tree based models, we have concluded that Naive Bayes is indeed the most performant, in terms of models we were able to only implement in Numpy.

Table 51: Testing accuracies across different models

Model tested	Test Accuracy
Scikit-learn Naïve Bayes	0.8583
Naïve Bayes Numpy (+ data preprocessing)	**0.8629**
Decision Tree	0.770
Random Forest	0.790
Gradient Boosting	0.802
Naive Bayes + Decision Tree	0.766
Naive Bayes + Random Forest	0.830
Naive Bayes + Gradient Boosting	0.846
Naïve Bayes + Gradient Boosting (+ data processing)	0.911*
Neural Network	0.842
Naive Bayes + Neural Network	0.854

** Accuracy achieved through scikit-learn, could not import the model to the prediction script*

Workload Distribution

While this project was an intensely collaborative effort for our team, we did find it necessary to establish roles from the start to keep expectations clear for everyone, and to ensure that all the necessary work was completed.

- **Ihor:** I focused on researching the performance of decision-tree based classifiers: decision tree itself, random forest, gradient boosting classifier. The training included cleaning the data, using grid search on various hyperparameters, combining with Naive Bayes classifier. I also wrote the code that saves the decision tree and gradient boosting classifiers, as well as the prediction script for them.
- **Robert:** I focused on writing the neural network from scratch, using Numpy, and writing the different hyperparameter tuning elements. I made sure to add regularization, writing the whole backpropagation script for the neural network. I also added the implementation of the Neural Network which took the inputs from Xin Lei's Naive Bayes model as logits, into the neural network to obtain the final accuracy.
- **Kai:** I focused on exploring the dataset and its correlations, then coming up with a method of cleaning it to highlight meaningful patterns while reducing noise and setting up our models for success. The goal was to retain as much semantic information as possible while ensuring that the data could be effectively used across different models. I also coordinated between different group members on combining our models (since they involve feeding data from one into the other).
- **Xin Lei:** I focused on training the baseline Naïve Bayes scikit-learn. I also implemented the Naïve Bayes from scratch using Numpy, and saving the different learned parameters (conditional probabilities, etc). Finally, I suggested the idea of trying the chaining, where the logits outputted from the Naïve Bayes are used as a latent representation that can help inference of the models that are reliant on numerical inputs, such as neural network, and tree-based models. I also helped in the debugging of the Neural networks, adding weight clipping, and in the tree representation.