CSC311: Introduction to Machine Learning

Project Report

Generative AI Classification

Submitted by

Amanda Wu, Emma Chow, Yanzun Jiang, Yousef Ibrahims

University of Toronto
Fall 2025

1 Executive Summary

In this project, we evaluated three different model families for predicting which large language model (ChatGPT, Claude, or Gemini) generated each response: Bagged Naive Bayes, Polytomous Logistic Regression, and Random Forest. Among these approaches, Bagged Naive Bayes model achieved the highest and most stable accuracy, reaching approximately 73% on unseen data. Naive Bayes stood out as the best model because it handled our high-dimensional, text-heavy features more reliably than more complex approaches.

We used two books ([1], [3]) as guidance. During data cleaning and model training/evaluation, we used various Python ([7]) libraries, including NumPy ([2]), Pandas ([5]), Matplotlib ([4]), Scikit-learn ([6]), and other Python standard libraries ([8]).

2 Data Exploration

Our dataset contains 275 students, to give a total of 825 data points, each labeled as ChatGPT, Claude, or Gemini. Each row has a mix of feature types: 3 open-ended text responses, 2 multi-select categorical questions, 4 Likert-scale ratings, and a model label. These features show a mixture of textual, categorical, and ordinal data. Since each student contributes exactly one response per model, the dataset is naturally balanced across three classes, so no concerns about label imbalance.

During exploration, we identified specific data consistency issues that required attention. Some rows contained missing values, especially in the text and multi-select fields, and several cells included the placeholder "#NAME?". The open-ended responses also contained some noise, including bracketed text references such as "[THIS MODEL]". The numerical features were encoded as strings mixed with text (e.g., "4 — Often"), preventing direct analysis.

To pre-process the data, we first imputed missing values to avoid dropping rows with some missing responses. We then cleaned the text responses by removing placeholders and punctuation and standardizing formatting for better model predictions. We then used TF-IDF vectorization to convert the three open-ended responses into numerical features. For the two multi-select questions, we extracted all task types from the training split and encoded each row with a MultiLabelBinarizer to obtain multi-hot vectors. These transformations were done so that they can be used for a supervised learning problem. For the rating fields, we extracted the leading numeric value and scaled all four ratings using MinMax scaling so they would be on comparable magnitudes with the text vectors to reduce feature bias. After these transformations, all numerical, textual, and categorical features were concatenated into a single feature matrix, which served as the input for our models.

We avoided using pre-made dataset splitting solutions as all responses by a student must be kept together. This prevents our model from "peaking" into the test set as a student's responses' are related. To avoid this data leakage, we performed a grouped split using "student_id" to ensure that all three responses from a given student always remain in the same split. We reserved 16% of the unique student IDs for the test set and used the remaining 84% for training and validation.

Finally, the insights from exploration influenced our model choices. The application of TF-IDF and MultiLabelBinarizer resulted in a high-dimensional, sparse dataset. This observation led us to deprioritize simpler distance-based methods like KNN, which often struggle with the "curse of dimensionality" in such spaces. Instead, we focused on models that work better with sparse, high-dimensional features: logistic regression with polynomial terms, random forests, neural networks, and Naive Bayes. We also wanted to test whether the relationships in our data were linear or more complex, so comparing these different model types made sense. Numerically data was of great importance when text data lacked depth. However, text data was more revealing when they had depth.

3 Methodology

We made the choice to focus on three model families: Decision tree ensembles (Random Forest), Naive Bayes, and a stacked classifier using logisitc regression with a polynomial feature map and random forests. These felt appropriate as they were able to handle high data irregularity as we could tell little about the data's distribution. For the logistic regression model in the stacked classifier, we used the saga optimizer with high L_1 ratio regularization strength. For all three models, we considered accuracy and recall as our main evaluation metrics, as this classification is not concerned with the consequences of a false miss-classification on some class, which would've warranted an optimization of precision. We chose a vocabulary size of 60 for Naive Bayes, 160 for our stacked classifier, and 500 for Random Forest. These feature sizes were discovered by mapping how the accuracy of each model varies for multiple hyperparameter options and 36 runs.

3.1 Naive Bayes

Naive Bayes was appropriate as it had little parameters and hyperparameters to learn, which made it quick; this made it more feasible to train as we were dealing with high dimensional text data. Moreover, although the responses were definitely dependent, it seemed that this dependence was not high enough when compared to what each feature contributed

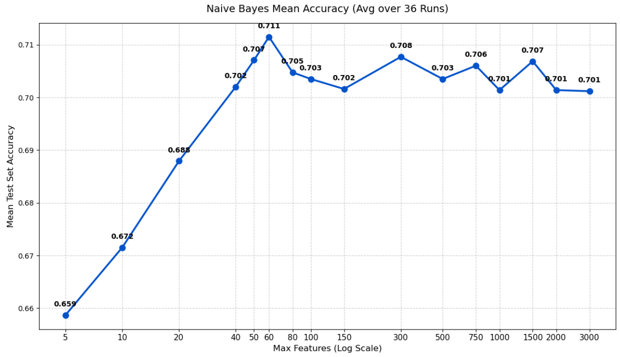

Figure 1: Sample of graph for Naive Bayes across 36 runs.

in isolation. Despite the naive assumption, the model performed surprisingly well.

For validation, we employed k-Fold cross-validation with 7 splits, using the student ID as the grouping variable. The hyperparameter tuning process used 'GridSearchCV' to explore combinations of two key parameters: the number of simple Naive Bayes models in the bagging ensemble (n_estimators: 10, 30, 50, 100) and the smoothing parameter (alpha: 0.1, 0.5, 1.0, 2.0, 5.0). Smaller alpha values (0.01-0.1) trust the training data more, while larger values (0.5-1.0) apply stronger smoothing. Our validation results showed that alpha values between 0.1 and 0.2 achieved the best performance (70-73% validation accuracy). After selecting the optimal alpha, we retrained on the combined training and validation sets before final test evaluation.

For Bagged Naive Bayes, we explored n_estimators [10, 30, 50, 100] and 'alpha' [0.1, 0.5, 1.0, 2.0, 5.0]. These values were chosen to cover a reasonable range - for n_estimators, starting from a modest ensemble size of 10 up to 100 to balance performance and computational cost, while for alpha, covering both smaller values (0.1, 0.5) for minimal smoothing and larger values (2.0, 5.0) for stronger regularization.

For evaluation, we used accuracy as our primary metric because the dataset is perfectly balanced across the three classes, making accuracy directly interpretable. To better understand class-specific behaviour, we also examined additional metrics, primarily recall, which gave us a sense of how well our model was identifying specific LLMs.

For implementation, all experiments were conducted using scikit-learn, supported by a structured feature-engineering pipeline. The three open-ended text responses were transformed using separate TF-IDF vectorizers (with max_features=3000 and English stop-word removal), while the two multi-select questions were encoded using MultiLabelBinarizer to produce multi-hot vectors. The four Likert-scale ratings were cleaned into numeric form, scaled using MinMaxScaler, and then expanded with PolynomialFeatures (degree 3) to capture simple interaction terms between ratings. These components were concatenated into a single high-dimensional feature representation used by all of our models.

3.2 Random Forest

Random Forest was our top contender as it imposed little assumptions; this was crucial as we could hardly ration any "normal" assumption about our data, given the presence of open ended questions, and the lack of regularity in responses. Moreover, being a bagged model, this provided more stable predictions that are less susciptible to noise. All in all, given our dataset's combination of high-dimensional TF-IDF text features, multi-hot categorical encodings, and polynomial-expanded numerical ratings, Random Forest provided was suitable to model the complex relationship across the inputs.

For training and validation, we used a 3-fold cross-validation strategy applied on the training split. Although Random Forest does not require an explicit optimizer, its performance is sensitive to tree-level hyperparameters, so we conducted systematic tuning using GridSearchCV. The hyperparameter grid included the number of estimators (100, 200, and 300), the maximum tree depth (5, 10, 15, and None), the minimum number of samples required for a split (2, 5, and 8), and the minimum number of samples required at a leaf (1, 5, and 10). We selected these ranges to balance computational cost with model expressiveness: shallow trees tend to generalize better on sparse TF-IDF features, whereas deeper trees allow the model to capture more interactions among the rating and categorical features. GridSearchCV with accuracy scoring was used to identify the best-performing configuration.

Because the dataset is exactly class-balanced, accuracy was used as the primary tuning metric. Additional inspection using recall and the confusion matrix helped identify whether particular classes were more susceptible to misclassification, especially given Random Forest's tendency to form uneven decision boundaries when sparse TF-IDF features dominate the feature space.

Random Forest used exactly the same feature pipeline as Naive Bayes and the stacked classifier. All three text fields were vectorized independently with TF-IDF (max_features = 1000, English stop-words removed). Both multi-select questions were encoded using MultiLabelBinarizer. The four Likert-scale rating features were scaled with MinMaxScaler and expanded with a degree-3 PolynomialFeatures transformer to expose interactions between rating values. All components were concatenated into a single feature matrix before training.

3.3 Stacked Classifier

The stacked classifier combined logistic regression with polynomial features and a Random Forest. This combination was chosen to integrate two complementary modeling styles: logistic regression captured structured relationships among the numerical ratings after polynomial expansion, while Random Forest contributed non-linear capacity suited for TF-IDF features and multi-hot categorical variables. Stacking allowed the final model to make use of the strong predictive power of each model in a linear and non-linear setting. The optimization component of the stacked model centered on the logistic regression block. We used the SAGA optimizer, which supports efficient training with high-dimensional sparse inputs and allows flexible regularization. We tuned the inverse regularization strength C, testing values 0.2, 0.4, 0.6, and explored different L1 ratios between 0.2 and 1.0 to test the two extremes; unsurprisingly, an L1 ratio of 1 was chosen as we expected some text features to be dropped. For the Random Forest, we reused the same hyperparameter grid as in the stand-alone Random Forest model, including variations in tree depth, estimator count, and splitting criteria.

For validation, we applied a 3-fold cross-validation scheme during hyperparameter tuning using GridSearchCV. The stacked classifier has a larger effective hypothesis space compared to single models, so cross-validation was especially important for preventing overfitting. Accuracy remained the primary selection metric due to class balance, but we monitored recall and the confusion matrix as additional diagnostic tools. In particular, the stacked model sometimes amplified class-specific biases if one model in the stack disproportionately favoured a label, so secondary metrics were useful for identifying these patterns.

The stacked classifier used the same feature-engineering pipeline as the other models for consistency. Logistic regression operated on the numerical ratings expanded with degree-3 polynomial features. Random Forest received both these outputs and the original TF-IDF and categorical features through scikit-learn's StackingClassifier framework. TF-IDF vectorization, multi-hot multi-select encoding, and MinMax-scaled rating features were all precomputed before being passed to the classifier.

4 Results

We evaluated our Naive Bayes model (with optimal n_estimators=100, alpha=0.1) across 200 different random seeds to systematically verify that our performance is not an artifact of a particular data split. Each seed produces an independent 84-16 train-test partition at the student level, ensuring no overlap in the students between splits. Out of 200 seeds tested, we found 31 seeds (15.5%) that achieved test accuracy of 70% or higher, with test accuracies ranging from 70.45% to 75.00%. This high success rate demonstrates that strong performance is achievable across many different data configurations, not just a single fortunate split.

Across 46 random runs, the following confusion matrix was produced. As expected most

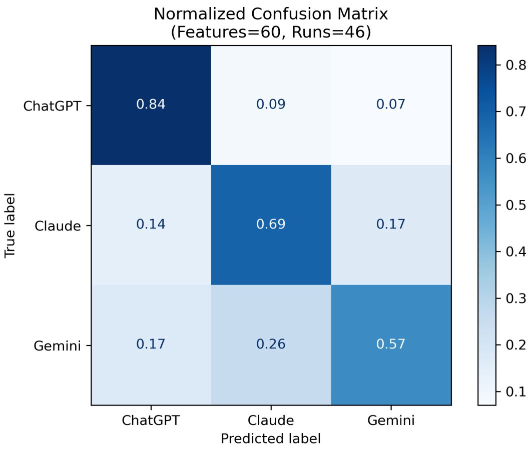

Figure 2: Confusion Matrix for Naive Bayes across 45 runs.

miss-classifications happen when it comes to differentiating between Claude and Gemini text responses. Indeed, this issue is further verified by subpar recall of Claude and Gemini responses:

Table 1: Classification Report Metrics

Class	**Precision**	**Recall**	**F1-Score**	**Support**
ChatGPT	0.73	0.84	0.78	2024
Claude	0.67	0.69	0.68	2024
Gemini	0.70	0.57	0.63	2024
Accuracy			0.70	6072
Macro Avg	0.70	0.70	0.70	6072
Weighted Avg	0.70	0.70	0.70	6072

At the end, taking into consideration how our model performs across many seeds, we can confidently say that the **Naive Bayes Projected Accuracy is 70%**

5 Contributions and Learning

Amanda:
Contribution: Completed the data exploration section of the report and helped shape the analysis for methodology section. Implemented the code for training Naive Bayes, including producing the JSON parameter file and writing the full pred.py script.
Lesson learned: I learned how important it is to fully understand the structure of the data before building models, and how tuning is important to the performance.

Emma:
Contribution: Completed implementation and tuned hyperparameters for Bagged Naive Bayes and Bagged polyreg. Contributed and implemented pred.py drafts.
Lesson learned: Do not overcomplicate models as it may not improve performance, and document every result and change.

Yanzun:
Contribution: Implemented code for data cleaning and Bagged Naive Bayes; completed methodology part for Bagged Naive Bayes in the report; created reference list for the report.
Lesson learned: Never drop any features just because you don't know how to fit that feature into your model, since this could lead to a biased result.

Yousef:
Contribution: Fixed code for data cleaning; made data processing pipeline shared across all models; implemented polyreg; edited the report, wrote the results section and produced graphics.
Lesson learned: Planning ahead of time and understanding any problems is worth the time spent.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. Array programming with NumPy, 2020.

[3] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[4] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[5] Wes McKinney. Data structures for statistical computing in python. *Proceedings of the 9th Python in Science Conference*, 445:51–56, 2010.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] Python Software Foundation. Python language reference, 2023. Version 3.10.5.

[8] Python Software Foundation. Python standard library, 2023. Includes random, re, and string modules.