POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

INF8245E - Fall 2022 - Kaggle Competition

Team name: EvilCorp

Sanmar Simon - 1938126 - sanmar-yared.simon@polymtl.ca
Matthew Fortier - 2216002 - matthew.fortier@polymtl.ca
Jire Christian Kandolo - 1877163 - Jire-Christian.kandolo@polymtl.ca

2022-12-03

Table of Contents

Feature Design

Data Preprocessing

The concert data in its initial form required cleaning. Each column had between 800 and 900 NaN values which would need to be filled or imputed in some way. A large part of this was initial data exploration and outlier elimination, and we've detailed that process extensively in Appendix A.

Since imputing is always approximate, we wanted to fill as many fields as we could deterministically. To accomplish this, we first constructed the data dependency graph shown in Figure 1 below.

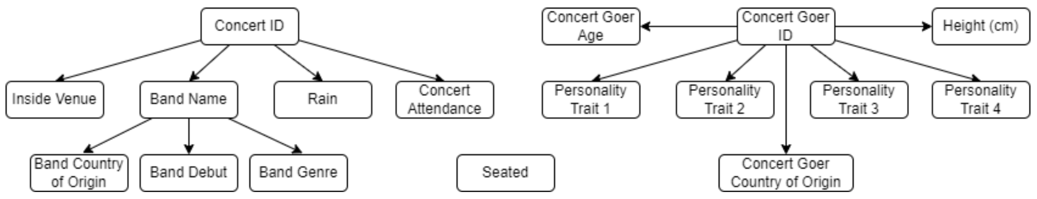
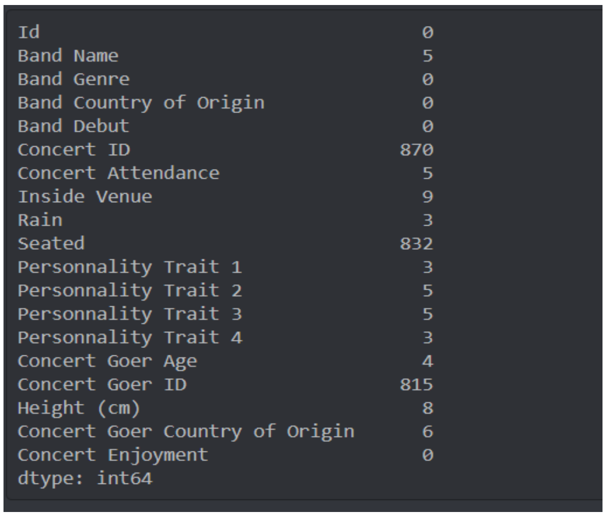


Figure 1: Data dependency graph

We confirmed that the attributes are in fact deterministic. For each concert goer, every associated record had the same personality traits, height, etc. This means that a record with a Concert Goer ID - but no height - could have that height filled with a value from another row from that concert goer. This method allowed us to fill every column besides Concert ID, Concert Goer ID, and Seated (the roots of the respective trees) at near 100% coverage. Figure 2 below shows the resulting NaN counts.



```
Id                                  0
Band Name                           5
Band Genre                          0
Band Country of Origin              0
Band Debut                          0
Concert ID                        870
Concert Attendance                  5
Inside Venue                        9
Rain                                3
Seated                            832
Personnality Trait 1                3
Personnality Trait 2                5
Personnality Trait 3                5
Personnality Trait 4                3
Concert Goer Age                    4
Concert Goer ID                   815
Height (cm)                         8
Concert Goer Country of Origin      6
Concert Enjoyment                   0
dtype: int64
```

Figure 2: NaN counts after deterministic filling

Missing IDs were given an “unknown ID” value, and the remaining empty values were imputed via column means. For boolean variables, we used a mapping of

[True,NaN,False] => [1,0,-1]. With these processes complete, the data was clean and free of null values.

Feature Engineering

Since there were many categorical (string) variables, we needed a way to represent them so that non-categorical algorithms could be used. Our first iteration used one-hot encoding for all these variables. This gave us a total of 226 columns, most of which were binary. Results from this experiment will be discussed later.

The next experiment was to transform categorical variables into a mean enjoyment score. The four target values were mapped to integer values:

```
data['Concert Enjoyment'].replace('Worst Concert Ever', '0', inplace=True)
data['Concert Enjoyment'].replace('Did Not Enjoy', '1', inplace=True)
data['Concert Enjoyment'].replace('Enjoyed', '2', inplace=True)
data['Concert Enjoyment'].replace('Best Concert Ever', '3', inplace=True)
```

Figure 3: Mapping target label to integer values

Categorical variables were then replaced with each category's mean score. So if the mean of all rows for "Band 1" was 2.35, then we replaced "Band 1" with 2.35. This yielded a data set with no categorical features and a reasonable dimensionality. With most columns in numerical format, we applied normal scaling to most columns (mean 0, standard deviation 1).This was especially helpful for columns such as Concert Attendance and Band Debut, which had exceptionally large values.

With the base data prepared, we experimented with synthesizing columns. The most successful of these were:

1. Concert score frequencies - Replace Concert ID with 4 columns, indicating the frequency of the 4 enjoyment scores for that concert.
2. Home country - A binary value; 1 if the concert goer and band were from the same country, 0 otherwise

Others were attempted, but their impact on the model was negligible. An exhaustive list of synthesized features (and their importances) are given in Appendix B.

Algorithms

We decided to throw a wide range of algorithms at this problem to get an initial idea of how each would perform. In the first iteration, the following algorithms were run (with mostly default parameters):

- KNN
- Logistic Regression
- Gaussian Naive Bayes
- Decision Tree
- Random Forest
- Gradient Boosting Tree
- XGBoost
- LightGBM

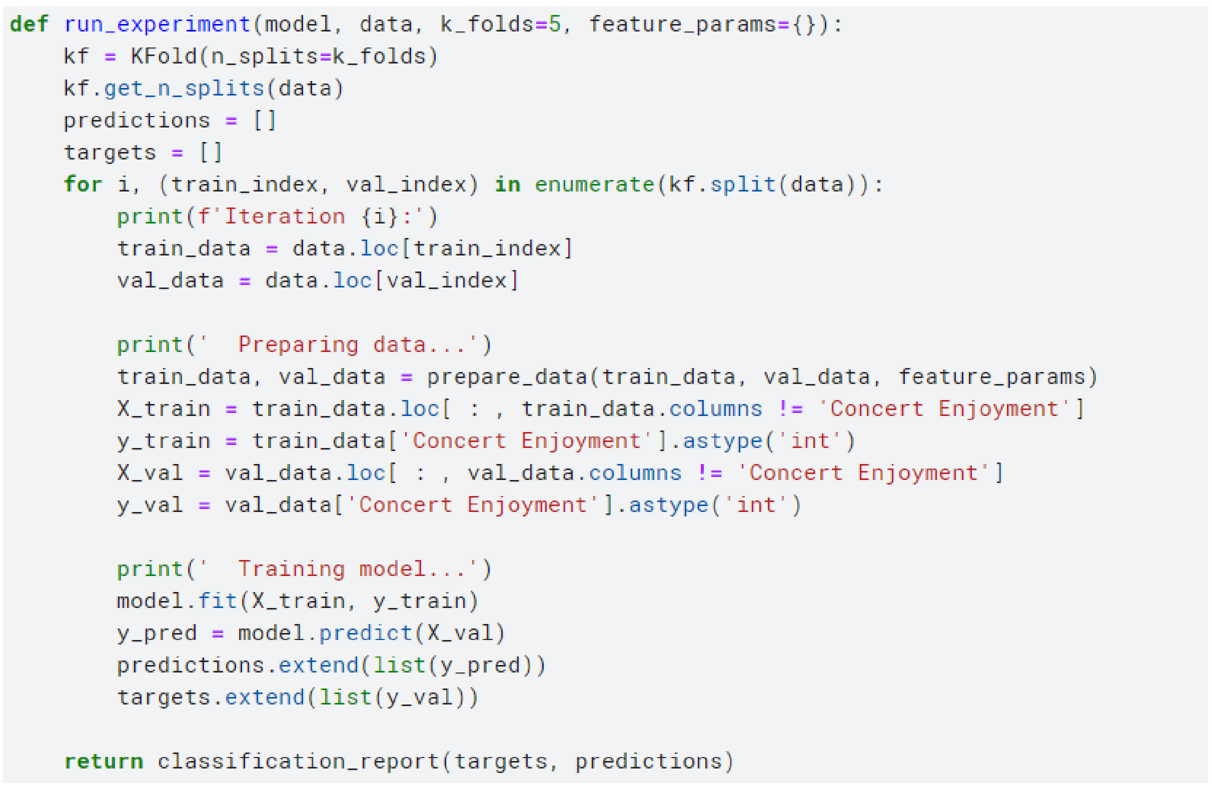
- Multilayer
  Perceptrons

The inclusion of logistic regression gave a good baseline for generalized linear models, but we suspected that this was going to be best solved with nonlinear models. The best results came from XGBoost, so most testing proceeded with that.

Methodology

Testbed Development

Due to the exploratory nature of this competition, we decided to set up a testbed function for easy experimentation. This testbed takes in the train data, a model type, and various parameters. It then runs a full experiment and prints the results to standard output:



```python
def run_experiment(model, data, k_folds=5, feature_params={}):
    kf = KFold(n_splits=k_folds)
    kf.get_n_splits(data)
    predictions = []
    targets = []
    for i, (train_index, val_index) in enumerate(kf.split(data)):
        print(f'Iteration {i}:')
        train_data = data.loc[train_index]
        val_data = data.loc[val_index]

        print('  Preparing data...')
        train_data, val_data = prepare_data(train_data, val_data, feature_params)
        X_train = train_data.loc[ : , train_data.columns != 'Concert Enjoyment']
        y_train = train_data['Concert Enjoyment'].astype('int')
        X_val = val_data.loc[ : , val_data.columns != 'Concert Enjoyment']
        y_val = val_data['Concert Enjoyment'].astype('int')

        print('  Training model...')
        model.fit(X_train, y_train)
        y_pred = model.predict(X_val)
        predictions.extend(list(y_pred))
        targets.extend(list(y_val))

    return classification_report(targets, predictions)
```

Figure 4: Testbed development

We leveraged k-fold cross validation to ensure performance was relatively consistent. Deterministic data filling is applied immediately, as it does not result in data leakage. However, imputation and feature engineering (performed inside the "prepare_data" function) are repeated for every train/val split to prevent leakage. Notably, 'feature_params' contained optional parameters controlling various feature engineering tasks such as column pruning, or specifying which synthesized columns to include.

Individualized Classifiers

In addition to traditional model development, we also attempted to create a system of individual classifiers for each concert goer. This meant one model for each unique concert goer in the training set, fitted only to that concert goer's records (and one dummy model for any new concert goer not in the training set). The hope was to capture more information about each individual's preferences.

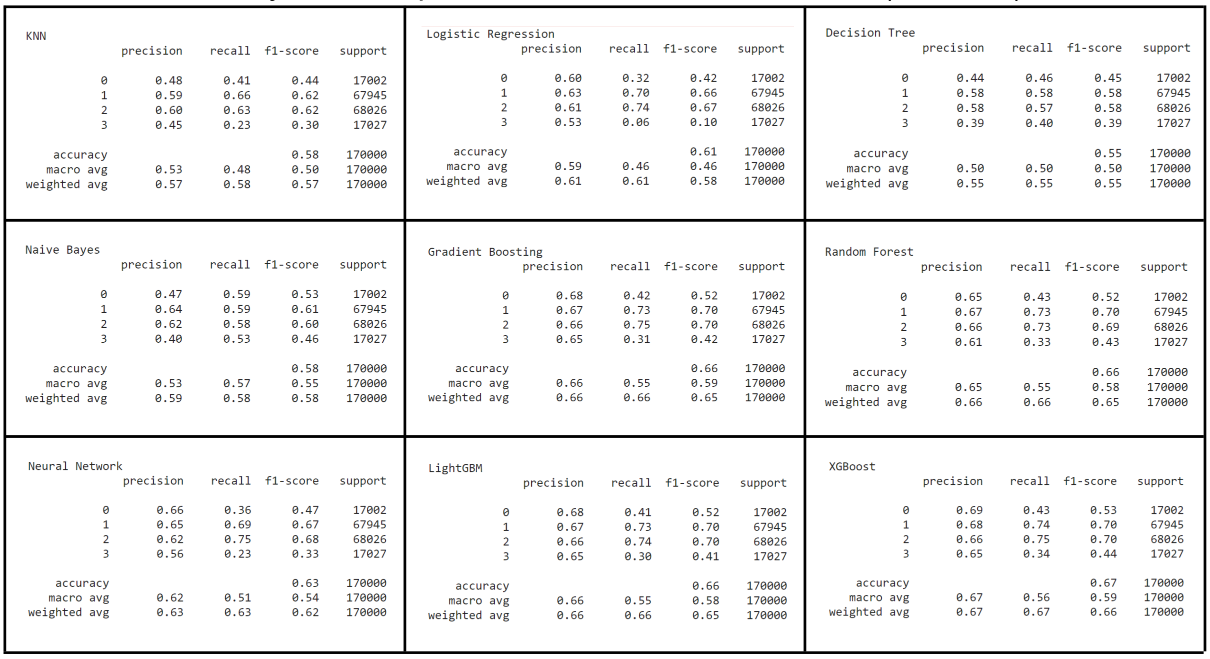
The data preparation was much more simple for this method, as we could effectively remove any columns with concert goer-specific information (height, personality traits, etc). Theoretically, this should help compensate for the smaller number of training samples per-model. However, we ended up abandoning this technique due to poor performance.

Results

Model Selection

Table 1 below shows the results of various model architectures using our test bed. These were performed after feature processing and largely using default hyperparameter values. An exception to this is neural networks, where we tried various hidden layer configurations (ex: (32), (64), (32,32), (64,64), etc). Ultimately, we found that gradient boosting classifiers tended to perform best, with XGBoost consistently coming out marginally better than the other models.

Table 1: Preliminary model reports with 5-fold cross validation (zoom in!)



```
KNN
              precision    recall  f1-score   support

           0       0.48      0.41      0.44     17002
           1       0.59      0.66      0.62     67945
           2       0.60      0.63      0.62     68026
           3       0.45      0.23      0.30     17027

    accuracy                           0.58    170000
   macro avg       0.53      0.48      0.50    170000
weighted avg       0.57      0.58      0.57    170000
```

```
Logistic Regression
              precision    recall  f1-score   support

           0       0.60      0.32      0.42     17002
           1       0.63      0.70      0.66     67945
           2       0.61      0.74      0.67     68026
           3       0.53      0.06      0.10     17027

    accuracy                           0.61    170000
   macro avg       0.59      0.46      0.46    170000
weighted avg       0.61      0.61      0.58    170000
```

```
Decision Tree
              precision    recall  f1-score   support

           0       0.44      0.46      0.45     17002
           1       0.58      0.58      0.58     67945
           2       0.58      0.57      0.58     68026
           3       0.39      0.40      0.39     17027

    accuracy                           0.55    170000
   macro avg       0.50      0.50      0.50    170000
weighted avg       0.55      0.55      0.55    170000
```

```
Naive Bayes
              precision    recall  f1-score   support

           0       0.47      0.59      0.53     17002
           1       0.64      0.59      0.61     67945
           2       0.62      0.58      0.60     68026
           3       0.40      0.53      0.46     17027

    accuracy                           0.58    170000
   macro avg       0.53      0.57      0.55    170000
weighted avg       0.59      0.58      0.58    170000
```

```
Gradient Boosting
              precision    recall  f1-score   support

           0       0.68      0.42      0.52     17002
           1       0.67      0.73      0.70     67945
           2       0.66      0.75      0.70     68026
           3       0.65      0.31      0.42     17027

    accuracy                           0.66    170000
   macro avg       0.66      0.55      0.59    170000
weighted avg       0.66      0.66      0.65    170000
```

```
Random Forest
              precision    recall  f1-score   support

           0       0.65      0.43      0.52     17002
           1       0.67      0.73      0.70     67945
           2       0.66      0.73      0.69     68026
           3       0.61      0.33      0.43     17027

    accuracy                           0.66    170000
   macro avg       0.65      0.55      0.58    170000
weighted avg       0.66      0.66      0.65    170000
```

```
Neural Network
              precision    recall  f1-score   support

           0       0.66      0.36      0.47     17002
           1       0.65      0.69      0.67     67945
           2       0.62      0.75      0.68     68026
           3       0.56      0.23      0.33     17027

    accuracy                           0.63    170000
   macro avg       0.62      0.51      0.54    170000
weighted avg       0.63      0.63      0.62    170000
```

```
LightGBM
              precision    recall  f1-score   support

           0       0.68      0.41      0.52     17002
           1       0.67      0.73      0.70     67945
           2       0.66      0.74      0.70     68026
           3       0.65      0.30      0.41     17027

    accuracy                           0.66    170000
   macro avg       0.66      0.55      0.58    170000
weighted avg       0.66      0.66      0.65    170000
```

```
XGBoost
              precision    recall  f1-score   support

           0       0.69      0.43      0.53     17002
           1       0.68      0.74      0.70     67945
           2       0.66      0.75      0.70     68026
           3       0.65      0.34      0.44     17027

    accuracy                           0.67    170000
   macro avg       0.67      0.56      0.59    170000
weighted avg       0.67      0.67      0.66    170000
```

Hyperparameter Tuning

After making the final decision on model architecture (XGBoost), we proceeded with hyperparameter tuning. XGBoost has many hyperparameters, but we focused mainly on the learning rate (“eta”) and tree depth (“max_depth”). Other parameters used to prevent overfitting such as “min_child_weight” and “subsampling” were explored, but were not found to have any impact on validation accuracy.

Random search was used for tuning. We first took a broad range of potential values; eta in [0.01, 0.5] and max_depth in [3, 20]. After 50 rounds of cross validation, we witnessed higher accuracies with eta around 0.2 and max_depth around 10. Random search was then repeated with eta range of [0.1, 0.3] and max_depth of [8,14]. The results are shown in Figure 3 below. This resulted in our final values of eta=0.2, max_depth=9.

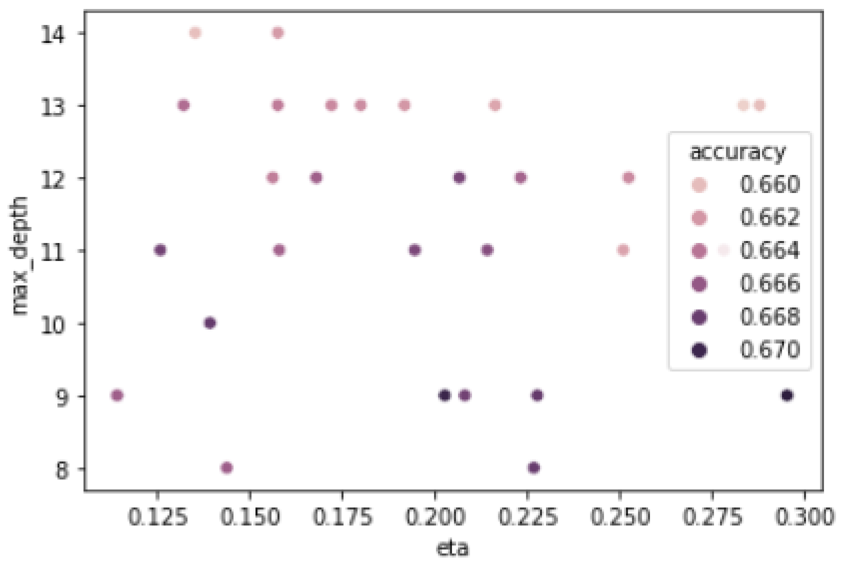


Figure 5: Refined random search of XGBoost hyperparameters

Performance

Our first pass on this problem used one hot encoding and naive mode-filling for data cleaning. Even so, we saw an accuracy of just over 0.65 on the test data. This gave us a baseline to beat, and guided our future data cleaning, feature engineering, and model selection processes.

Our final accuracy was 0.67235 using XGBoost and the hyperparameters specified above. Efforts to improve the score in the last week of the competition were difficult; further feature refinement and hyperparameter adjustments had diminishing returns. We feel that this is a solid score given the presented problem.

Discussion

Our approach to this problem had several strengths. We took a lot of time to analyze the data and understand its structure. Missing values were handled in a creative way, and minimal information was lost during imputation. Categorical variables were processed in more than one way, allowing for a comparison of techniques. We tried a wide variety of model architectures, and set up an easy-to-use test bench which made the testing process easier. Finally, we took great care to prevent data leakage during testing by interleaving our data processing pipeline within the cross-validation.

The complexity of our data pipeline had some drawbacks. It became difficult to test new synthetic features as our feature engineering functions grew. Some columns had to be created before imputation or normalization, while others relied on transformations that happened after imputation. The overhead of this process, along with our cross-validation approach, made it cumbersome to do ablative studies - especially if we wanted to test various architectures as well.

We also found limited value in hyperparameter tuning. This could be due to the robustness of XGBoost in general, but it may mean that we found a local minimum for our loss, with the true minimum being far away. Future work would include a more thorough exploration of model architecture and refactoring of our data pipeline to reduce developmental friction.

Statement of Contributions

All members of our group contributed to every stage of this project. However, there were tasks where one of us would take more of a leading role. Sanmar took the initiative as group leader to form the team, register for the competition, and set up our coding framework for data analysis and preliminary results. Jire focused on conceptualizing the problem, running experiments, and documenting results. Matthew worked on feature engineering, model fine-tuning, and document editing. We hereby state that all the work presented in this report is that of the authors.

Appendix A: Data Analysis and Cleaning

Exploratory Data Analysis

Exploratory data analysis (EDA) is an important step in a classification problem because it allows us to get a better understanding of our data and identify potential issues or interesting patterns. By examining the characteristics and relationships of the variables in a dataset, we can gain insights that can inform our decision making.

Additionally, EDA can help identify potential problems with the data, such as missing values, outliers, or errors, that need to be corrected before conducting more sophisticated analysis. This can help prevent incorrect or misleading conclusions from being drawn from the data.

Know Your Data

First step of EDA is getting familiar with the data we have. This includes reading our csv files and storing their content into Pandas dataframe, then looking at the number of features, the type of these features, whether there are null values or not etc… This first step is essential and once we notice the particularities of our dataset then we will proceed with cleaning it as explained in a few subsections below.

Univariate Analysis

In our case we used an univariate analysis on all the initial numerical features we had to get a look at their frequency distribution. We were able to get few insights out of the distributions. We notice that Personality Trait 1, 3 and 4 are apparently following a Normal distribution, and Concert Goer Age, and Height are following a uniform distribution.

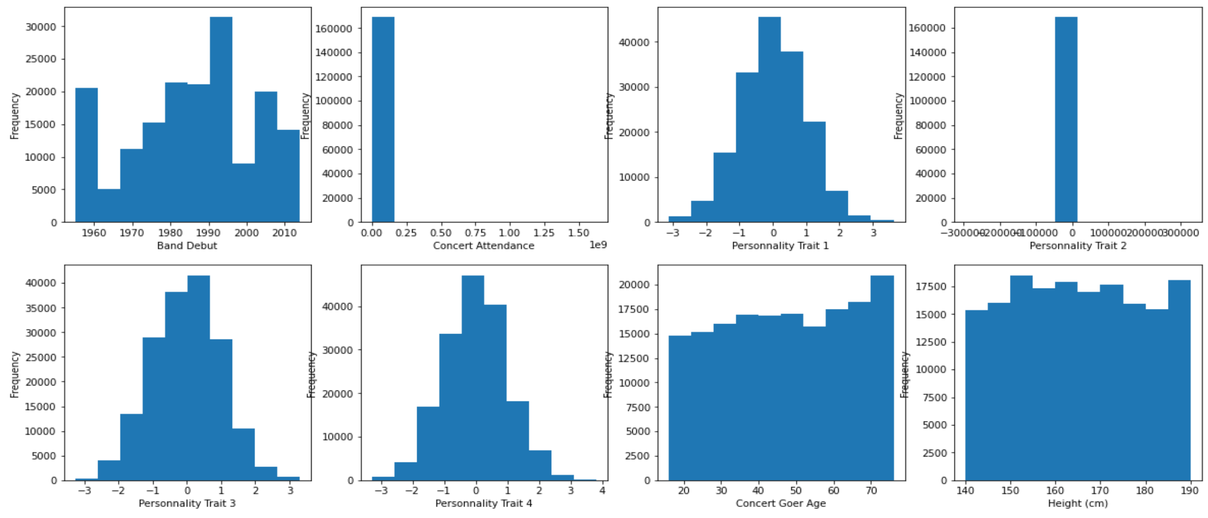


Figure A.1: Univariate analysis of numerical features

Bivariate analysis

Bivariate analysis was used to examine the relationship between our features and the target. This type of analysis allowed us to understand how the variables are related and how they affect each other. We compared all the categorical features to the target, and did the same for the numerical features. Here are the results.

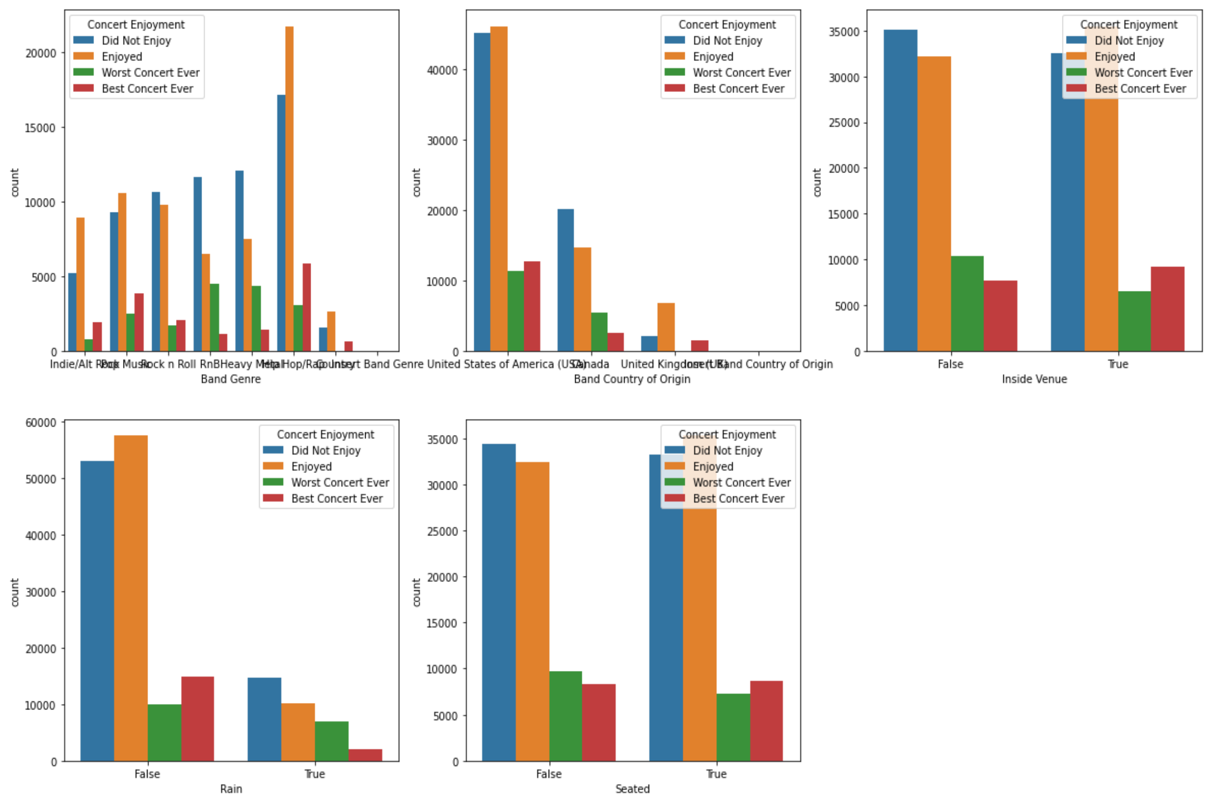


Figure A.2: Categorical features VS target

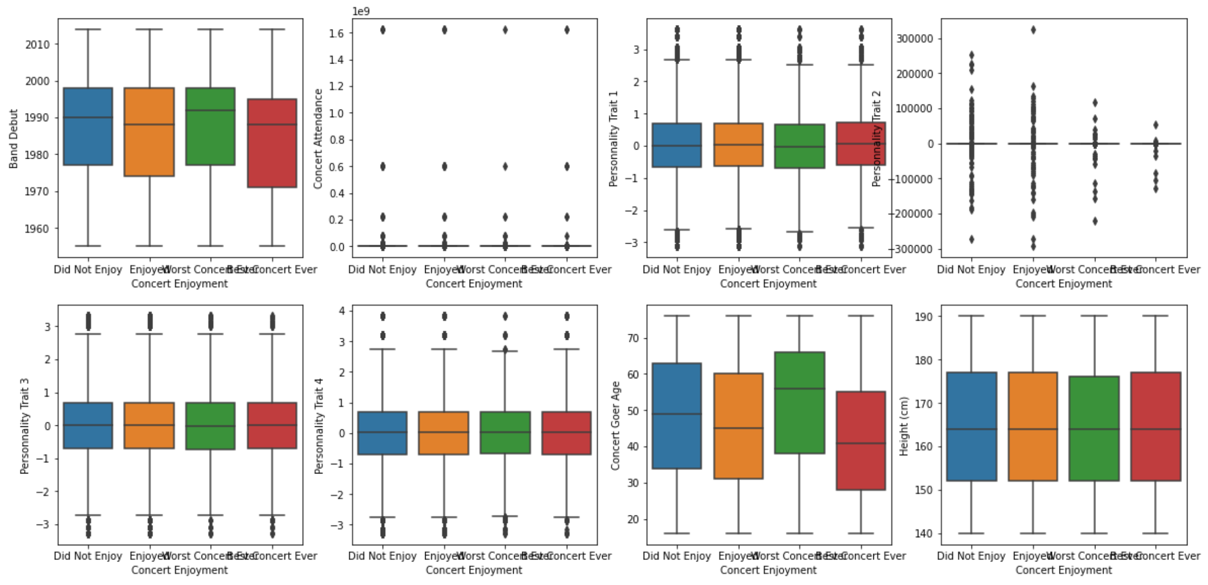


Figure A.3: Categorical features VS target

Cleaning and Imputation

We start with the data dependency graph described earlier. The first step was to turn this into a dependency dictionary; for each (k,v) pair if we know ‘v’, we can find ‘k‘ from the data:

```
# For each (k,v) pair here, if we know 'v', we can find 'k' from the data
dep_graph = {
    'Band Name': 'Concert ID',
    'Band Genre': 'Band Name',
    'Band Country of Origin': 'Band Name',
    'Band Debut': 'Band Name',
    'Concert Attendance': 'Concert ID',
    'Inside Venue': 'Concert ID',
    'Rain': 'Concert ID',
    'Personnality Trait 1': 'Concert Goer ID',
    'Personnality Trait 2': 'Concert Goer ID',
    'Personnality Trait 3': 'Concert Goer ID',
    'Personnality Trait 4': 'Concert Goer ID',
    'Concert Goer Age': 'Concert Goer ID',
    'Height (cm)': 'Concert Goer ID',
    'Concert Goer Country of Origin': 'Concert Goer ID'
}
```

Figure A.4: Features dependency dictionary

We use this dictionary to construct a lookup table which takes a Concert ID, Band Name, or Concert Goer ID, and returns a list of clean values which depend on that attribute.

```
data_map = {}
c_order = []
for att in ['Concert ID', 'Band Name', 'Concert Goer ID']:
    dependants = []
    for k,v in dep_graph.items(): # Get all dependant features
        if att == v:
            dependants.append(k)
    d = raw_train_data[dependants + [att]].groupby(att).agg(pd.Series.mode).to_dict('index')
    data_map.update(d) # Update the lookup table
    c_order.extend(dependants)
```

Figure A.5: Construction of lookup table

For example, the lookup value for “32.0” (a concert ID) would yield:

```
{
        ‘Band Name’: ‘Rubbish Devon Frogs’,
        ‘Concert Attendance‘: 2980.0,
        ‘Inside Venue’ : True ,
        ‘Rain’ : False
}
```

With this lookup table complete, we implemented a row-wise cleaning function to fill the missing data (note that ‘c_order’ and ‘data_map’ are globally accessible):

```python
def fill_NaN_deterministic(data):
    # define row-wise cleaning function
    def fill_NaN_row(row):
        for c in c_order:
            if pd.isnull(row.get(c)):
                row[c] = data_map.get(row[dep_graph[c]],{}).get(c)
        return row

    return data.apply(fill_NaN_row, axis=1)

# two things happenening here:
#   1. fill NaN values we can empirically determine. eg if we know the band name, we know the band debut & genre
#   2. Change boolean columns to [1, 0, -1] for [True, NaN, False]
def fill_missing_data(data):
    d = fill_NaN_deterministic(data)
    for c in ['Seated', 'Rain', 'Inside Venue']:
        d[c] = d[c].replace({True: 1, False: -1, np.nan: 0})
    return d
```

Figure A.6: Cleaning methods

The cleaning method does two things:

1. Fill NaN values we can empirically determine using the lookup table.
2. Change boolean columns to [1,0,-1] for [True ; Nan , False.
   this is further enhanced in this part of the algorithm

The last step was to create a custom imputing function to fill the remaining missing values. The two most difficult columns were Concert ID and Concert Goer ID, which can not be deterministically filled. We decided to create an ‘Unknown’ class for these, as it seemed to introduce the least amount of bias. Remaining NaN values for numerical attributes were simply filled with the column mean:

```python
# For categorical, create a new category. For numerical, take the mean.
def custom_impute(train_data, test_data):
    train_data['Concert ID'] = train_data['Concert ID'].astype(str)
    test_data['Concert ID'] = test_data['Concert ID'].astype(str)
    numerical_features = train_data.select_dtypes(include=np.number).columns.tolist()
    string_features = train_data.select_dtypes(include=object).columns.tolist()
    tr = train_data.fillna({c: f'unknown_{c}' for c in string_features})
    tr[numerical_features] = tr[numerical_features].fillna(tr[numerical_features].mean())
    te = test_data.fillna({c: f'unknown_{c}' for c in string_features})
    te[numerical_features] = te[numerical_features].fillna(tr[numerical_features].mean()) # make sure to use training data means
    return tr, te
```

Figure A.7: Custom impute function

Care was taken to only fill test data with values from the training set. For this reason, most of our data preparation functions work on both sets at the same time.

Appendix B: Feature Synthesis

Name	Description
Categorical Score Averages	As described in the Feature Engineering section, one of the first modifications we tried after one-hot encoding categorical columns was categorical score averages. A Concert ID was replaced with the mean Concert Enjoyment for that ID across all training samples (with Concert Enjoyment being recast as a numerical value). This was repeated for all categorical variables. A few of these were kept in the final model, while others were replaced with other representations.
Raining Outdoors	A boolean feature. Value: 1 if (Inside Venue = 0 && Rain = 1) 0 otherwise This was abandoned after being shown to have very little predictive capacity
Home Country	A boolean feature. Value: 1 if (Concert Goer Country of Origin == Band Country of Oirign) 0 otherwise Some predictive capacity found. Used in the final model
Band Personality Matrix	For each band, we created a 4x4 heatmap showing the mean value of each personality trait, aggregated by concert enjoyment level. An example heatmap is shown in Figure B.2. This was flattened into a 16-dimensional feature based solely on the Band Name for each sample. Despite promising results from the heatmaps, these features had little predictive power and were scrapped
Categorical Score Frequencies	Last feature tried. Similar to categorical score averages, except we used concert enjoyment frequencies for each row. For example, concert with ID 34.0 may have been replaced with: Concert ID Best%: 0.2 Concert ID Enjoyed%: 0.15 Concert ID Not Enjoyed%: 0.48 Concert ID Worst%: 0.17 This worked well for bimodal data, and frequencies were kept for a few categorical columns (Band Name and Concert ID). The rest were replaced with Categorical Score Averages.

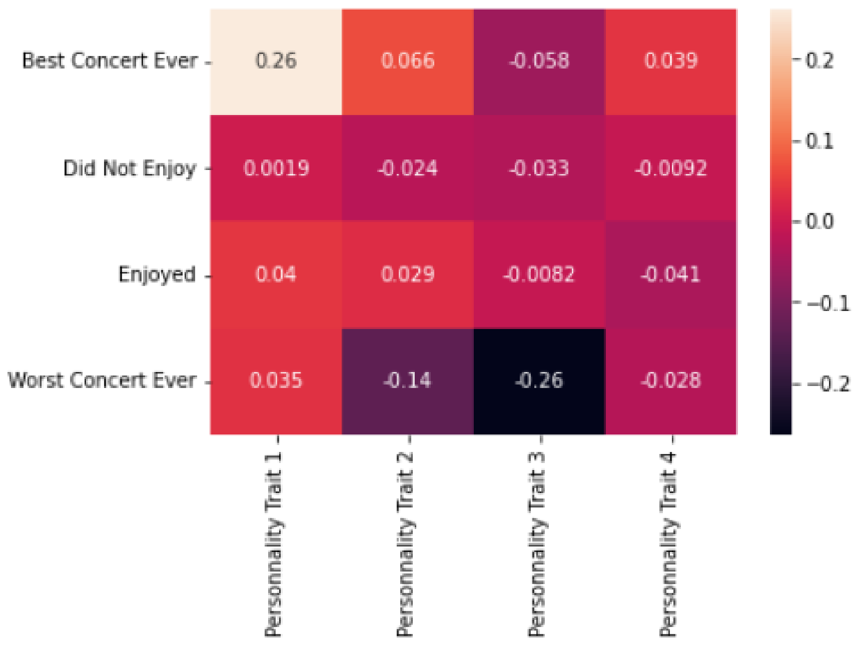


Figure B.1: Band Personality Matrix for “Teenage Crazy Blue Knickers”

```
{
    "Band Name": "0.015893925",
    "Band Genre": "0.010272022",
    "Band Country of Origin": "0.016209459",
    "Band Debut": "0.018740438",
    "Concert ID": "0.025802653",
    "Concert Attendance": "0.008084735",
    "Inside Venue": "0.0077593136",
    "Rain": "0.0074437824",
    "Seated": "0.036802933",
    "Personnality Trait 1": "0.007956608",
    "Personnality Trait 2": "0.007872433",
    "Personnality Trait 3": "0.0097208535",
    "Personnality Trait 4": "0.007925981",
    "Concert Goer Age": "0.02471738",
    "Concert Goer ID": "0.039441817",
    "Height (cm)": "0.0080873",
    "Concert Goer Country of Origin": "0.0098472405",
    "Band Name Best%": "0.012836473",
    "Band Name Enjoyed%": "0.0106812185",
    "Band Name Not Enjoyed%": "0.009760187",
    "Band Name Worst%": "0.013488257",
    "Concert ID Best%": "0.14869164",
    "Concert ID Enjoyed%": "0.13135776",
    "Concert ID Not Enjoyed%": "0.14048058",
    "Concert ID Worst%": "0.19985951",
    "home_country": "0.07026541"
}
```

Figure B.2: Final feature importances