1 Introduction [15 points]

- Group members: Stavya Arora, Isha Goswami, Ivy Brainard
- Kaggle team name: SII
- Ranking on the private leaderboard: 19
- AUC score on the private leaderboard: 0.84191
- Colab link: https://colab.research.google.com/drive/1aIMDFm4L2Vo_E0uBjltnUUZAa7ShxbV3?usp=sharing
- Piazza link: https://piazza.com/class/m4xf8vvl4206zc/post/127
- Division of labor: Each of the group members contributed equally to this project. We each tried different initial models and explored data processing and error measurement techniques.

2 Overview [15 points]

Models and techniques attempted

Upon examining the training and test data provided to us, we first cleaned the data before it could be trained in a model, including handling missing values, normalizing numerical features, and more. We ensured consistency between the training and test sets by aligning features and scaling the data using *StandardScaler*.

After some initial data exploration and feature importance analysis, we, experimented with several machine learning techniques for this binary classification problem. Initially, we trained Logistic Regression as a baseline model, which provided a reasonable but suboptimal AUC score. Next, we configured a neural network with various dense, batch normalization, leaky ReLU, and dropout layers, but it was quickly clear that this was not a problem best solved by neural networks.

Our next attempt, a Random Forest Classifier, had promising results because it leveraged ensemble learning, but we noticed that further tuning was required. We extracted feature importance from the model, identifying the most relevant variables contributing to predictions.

Finally, we experimented with extreme gradient boost (XGBoost), which yielded the best results. By tuning hyperparameters such as learning rate, max depth, number of estimators, and regularization parameters with Bayesian optimization with cross-validation, we achieved an AUC cross-validation score of approximately 0.8332 on our validation set. This confirmed that XGBoost was the best-performing model in our pipeline.

In summary, we tested multiple machine learning techniques, with Random Forest and XGBoost proving to be the most effective models for our dataset.

Work timeline

Step 1: Data cleaning and examination

We first determined which features and columns should be removed from training and test sets, observing the best approaches for removing or filling in missing data.

Step 2: Quantitative feature analysis

Using robust statistical methods, we drew conclusions about more features that should be dropped or emphasized. We also constructed visual representations of data and features.

Step 3: Initial model testing

We started with preliminary machine learning techniques of logistic regression, random forests, neural networks. This step also involved experimentation with the best accuracy optimization techniques.

Step 4: More sophisticated model testing

After initial models were not successful, we tried XGB, model stacking with decision trees, LGBM, and XGB using an implementation of Bayesian optimization, finally deciding that XGB will be the most successful.

Step 5: Hyperparameter finetuning

This step heavily utilized cross-validation and error metrics for finding the best hyperparameters. We made use of iterative improvement of bounds for hyperparameters being searched in Bayesian optimization as well as manual tweaking.

3 Approach [20 points]

Data exploration, processing and manipulation

Our first approach to data preprocessing involved handling missing values, removing irrelevant columns such as ID features that did not match between training and test sets, retaining only numeric columns, and normalizing these numerical columns (using z-score normalization) to reduce potential biases from differing scales within features. Initially, we assumed this preprocessing was sufficient, but early model testing revealed suboptimal performance, prompting further feature engineering.

To refine our approach, we began by analyzing pairwise feature correlations using a heatmap (*seaborn.heatmap*) to help identify highly correlated features as these could introduce unnecessary noise. Additionally, we used feature importance scores from our baseline Random Forest model to determine which features had minimal predictive value. Then we examined feature distributions to understand their skewness, realizing that the standard normalization might not apply to all. As you can see in our images, several features including 'speechiness', 'acousticness', 'instrumentalness', 'liveness', and 'duration_ms', exhibited strong either right-skewed distributions, to which we applied log transformations, or left-skewed features, which were standardized to center them around zero. Additionally, we experimented with treating variables that were clustered at 0 and 1 as binary categorical features rather than continuous. However, after computing Pearson correlations with the Popularity Type, we found that keeping them as continuous variables resulted in stronger correlations, justifying our decision to leave them unchanged.

Our most rigorous approach involved creating new features to enhance model performance. One key transformation was converting the track album release date into a numerical representation of its age in years. Since some entries lacked a recorded release year, we imputed missing values using the median of available data to avoid introducing bias.

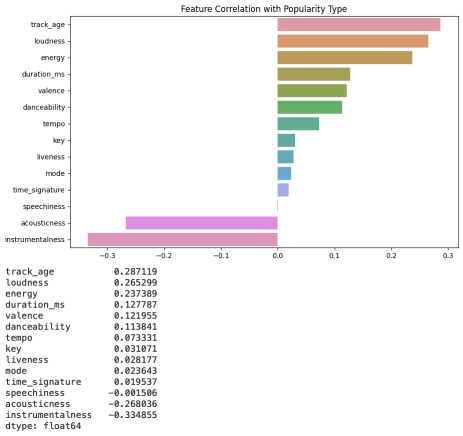
After all of these techniques, we plotted the correlation matrix of our normalized features with the popularity type. Based on weak correlations and feature importance rankings, we ultimately dropped features such as 'speechiness', 'mode', 'liveness', 'key', and 'time_signature'. While we expected these feature selection and transformation techniques to significantly improve model performance, we found that models trained on entirely unnormalized features performed better. However, we retained the step of removing irrelevant and weakly predictive features, which contributed to cleaner data and an overall improvement in our final accuracy scores.

Details of models and techniques

For model selection, we experimented with logistic regression, random forests, and XGBoost, testing different feature preprocessing and selection strategies. Logistic regression was chosen as a baseline for its easy interpretability and efficiency, but it struggled to capture more complex relationships from non-linearly separable data. Random forests provided robustness against overfitting due to bagging and handled these non-linear relationships well, but it still didn't provide the most accurate predictions. Our XGBoost method, which leveraged gradient boosting, proved to be the most accurate, however it required careful hyperparameter tuning to prevent overfitting.

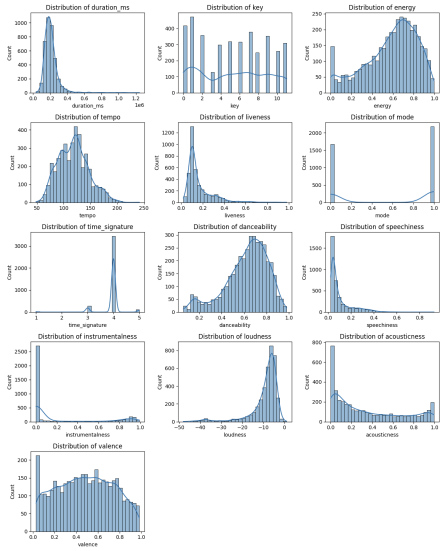
4 Relevant Figures

Here are some relevant figures related to our previously discussed data analysis and approach.

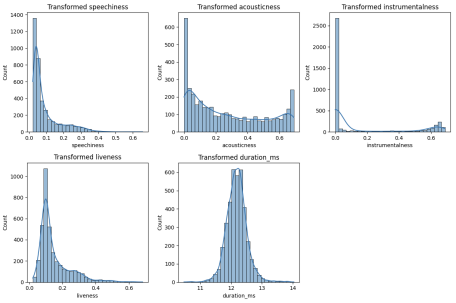



```
track_age           0.287119
loudness            0.265299
energy              0.237389
duration_ms         0.127787
valence             0.121955
danceability        0.113841
tempo               0.073331
key                 0.031071
liveness            0.028177
mode                0.023643
time_signature      0.019537
speechiness        -0.001506
acousticness       -0.268036
instrumentalness   -0.334855
dtype: float64
```

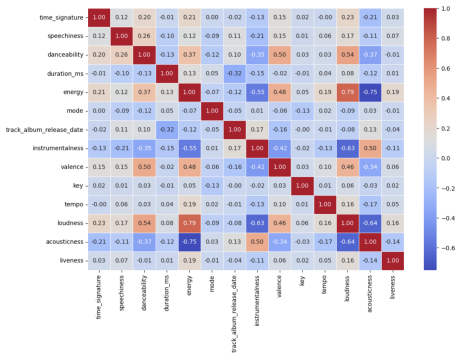
(a) Final Normalized Feature Correlation



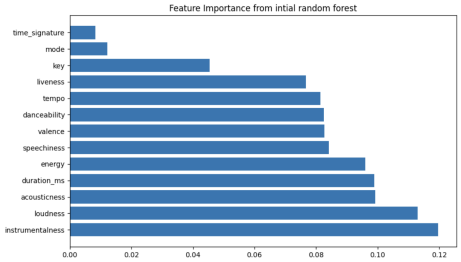
(b) Feature Distribution Before Normalization



(c) Feature Distribution After Normalization



(d) Pairwise Feature Correlation Heatmap



(e) Feature Importance from Random Forest

5 Model Selection [20 points]

Scoring

Since the main goal of the models was to accurately predict the popularity of various songs in the test set based on the audio and time characteristics, we prioritized this in the evaluation of our models. More specifically, we used the *roc_auc_score* from the *sklearn.metrics* library because this is what would be used to evaluate our final model on the test set. This optimization technique ensures that our model is robust in terms of minimal false positive rates while maximizing true positive rates. We fine-tuned hyperparameters for primarily the extreme gradient boost model (XGB) with roc_auc as the parameter for scoring. The Random Forest and XGBoost models scored the best in terms of accuracy and roc-auc score. XGBoost in particular was able to handle complex interactions between features, due to its flexible, decision tree nature, which resulted in higher performance compared to the logistic regression model.

Validation and test

From the training set, we separated a small proportion of it for validation to measure accuracy as specified above and to fine-tune hyperparameters based on this. A validation set allows us to determine whether the model may be overfitting to the training set and differentiate between various models. We used random splitting with *train_test_split* from the *sklearn.model_selection* library, using the parameter 0.2 for test size. So, 20% of our training data, about 772 samples, were not used during training but rather during validation. We also stratified on the prediciton variable, y, so that there was an even distribution between the training and validation set of positive and negative examples. This ensures that the model remains balanced and that both classes are represented proportionally in both the training and validation sets. This also avoids bias that could have resulted from one category being emphasized more.

To optimize hyperparameters, we explored three levels. We began with manual adjustments to establish baselines, which helped develop intuition but was pretty inefficient for fine tuning our results. Next, we applied randomized search (RandomizedSearchCV), which sampled different combinations of hyperparameters randomly within ranges that we predefined. This approach was more efficient than a brute force grid search but it had no refinement process, i.e did not improve based on promising results. Our most advanced approach used Bayesian optimization (hyperopt and scikit-optimize), which modeled the search space using techniques utilizing probability, essentially guiding the optimization towards the hyperparameter sections that were higher performing. While Bayesian optimization yielded the best results, it took a long time to run (computationally expensive). Given the timeline of the project, we balanced both the Bayesian optimization with randomized search for more rigorous tuning, while manual tuning remained useful for initial parameter exploration.

We also used cross-validation during training to assess model performance throughout training when optimizing hyperparameters and with Bayesian optimization, using *BayesSearchCV* from the *skopt* library. This allowed various combinations of hyperparameters to be searched while also keeping track of how well these hyperparameters generalize, as per the cross-validation score.

6 Conclusion [20 points]

Insights

Through this process of systematic testing, we refined our strategies to optimize our model's performance. While extensive normalization did not improve results for some of our models, our feature selection process helped to reduce unnecessary noise, ensuring that only the most relevant features contributed to the final predictions. By rigorously testing various modeling approaches and optimization strategies, we arrived at our final accuracy scores generated by our predictive model.

Among the models we tested, the XGBoost classifier with Bayesian optimization for hyperparameters proved to be the most effective in predicting song popularity. The Bayesian search allowed us to efficiently explore the hyperparameter space with many different combinations of hyperparameters, outperforming traditional grid search and or random cross-validation search. Compared to alternative models, XGBoost's ability to handle feature interactions and non-linearity in the data made it the superior choice.

Additionally, modifying the feature set such as incorporating the difference between the track release date and the present day instead of using given release dates contributed to a slight improvement in model accuracy. This adjustment better captured temporal trends and the impact of a song's age on its popularity. Based on our feature importance analysis, we identified all the columns to be important predictors besides time_signature, mode, and key. All the features were positive predictors, track_age having the highest predictive power, except instrumentalness and acousticness, which negatively infleunced the prediction target (shown in Figure A in section 4).

Overall, the combination of feature selection, hyperparameter tuning, and algorithm choice of XBG boosts with Bayesian cross-validation optimization led us to develop the most robust and predictive model for forecasting song popularity.

Challenges

One of the biggest challenges we faced was the long runtime of our model. Since XGBoost is a tree-based ensemble learning algorithm, training the model and fine-tuning hyperparameters using Bayesian optimization required significant computational power. Iterating multiple times on Bayesian search made the process even more time-consuming. To improve efficiency, we optimized feature selection and made parameter ranges closer from previous outputs.

Another key challenge was data cleanup and preprocessing. One major decision we made was to modify the date feature by calculating the difference between the track release date and the present day, rather than using the raw release date. We had tried other methods of not including the date because of inconsistent formats or standardizing the dates relative to the oldest one, but the one we ended up using had the best impact on popularity predictions. Additionally, after multiple tests, we decided not to normalize the data, as it did not significantly improve performance for our tree-based model, which naturally handles varying feature scales.

Time management was also a crucial factor, as model training and hyperparameter tuning took a considerable amount of time. Balancing different experiments while ensuring thorough testing required careful scheduling and prioritization to avoid bottlenecks.

Finally, we encountered an accuracy discrepancy when submitting predictions. Initially, we submitted binary classification outputs (0 or 1), but we later realized that using probability scores led to better results by providing a more precise representation of song popularity.

7 Extra Credit [5 points]

AUC is well-suited for this Kaggle competition because it evaluates the model's ability to rank popular and non-popular songs regardless of the threshold. AUC checks whether the model assigns higher scores to popular songs, even though some songs are considered popular with lower scores. This is especially important because popularity is quite subjective and in reality, popularity is more of a spectrum. This could be utilized to transform the problem into a prediction one instead of just classification (ie. *how* popular a song may be). AUC evaluates the model across all possible thresholds and it is less sensitive to class imbalance, so it is still a good indicator of accuracy even if there were more positive or negative examples in the test set.

Some additional data that could be interesting for this problem might be listener demographics, as certain types of music may be popular with certain types of listeners. Lyrics of the songs could have been useful too, as lyrics are often a large factor in the popularity of a song. Conducting sentiment analysis with natural language processing on the lyrics may have led to more refined predictions. Streaming platform data, such as the number of playlist inclusions, total listening minutes, or skip rates are all reflections of popularity. The additional data apart from audio characteristics would allow us to extend our results to different aspects of song popularity, providing a more holistic view of the factors that influence a song's success. By incorporating data about listener behavior and sentiment, we could better capture the multifaceted nature of popularity, which goes beyond just the audio features. This could also help refine models to predict not only whether a song might become popular but also why it resonates with certain audiences, enabling targeted recommendations or marketing strategies.

8 LLM Usage

Explanation for LLM

The ChatGPT response, for the first prompt, in the screenshot below provides a clear breakdown of hyperparameter tuning techniques, which is helpful for improving the performance of a song popularity prediction model. By explaining methods like GridSearchCV, RandomizedSearchCV, and Bayesian optimization, it helped us understand how to efficiently search for the best model parameters. This ensures our model is not just accurate but also computationally efficient, leading to better predictions of song popularity while optimizing training time. The second ChatGPT response, for the second prompt, provided clarity on how to organize and clean the test and train CSV files to use for our model. This helped build our model and later we were even able to adjust some features given to help improve our accuracy.

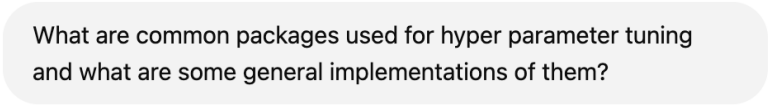


Figure 1: Interaction with L LM for data preprocessing.

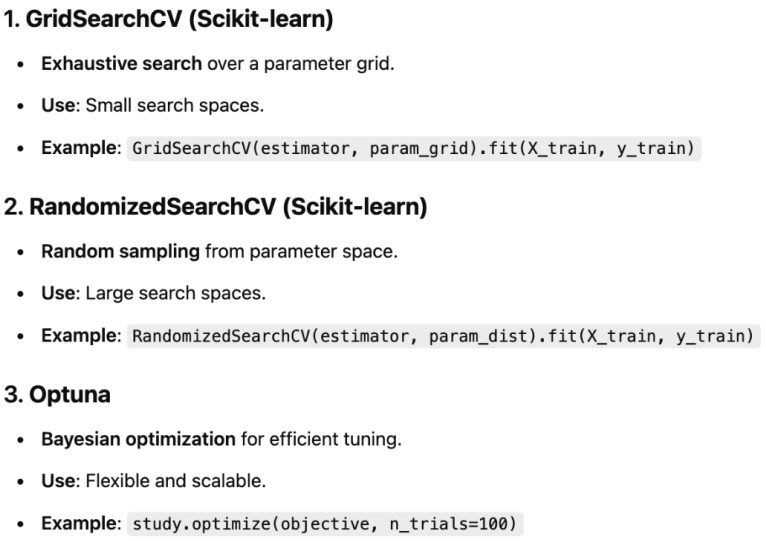
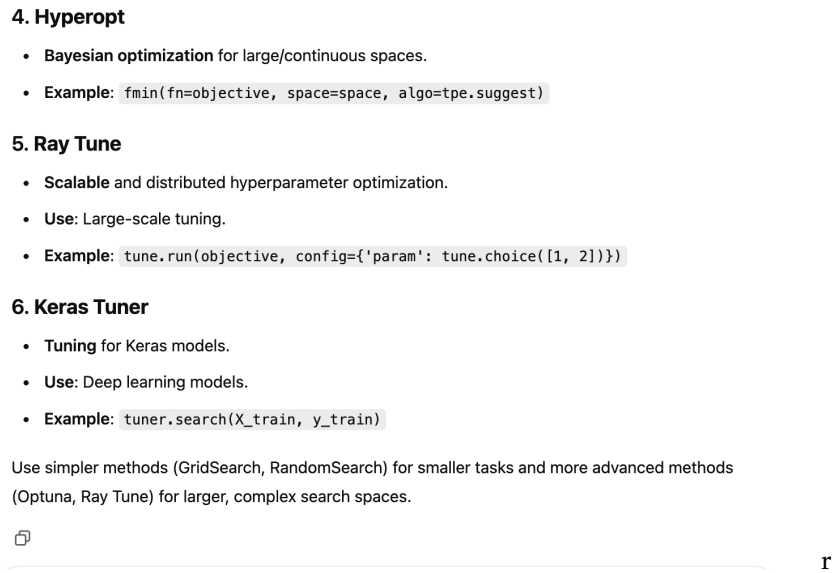


Figure 2: Integration of LLM's output for data preprocessing



r

Figure 3: Interaction with L LM for data preprocessing.

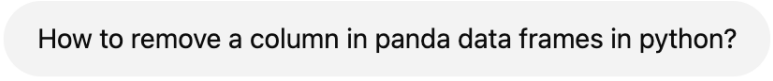


Figure 4: Interaction with L LM for data preprocessing.

To remove a column from a pandas DataFrame in Python, you can use the `drop()` method. Here's how to do it:

Syntax:

```python
df.drop('column_name', axis=1, inplace=True)
```

- `'column_name'`: The name of the column you want to remove.
- `axis=1`: This specifies you want to drop a column (not a row).
- `inplace=True`: This modifies the original DataFrame without returning a new one. If you set `inplace=False` or omit it, `drop()` will return a new DataFrame without the column.

Figure 5: Interaction with L LM for data preprocessing.

Example:

```python
import pandas as pd

# Sample DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data)

# Drop column 'B'
df.drop('B', axis=1, inplace=True)

print(df)
```

Output:

```
   A  C
0  1  7
1  2  8
2  3  9
```

Figure 6: Interaction with L LM for data preprocessing.

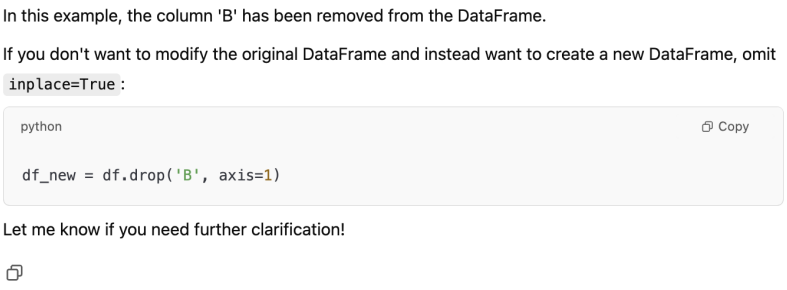


Figure 7: Interaction with L LM for data preprocessing.