Machine Learning Challenge

Final Report

University of Toronto

CSC311: Introduction to Machine Learning

4 April 2025

Dhvani Patel: dhvanii.patel@mail.utoronto.ca

Mahak Mishra: mahak.mishra@mail.utoronto.ca

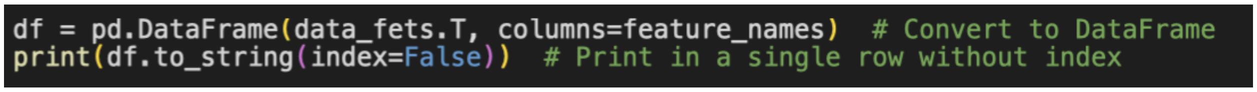
Hia Aggrawal: hia.aggrawal@mail.utoronto.ca

Table of Contents

Part I: Data

Data Exploration and Feature Selection

To start off, we performed a thorough exploration of the format and content of the survey dataset provided in `cleared_data_combined.csv`, which consists of multiple-choice and open-ended responses. We first loaded the data using `pandas,` and for ease, we renamed the verbose column names to descriptive identifiers (i.e. Q1 became `complexity`, Q2 became `num_ingredients`, etc.). Numerical features such as `complexity` and `price` were cast as integers, `num_ingredients` was preprocessed using the `get_num_ingredients()` function, and `hot_sauce` was converted to an integer scale (0 to 4). Categorical features (`setting movie`, `drink`, and `remind`) were converted to indicator features using `.str.contains()` or `.str.lower().str.contains()`. All transformed features were stacked into a NumPy array (`data_fets`) and then transposed and converted to a DataFrame for easy inspection using:



```
df = pd.DataFrame(data_fets.T, columns=feature_names)  # Convert to DataFrame
print(df.to_string(index=False))  # Print in a single row without index
```

Feature Engineering

We designed the following features directly from survey questions:

- From Q1: `complexity` : numerical, converted from text
- From Q2: `num_ingredients` : numerical, last integer in the string (to account for responses that listed items), or split response around “,” if no integers are found

- From Q3: Six indicator features were created (`s_wday_lunch`, `s_wday_dinner`, `s_wkend_lunch`, `s_wkend_dinner`, `s_party` and, `s_night_snack`). These are binary indicators based on whether the corresponding string was found in the text using `data["setting"].str.contains("...", na=False).astype(int).`
- From Q4: `price`: Extracted the first number using `.str.extract('(\d+)')`, filled missing values with 0, and converted to integers.
- From Q5: `movie`: Created ten indicator features corresponding to movies, which frequently appeared in responses using `.str.contains(...)` with relevant keywords per movie.
- From Q6: Seven binary indicators based on drink types: `d_water`, `d_soda`, `d_juice`, etc.
- From Q7: Five binary indicators were created (`r_parents, r_siblings, r_friends, r_teachers, r_strangers`) based on wether the corresponding string was in the response using `.str.contains("...")`
- From Q8: Transformed responses to integer using `hot_sauce_mapping` to create the feature `hot_sauce`

Feature Matrix and Dataset Construction

After feature engineering, we used `np.stack()` to compile all features into a NumPy array `data_fets`, with shape `(n_features, n_samples)`. We transposed this array to align samples along rows and created a `pandas.DataFrame`, named `df`, allowing us to inspect individual feature distributions.

Next, we made a list named `feature_names` to ensure the ordering and semantics of each column in `data_fets`, enabling traceability when attempting to interpret model outputs.

To pair each feature vector with its corresponding label, which is stored in the `"Label"` column of the original dataset, we made a list of tuples `D`, where each tuple `(x, t)` denotes the feature vector `x` and its label `t`.

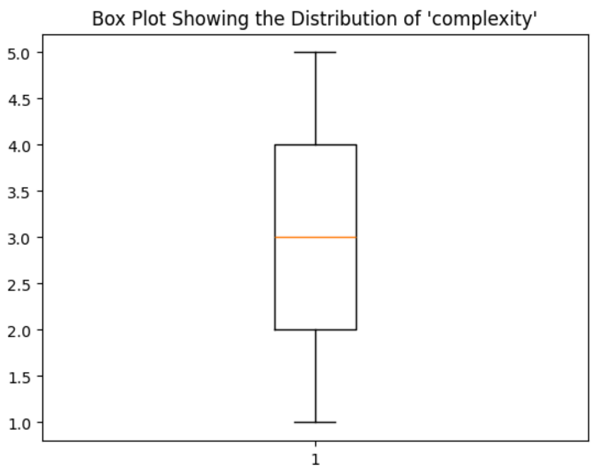
Training/Validation/Test Sets Split:

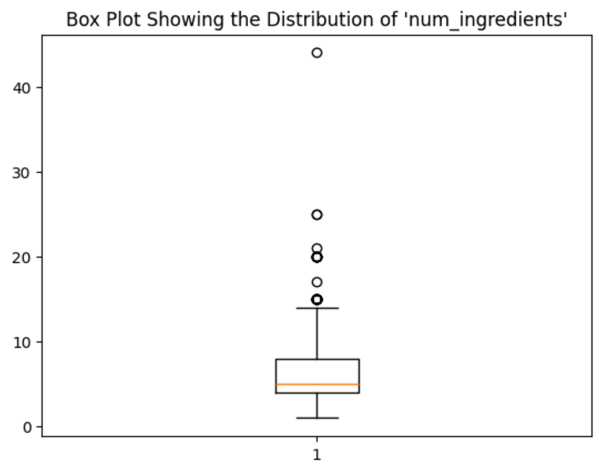
To start, we shuffled the dataset to ensure randomness and prevent bias. Next, we split the data into training, validation, and test sets. To ensure the best fit and accuracy, out of the 1644 total samples, we allocated 1100 data points to the training set to train the models properly, 275 samples to the validation set to tune hyperparameters and prevent overfitting, and 269 samples to the test set to ensure proper testing. We further extract the feature vectors and labels into the corresponding matrices: `X_train, t_train, X_valid, t_valid, X_test,` and `t_test`, stored as NumPy arrays.

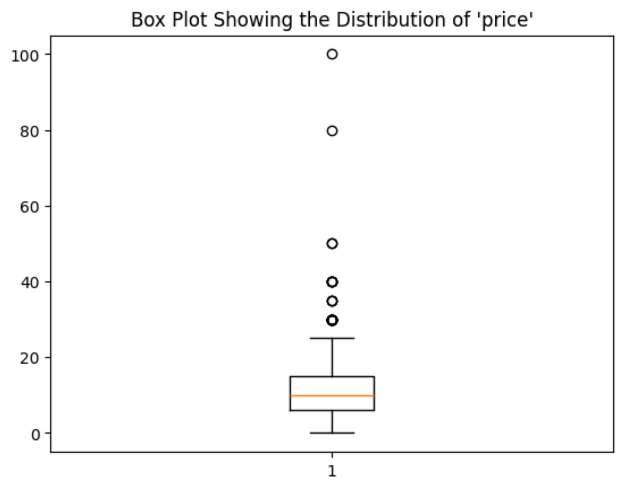
Exploratory Data Analysis:

Now, to better understand what models to produce, we carried out an analysis to better understand the structure of our dataset. To do this, we performed the following 3 analyses:

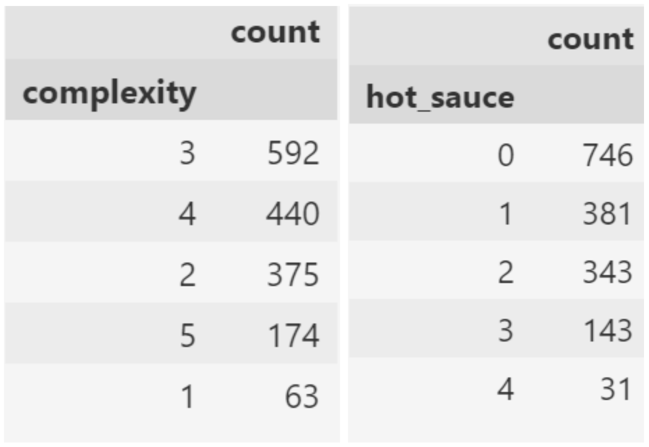
- Description Statistics: Using `df.describe()`, we computed mean, standard deviation, and percentiles for all numerical features.
- Box Plots: We also plotted distributions for `complexity`, `num_ingredients`, and `price` using `plt.boxplot(df[...])` to visualize spread and skewness, and detect any outliers. We have provided the box plots below:







- Distribution Counts: We used `df["complexity"].value_counts()` and `df["hot_sauce"].value_counts()` to assess the frequency of responses across these ordinal features.



complexity	count
3	592
4	440
2	375
5	174
1	63

hot_sauce	count
0	746
1	381
2	343
3	143
4	31

These analyses highlighted variations in participant responses, for example, some foods were consistently rated as simple or cheap, while others showed wide variability. Understanding this distribution is critical to interpreting model performance, especially in the presence of class imbalance or skewed features.

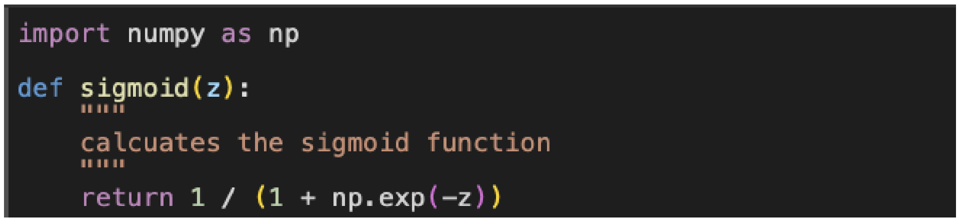
Part II: Model

The following section describes the models we are exploring, including logistic regression, k-nearest neighbour (KNN), neural networks, and decision trees.

Logistic Regression

Since the task requires classifying food items into discrete categories, we implemented logistic regression (rather than linear regression, which predicts continuous values). It models the probability that an input belongs to a particular class, which is important for food classification because sometimes there can be an overlap between categories (i.e. a food could be eaten for lunch or dinner). Since logistic regression gives a probability for each class, and the highest probability is selection, it naturally handles uncertainty well. Another reason why we chose it as one of our models is because logistic regression is fast to train and easily interpretable.

We used the sigmoid function to map a real-value input to a value between 0 and 1 (probabilities), which allowed us to interpret the likelihood of a food item belonging to a specific class:



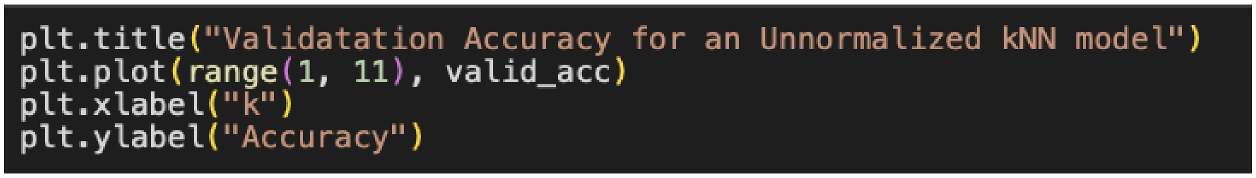
```python
import numpy as np

def sigmoid(z):
    """
    calcuates the sigmoid function
    """
    return 1 / (1 + np.exp(-z))
```

Then, to optimize the parameters (*w*, *b*), we implemented gradient descent, which iteratively adjusted *w* and *b* to minimize *L* (the cross-entropy loss function). The function `hyperparameter_tuning` uses a grid-search approach to iterate over all possible hyperparameter values and select the best model based on its validation accuracy. We chose grid-search rather than random search because our hyperparameter space was relatively small, allowing us to evaluate all possible combinations without too much computational cost. The training accuracy is ~83.00%, the highest validation accuracy is ~82.18%, and the test accuracy is ~85.13%.

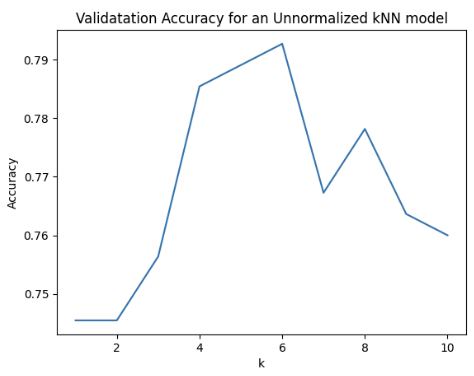
K-Nearest Neighbour (KNN)

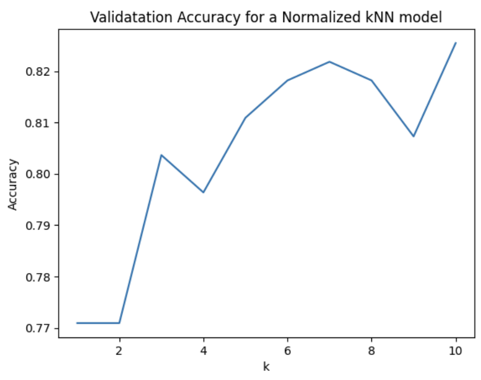
The K-Nearest Neighbours (KNN) algorithm is effective for multi-class classification problems like ours because it naturally handles multiple categories. The helper functions `dist_single` and `dist_all` are used to compute distance, and `predict_knn` is used to return a prediction. To select the best *k*, we performed a manual grid search over the range of possible values from 1 to 10:



```
plt.title("Validatation Accuracy for an Unnormalized kNN model")
plt.plot(range(1, 11), valid_acc)
plt.xlabel("k")
plt.ylabel("Accuracy")
```

Before applying KNN, we normalized the input features in the training, validation, and test sets to ensure that each feature equally contributes to the distance calculations (since KNN is a distance-based algorithm). After normalization, we re-ran the hyperparameter tuning, and plotted the validation accuracy for different *k* values:





The figures above show the calibration accuracy before and after normalization. The highest validation accuracy is ~82.5% and the test accuracy is ~82.2%

Neural Network

To classify food items into pizza, sushi, or shawarma, we also built a neural network model – specifically, a multi-layer perceptron (MLP) with one hidden layer of 64 hidden units. A number smaller than 64 such as 16 or 32 did not perform as nicely, and a large number such as 128 increased computation time and risked overfitting on our relatively small dataset. Thus, 64 was a stable choice and a middle ground between model capacity and overfitting.
The input layer took in 32 features from the survey responses obtained during the data collection phase. These features consolidate numerical values like complexity, price, and hot sauce level, with one-hot indicator features based on settings, movies, drinks, and associations. The model utilizes ReLU as the activation function in the hidden layer and softmax at the output to produce class probabilities.

In the training phase, we used cross-entropy loss and applied mini-batch stochastic gradient descent with batches of size 64. We chose a batch size of 64 because any smaller, such as 16, increased noise in the updates and led to less stable training, and larger one such as 128 or full batch slowed everything down, without providing meaningful gains. Hence, we realized 64 hits the sweet spot between speed and stability. The training was done with over 30 epochs with a learning rate of 0.1, since a smaller learning rate made the training very slow, and a higher one caused the model to diverge. Moreover, we used 30 epochs since any more epochs were starting to show diminishing returns, and validation loss didn't improve much after 30 epochs.

There was, however, a slight challenge we faced. Since the model is based on the randomness in parameter initialization and how mini-batches are formed during the training, we noticed that the performance of each model produced varied. Despite this, we achieved a high accuracy of 86% with a specific model that was produced, and the lowest observed was 77%. Overall, we saw that this model worked well for the classification task, as it handled mixed-type features effectively and was flexible enough to learn non-linear decision boundaries. However, due to its uncertain behaviours, we did not go ahead with this model and instead went forward with the logistic regression model.

Decision Tree

The decision tree model could be effective for a classification task such as predicting food types. The decision tree built from our training set provided results with an accuracy of 81% on the training set, 78.5% on the validation set, and 82.5% on the test set, indicating reasonable generalization. However, the issue with using this prediction model is that it is trained on a relatively small dataset for the highly complex feature set of 32 features, which may lead to overfitting. Furthermore, using a decision tree model with concrete splits on features may not be ideal given that the prediction will be tested on responses from a slightly different demograph (professors and teaching assistants). Professors, for instance, might find certain dishes less complex than students do. As a result, the model could perform poorly when applied to a different set of responses, as the learned splits might not align with the new dataset. Other models, such as logistic regression would be better suited as it models the relationship between features and the target variable more flexibly, without rigid splits, allowing it to generalize better across different demographic groups.

Part III: Model Choice and Hyperparameters

Model Selection

We chose multi-class logistic regression as our final model because our prediction task is classification-based and logistic regression is a well-established model for binary/multiclass classification problems. Given the relatively small number of features extracted and the moderate size of the dataset, it provides a good balance between bias and variance without needing extensive computational resources. Also, since we are not using libraries such as scikit-learn, logistic regression has straightforward matrix operations and can easily be trained using gradient descent. In addition, the logistic regression model had the highest test accuracy compared to the other 3 models on the same test set, with ~85.13% accuracy.

Furthermore, compared to neural networks, decision trees, and k-nearest-neighbours, there is a lower risk of overfitting. For example, models such as KNN are highly sensitive to training data noise, so it may overfit with lower *k* values (i.e. $k = 1$), and larger k values risk underfitting. Similarly, decision trees are prone to overfitting due its concrete splits, and neural networks can memorize training data if not properly regularized. For our logistic regression model, we mitigate overfitting through L2 regularization (adding a penalty term proportional to the squared magnitude of the model's weights to the overall loss function): `loss += (reg / (2 * num_samples)) * np.sum(weights**2)`.

Evaluation Metrics

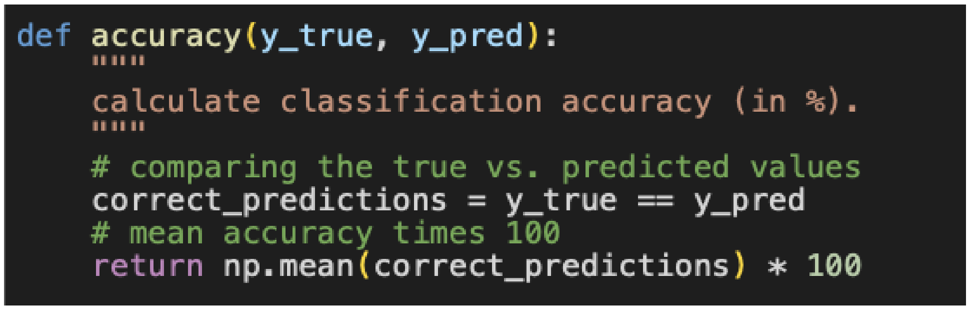
To ensure that evaluation metrics across different model and hyperparameter combinations were comparable, we maintained a consistent data split throughout the experiment:

- The dataset was split into training `(X_train, y_train)`, validation `(X_valid, y_valid)`, and test sets `(X_test, y_test)`.
- The test set was never used during training or hyperparameter tuning and was only used to evaluate the final model. That way, all accuracy values used for hyperparameter comparison are generated from models that were trained/evaluated on comparable, mutually exclusive data sets. It also made sure there was no data leakage between the training and evaluation phases.
- The validation set was used to perform hyperparameter selection using `hyperparameter_tuning`, which trains/evaluates models using the same validation set.

We chose to use classification accuracy as the primary evaluation metric. It is defined as:

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions}$$

We chose classification accuracy because the dataset is multi-class and reasonably balanced (i.e. no single class dominates the responses), so accuracy provides an honest measure of model performance. Other metrics (i.e. precision, F1 score) are usually more appropriate when there is a significant class imbalance, which is not the case here. In our code, accuracy was computed using the `accuracy()` function, which was applied uniformly across all experiments:



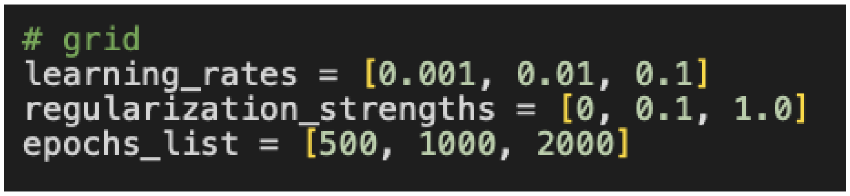
```python
def accuracy(y_true, y_pred):
    """
    calculate classification accuracy (in %).
    """
    # comparing the true vs. predicted values
    correct_predictions = y_true == y_pred
    # mean accuracy times 100
    return np.mean(correct_predictions) * 100
```

Hyperparameter Tuning

We explored three key hyperparameters in our logistic regression model:

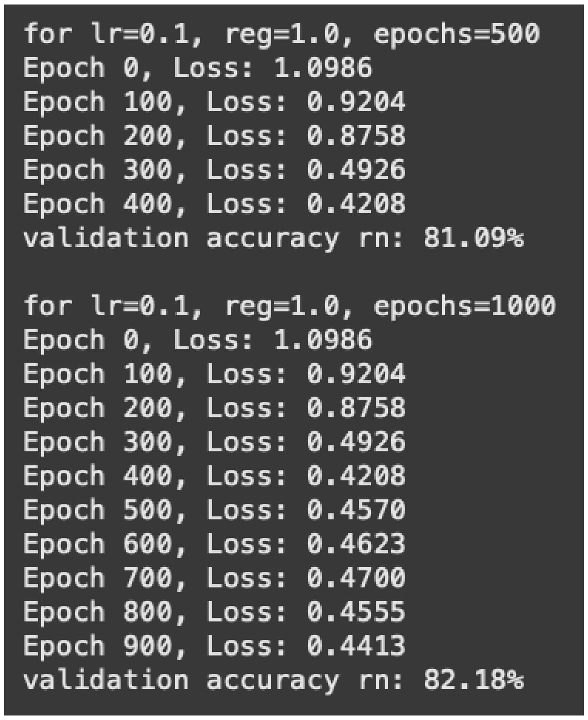
- Learning rate (`lr`), which controls the size in gradient descent.
- L2 regularization strength (`reg`), which determines the impact of the L2 penalty.
- Number of training epochs (`epochs`), which dictates how many full passes over the training data.

These parameters were tuned using a grid search over the following values:



```
# grid
learning_rates = [0.001, 0.01, 0.1]
regularization_strengths = [0, 0.1, 1.0]
epochs_list = [500, 1000, 2000]
```

For each combination, we trained a logistic regression model and evaluated its validation accuracy. The `hyperparameter_tuning()` function returned the best-performing model and the corresponding hyperparameters. Here are some sample outputs:



```
for lr=0.1, reg=1.0, epochs=500
Epoch 0, Loss: 1.0986
Epoch 100, Loss: 0.9204
Epoch 200, Loss: 0.8758
Epoch 300, Loss: 0.4926
Epoch 400, Loss: 0.4208
validation accuracy rn: 81.09%

for lr=0.1, reg=1.0, epochs=1000
Epoch 0, Loss: 1.0986
Epoch 100, Loss: 0.9204
Epoch 200, Loss: 0.8758
Epoch 300, Loss: 0.4926
Epoch 400, Loss: 0.4208
Epoch 500, Loss: 0.4570
Epoch 600, Loss: 0.4623
Epoch 700, Loss: 0.4700
Epoch 800, Loss: 0.4555
Epoch 900, Loss: 0.4413
validation accuracy rn: 82.18%
```

The following combinations were tried:

Learning rate (`lr`)	L2 regularization strength (`reg`)	Number of training epochs (`epochs`)	Validation accuracy (%)
0.01	0	1000	79.72
0.1	0	1000	82.65
0.1	0.01	1000	82.38
0.1	0.1	1000	80.49
0.1	0	2000	85.13
0.1	0.01	2000	83.88
0.1	0.1	2000	82.90
0.01	0.01	2000	80.11
0.01	0.1	2000	78.43

From this table, we observed the following trends:

- A learning rate of 0.1 consistently outperformed 0.01, indicating that faster convergence was beneficial.
- No regularization (i.e. `reg = 0`) yielded the highest accuracy, likely due to the simplicity and limited capacity of the model.
- Increasing epochs from 1000 to 2000 improved performance, suggesting that the model benefited from additional training.

Thus, the final hyperparameters selected (based on the validation accuracy) were:

- Learning rate (`lr`) = 0.1
- L2 regularization strength (`reg`) = 0
- Number of training epochs (`epochs`) = 2000

Final Model in `pred.py`

The final model in `pred.py` is based on multiclass softmax logistic regression and is trained with the best hyperparameters found during validation, as done in the Google Collab notebook. Key components include:

- The function `get_num_ingredients`, which is a helper function that extracts the number of ingredients from people's survey answers. If it finds numbers inside the text (i.e. "5"), it returns the last one as an `int`. Otherwise, it counts the number of commas and adds one (i.e. "salt, pepper, garlic" is three ingredients).
- The function `predict_all()` which takes a CSV file path containing survey data, pre-processes data, and generalizes predictions.
- The function `process_data()` which applies pre-processing (i.e. feature standardization) consistent with training data.

By using this setup, we ensure that the model logic in `pred.py` is consistent with training in the Google Collab notebook.

Part IV: Prediction

Throughout 1000 epochs, the logistic regression model converged steadily, reducing the loss from 1.0986 at initialization to 0.5250 at epoch 1000. We attained a training accuracy of 83%, a validation accuracy of 82.2%, and a test accuracy on unseen data (split in the data processing stage) of 85.1%. Thus, we report 85.1%, our test accuracy, as the expected performance as this comes from the unseen data and reflects our model's ability to generalize. Also, during the data processing stage, we ensured proper splitting of sets, so one can be assured that no bias was introduced.

Justification and Supporting Evidence

- Empirical evidence: The final test accuracy was computed on a true test set that was never seen by the model during training or validation. The performance improvement on the test set relative to training/validation suggests that the model generalizes well and is not overfitting.
- Feature representation: The features in our model were thoughtfully engineered through a variety of question types. Each categorical field was broken down into indicator features, while open-ended responses were handled with regex-based parsing. This ensured the model could learn from meaningful and structured inputs.
- Model simplicity: The model used in prediction is relatively simple (a linear classifier with softmax output and pre-learned weights and bias). Despite this simplicity, it performs well, indicating that the feature set is sufficiently expressive for the classification task.

- Consistency across splits: The close alignment between training (83.0%), validation (82.2%), and test (85.1%) accuracy indicates a stable model that does not overfit or underfit. There is no significant gap between training and test performance, which supports the reliability of our test accuracy estimate.

In conclusion, we expect our final model to perform at approximately 85.1% accuracy on the test set. This estimate is supported by empirical test results, stable performance across data splits, and careful feature engineering from diverse survey responses.

Part V: Workload Distribution

Our team divided the work strategically to ensure efficient collaboration and balanced contribution across all stages of the project.

Collaboration Schedule

As per our initial planning, we met weekly to discuss progress, align on implementation goals, and review each other's work. During the two critical weeks leading up to the model and report submission deadline, we increased our meeting frequency to 2 to 3 times/week. Meetings were held online via Zoom, on Mondays and Tuesday evenings.

Models Task Distribution

- Mahak Mishra completed the logistic regression model independently.
- Hia Aggrawal completed the neural network model independently.
- Dhvani Patel completed the nearest neighbour and decision tree models independently.

Hia Aggrawal and Dhvani Patel also worked on the data processing of the original data.

Communication

We coordinated using our Instagram and iMessage group chats for quick updates and shared documents via email. Code and results were regularly discussed and shared during virtual meetings to ensure alignment.

Report Collaboration

The report was collaboratively written using Google Docs, allowing all members to edit in real-time and divide sections based on the work they completed.

Task breakdown:

- Part 1: Data - This section was collaboratively completed by Mahak Mishra and Hia Aggrawal.
- Part 2: Model - This section was collaboratively completed by Mahak Mishra, Hia Aggrawal, and Dhvani Patel according to the models they completed. With that said, Mahak Mishra completed the Logistic Regression section independently, Hia Aggrawal completed the Neural Network section independently, and Dhvani Patel completed the Decision Tree and Nearest Neighbour sections independently.
- Part 3: Model Choice and Hyperparameters - This section was completed independently by Mahak Mishra.
- Part 4: Prediction - This section was completed independently by Hia Aggrawal.
- Part 5: Work Distribution - This section was completed collaboratively by Mahak Mishra, Hia Aggrawal, and Dhvani Patel.

Collaboration and Final Integration

In addition to implementing individual models, all members also:

- Provided feedback and debugging help for each other's code.
- Participated in deciding which model to use for final predictions, based on comparative performance.

- Contributed to writing and editing the final report, documenting the sections corresponding to their implementations.

This structured a flexible division of work that allowed us to stay on track, build a deeper understanding of the models, and ensure that all members contributed meaningfully to the project.