# Audio Engagement Challenge

Tiurina Tamila, Rubtsov Maksym
October 2025

# Outline

- Executive Summary
- Introduction
- Methodology
- Results
- Conclusion
- Appendix

# Executive Summary

## Summary of methodologies:

- Data Collection (CSV to Pandas)
- Data Wrangling (Cleaning, Formatting, Storing)
- Feature Engineering (ratios, bins, categorical encoding)
- Model Development (feature fusion, k-fold ,Optuna, LightGBM)
- Inference and reporting

## Summary of all results:

- EDA Results
- Model Performance
- Data-point & Key Findings

# Introduction

## Project background and context:

This work is prepared as part of the Winter 2025 Recruitment Challenge for the Data Science Club at PJATK. The challenge provides a realistic dataset emulating a common industry task: estimating user engagement with media content. The dataset contains metadata about audio episodes and information that may correlate with listening behavior. The core modeling objective is to predict the expected listening time of a user for a given audio episode.

The formulation directly mirrors applied machine learning use cases in media analytics — predicting engagement for recommendation, scheduling, and monetization strategies. Unlike toy problems, the dataset includes heterogeneous features (text, categorical, numeric, engineered).

Submissions are evaluated quantitatively using Root Mean Squared Error (RMSE), enforcing precise regression performance rather than qualitative judgement. The task is open-ended with respect to features and methodology, making it a suitable benchmark of end-to-end ML capability: data handling, feature construction, model selection, and evaluation under a standardized metric.

Section 1

# Methodology

# Methodology

## Executive Summary

This project takes a thorough approach to predicting the expected listening time of a user for a given audio episode.

It integrates data collection, preprocessing, exploratory analysis, interactive visualizations, and predictive modeling.

### Perform data wrangling

- The dataset was cleaned to handle missing values, standardize formats, and ensure consistency. Relevant features were extracted, and new features were engineered to improve the quality and informativeness of the data.

### Perform exploratory data analysis (EDA) using visualization

- Key patterns, distributions, and relationships among features were analysed to understand the dataset and inform modeling decisions.

### Perform predictive analysis using classification models

- Multiple regression models were built and evaluated. Hyperparameter optimisation was conducted to improve model performance. Model performance was assessed using RMSE to identify the most accurate model for predicting the target variable.

# Data Wrangling

Data wrangling is the process of preparing raw data for analysis by cleaning, restructuring, and organizing it into a useful format.

### Step 1: Data Cleaning

- Handle missing values in numerical columns by imputing with the median and creating indicator flags for imputed entries.
- Fill missing values in important metrics with a default value.
- Replace missing text entries with a placeholder string

### Step 2: Data Transformation

- Ensure numeric and categorical columns have correct data types.
- Generate new features from existing columns, such as combining related metrics, calculating ratios, creating flags, or extracting numeric identifiers from text.
- Bin numerical values into categorical groups to simplify analysis

### Step 3: Data Integration

- Encode categorical features into numerical format for modeling.
- Combine numeric, categorical, and text-derived features into a single unified dataset

### Step 4: Data Validation

- Ensure no missing values remain after imputation and feature engineering.
- Verify consistent data types and correct feature creation.
- Confirm proper alignment and consistency of features for model input.

# EDA with Data Visualization

Exploratory Data Analysis (EDA) focuses on visually examining and summarizing the main features of a dataset. Its purpose is to understand data distributions, uncover trends, and highlight relationships among variables.

## Tools used:

- Autoviz Analysis
- Scatter plots
- UMAP
- T-SNE
- Pandas Profiling

Section 2

# Insights drawn from EDA

# AutoViz Analysis

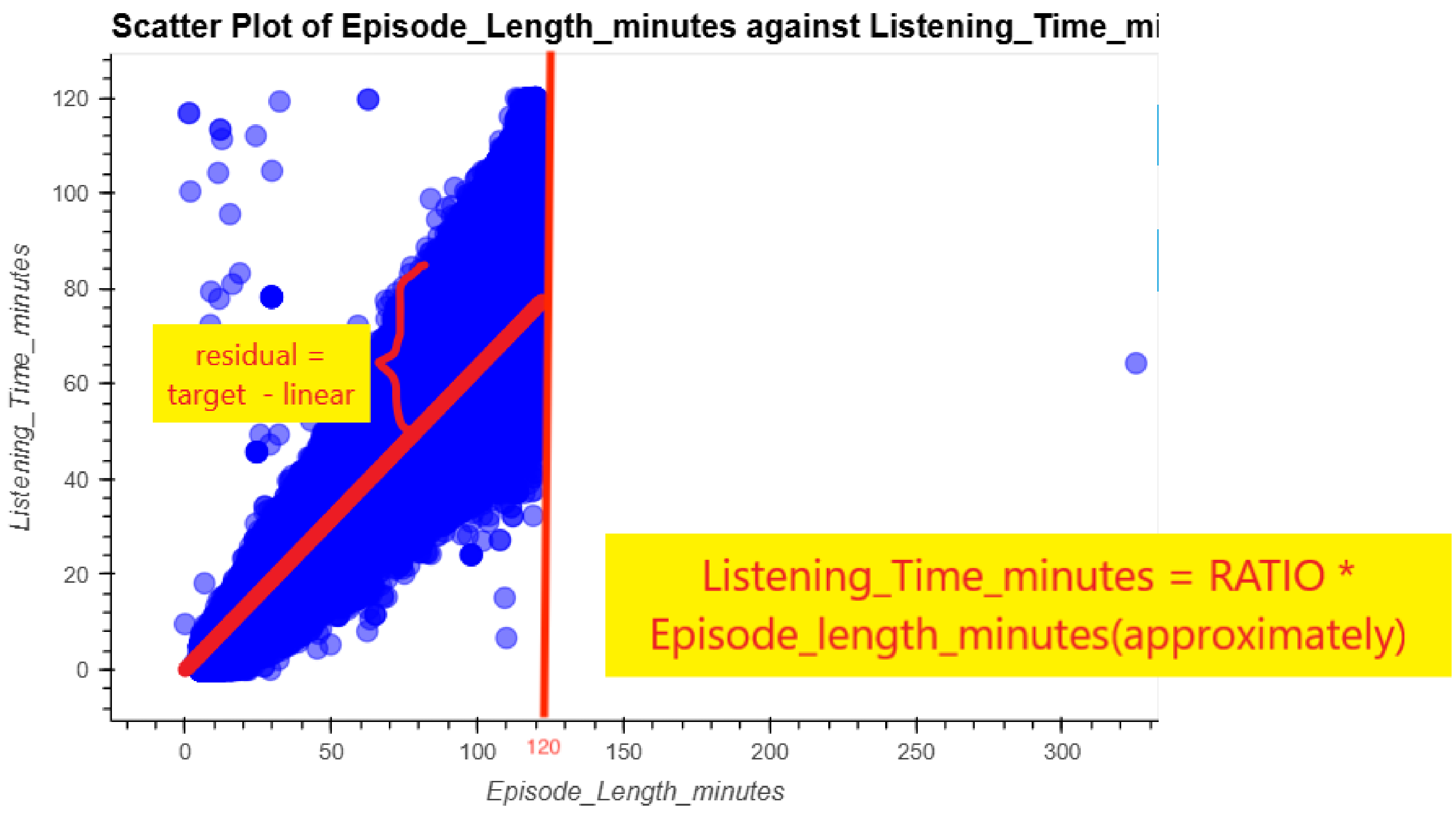
From the AutoViz analysis was observed significant problem connecting to the large amounts of missing data in two columns. Guest_Popularity_percentage is missing 19.5% of its values, and Episode_Length_minutes is missing 11.6%.

| | | | | | | |
|---|---|---|---|---|---|---|
| **Episode_Length_minutes** | float64 | 11.612400 | NA | 0.000000 | 325.240000 | 87093 missing values. Impute them with mean, median, mode, or a constant value such as 123., Column has 1 outliers greater than upper bound (181.58) or lower than lower bound(-51.78). Cap them or remove them. |
| **Guest_Popularity_percentage** | float64 | 19.470667 | NA | 0.000000 | 119.910000 | 146030 missing values. Impute them with mean, median, mode, or a constant value such as 123. |

```
################ Regression problem ####################
To fix these data quality issues in the dataset, import FixDQ from autoviz...
    All variables classified into correct types.
```

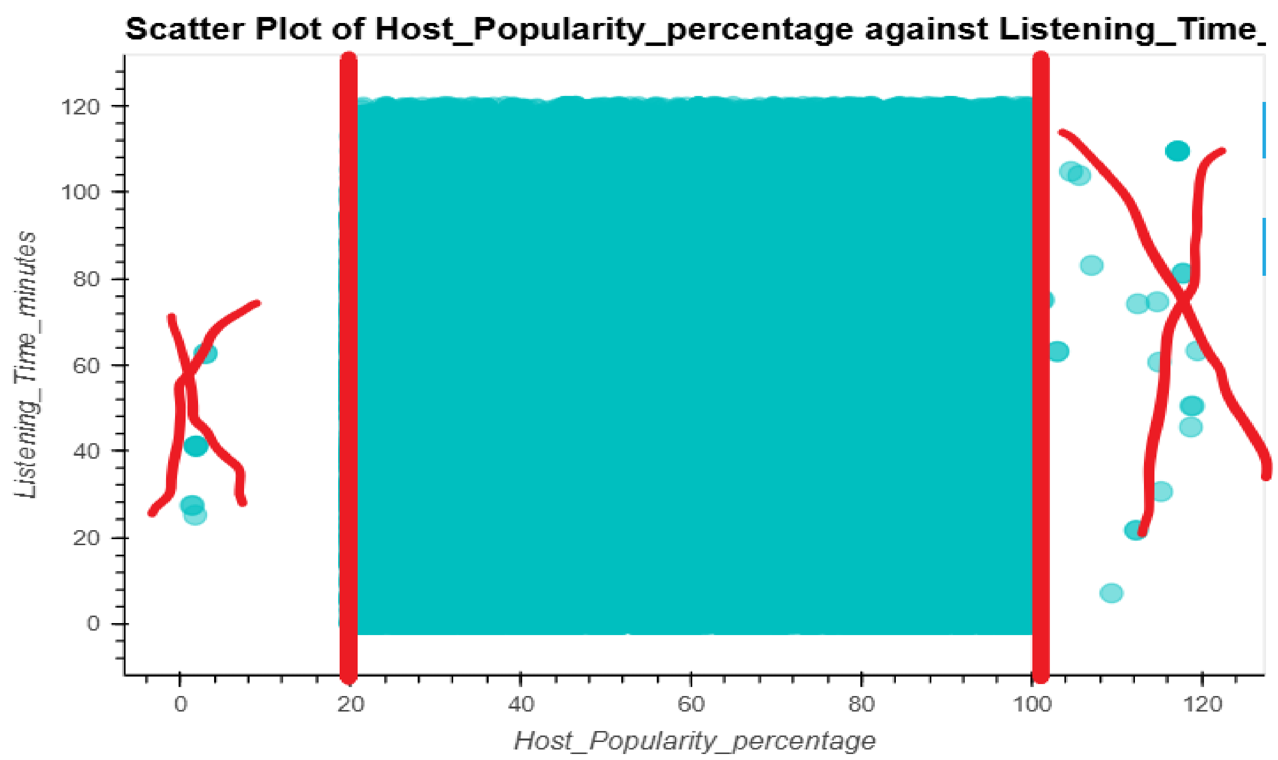
| | Data Type | Missing Values% | Unique Values% | Minimum Value | Maximum Value | DQ Issue |
|---|---|---|---|---|---|---|
| **Podcast_Name** | object | 0.000000 | 0 | | | No issue |
| **Episode_Title** | object | 0.000000 | 0 | | | 52 rare categories: Too many to list. Group them into a single category or drop the categories. |
| **Episode_Length_minutes** | float64 | 11.612400 | NA | 0.000000 | 325.240000 | 87093 missing values. Impute them with mean, median, mode, or a constant value such as 123., Column has 1 outliers greater than upper bound (181.58) or lower than lower bound(-51.78). Cap them or remove them. |
| **Genre** | object | 0.000000 | 0 | | | No issue |
| **Host_Popularity_percentage** | float64 | 0.000000 | NA | 1.300000 | 119.460000 | No issue |
| **Publication_Day** | object | 0.000000 | 0 | | | No issue |
| **Publication_Time** | object | 0.000000 | 0 | | | No issue |
| **Guest_Popularity_percentage** | float64 | 19.470667 | NA | 0.000000 | 119.910000 | 146030 missing values. Impute them with mean, median, mode, or a constant value such as 123. |
| **Number_of_Ads** | float64 | 0.000133 | NA | 0.000000 | 103.910000 | 1 missing values. Impute them with mean, median, mode, or a constant value such as 123., Column has 9 outliers greater than upper bound (5.00) or lower than lower bound(-3.00). Cap them or remove them. |
| **Episode_Sentiment** | object | 0.000000 | 0 | | | No issue |
| **Listening_Time_minutes** | float64 | 0.000000 | 5 | 0.000000 | 119.970000 | Target column |

# Episode_Length_minutes vs. Listening_Time_minutes



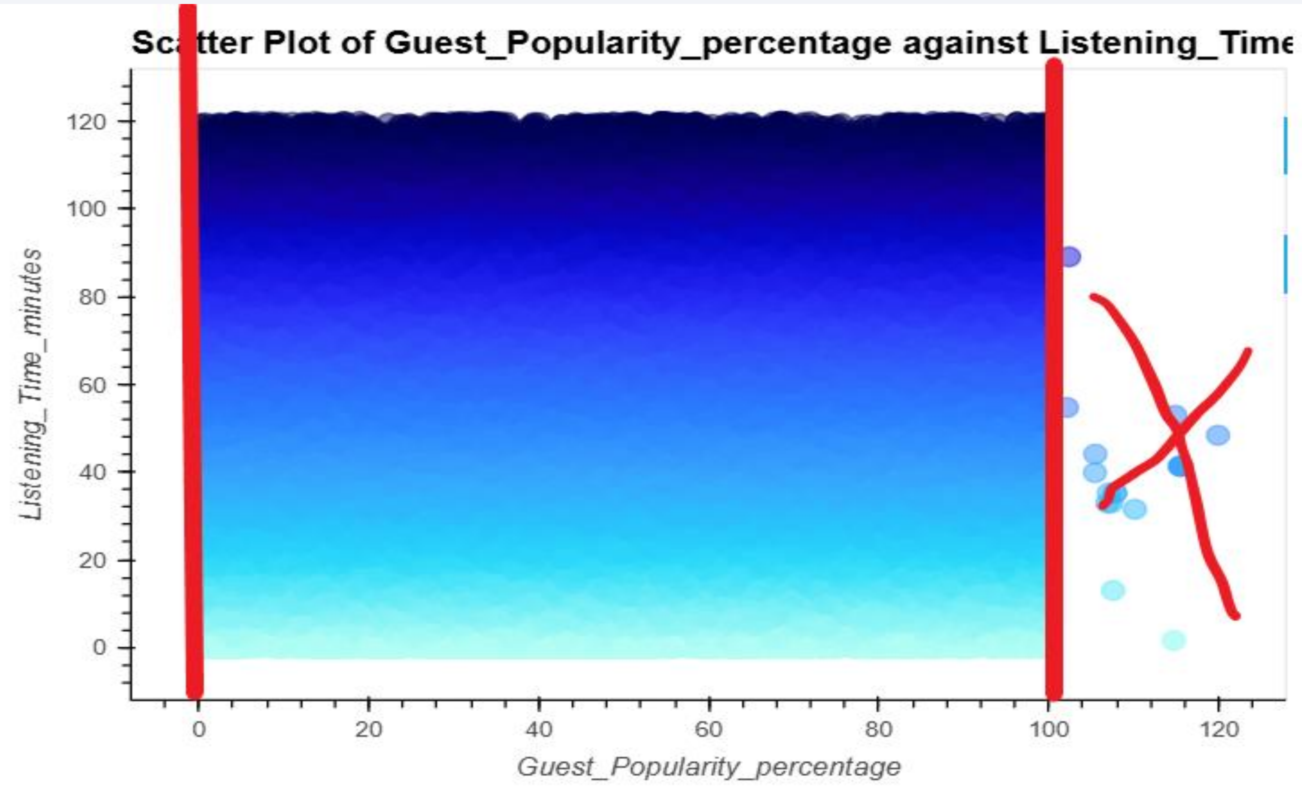

From the graph above we estimated that the relationship between Episode_Length_minutes and target Listening_Time_minutes is approximately Listening_Time_minutes = 0.728 x Episode_Length_minutes which means that people watch 72.8% of podcasts.

# Host_Popularity_percentage vs. Listening_Time_minutes



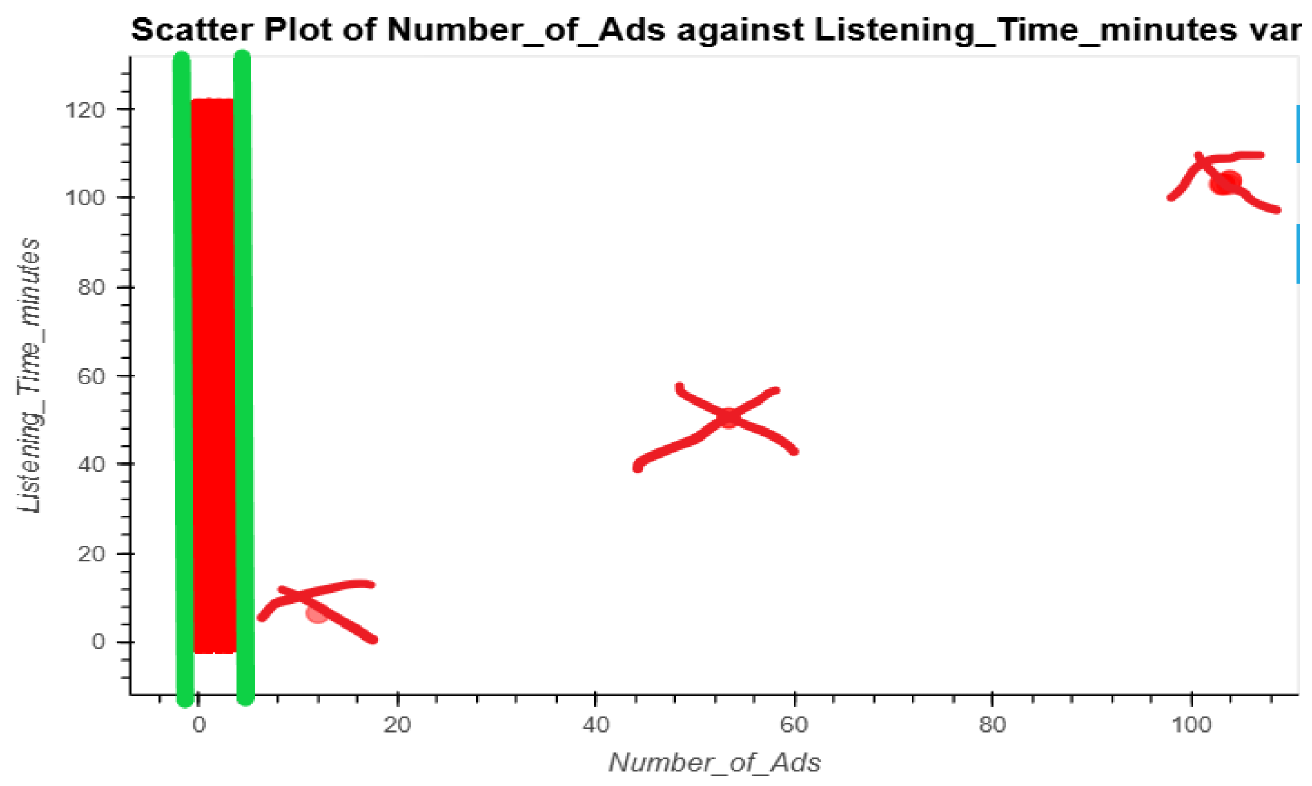

This plot shows outliers between Host_Popularity_percentage and Listening_Time_minutes, which will be capped to min 20 and max 100 during the data cleaning.

# Host_Popularity_percentage vs. Listening_Time_minutes



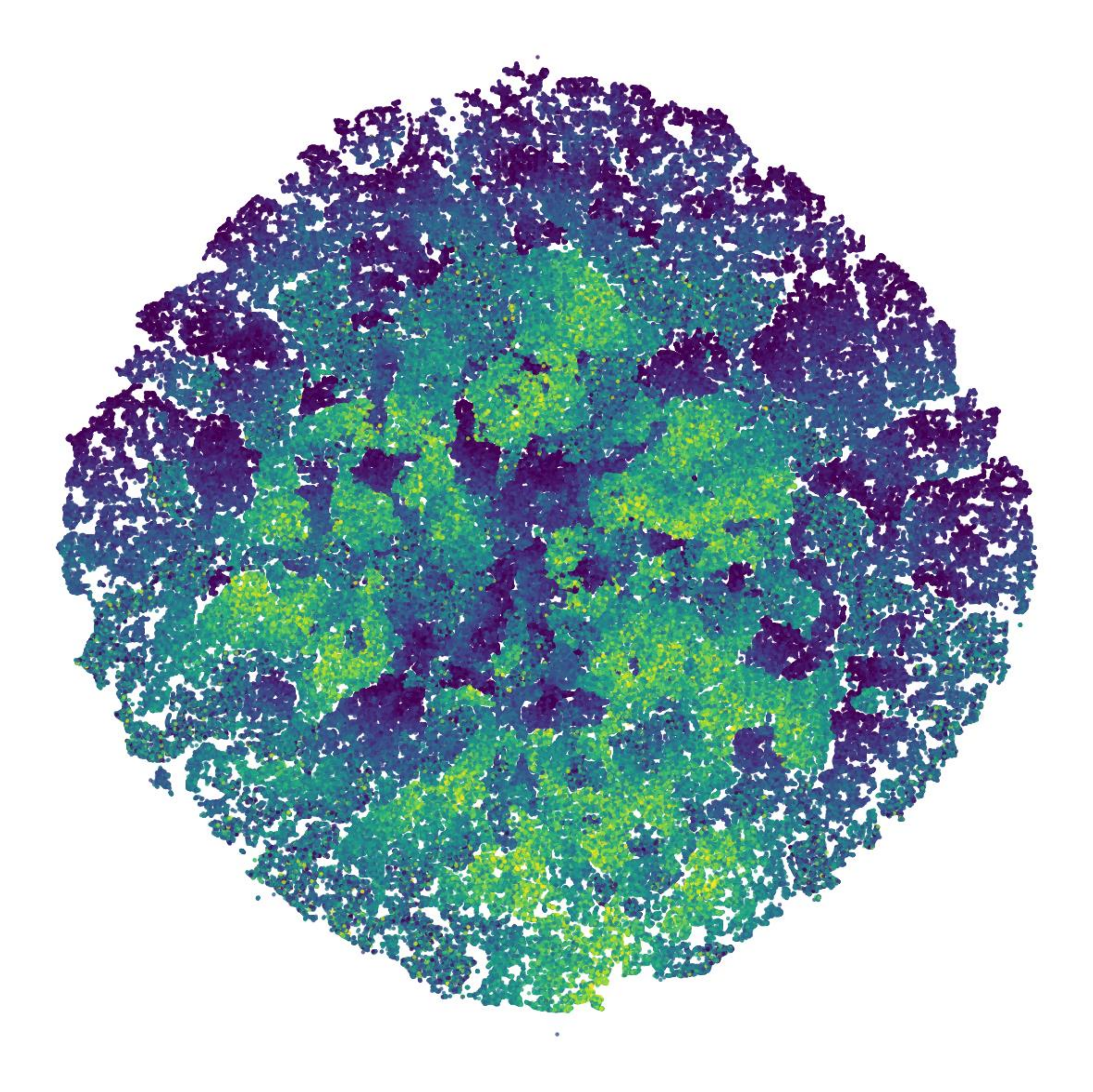

This scatter plot shows no correlation between Guest_Popularity_percentage (from 0 to 100) and Listening_Time_minutes, although there are a few distinct outliers visible where the guest popularity value exceeds 100.

# Host_Popularity_percentage vs. Listening_Time_minutes




This plot shows no clear correlation, as the vast majority of data is clustered at a very low Number_of_Ads (around 0–5) across the entire 0–120 minute listening range, with a few separate outlier groups (marked in red) appearing at much higher ad counts.

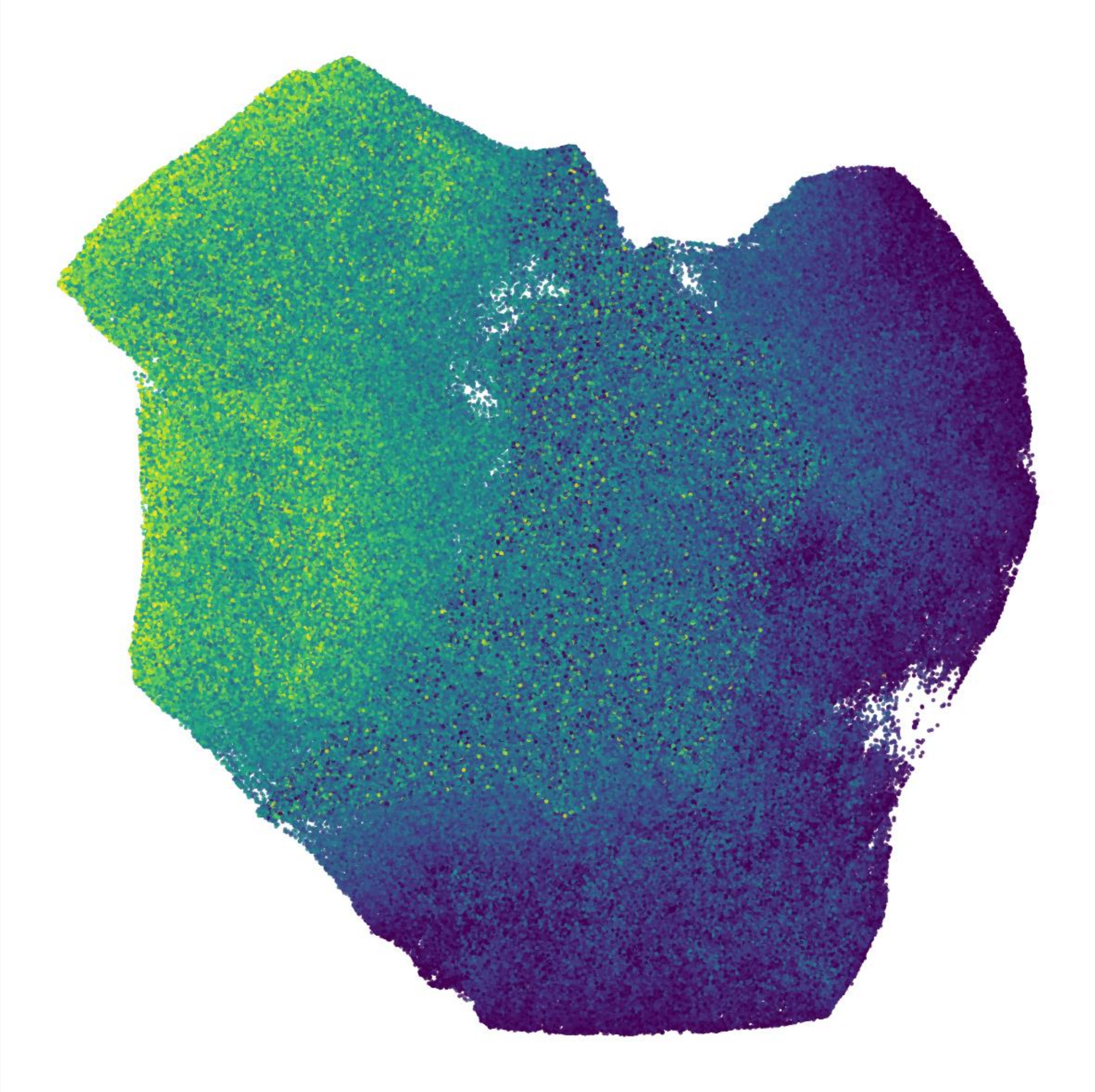
# t-SNE



The plot shows one large, continuous cloud of data points. There are no separate, well-defined clusters, which suggests the dataset is relatively homogeneous and doesn't naturally break into distinct, separate groups.

Despite the lack of large clusters, there is clear local structure. The colors, which represent the value of Listening_Time_minutes , are not randomly scattered. Points with similar colors are grouped together in "patches" or "islands."

# Umap



The plot shows that all the data points form one large, connected "landmass" or shape. This strongly reinforces the finding from the t-SNE plot: there are no distinct, separate clusters in the data. The dataset appears to be one continuous, homogeneous group.

There is a very clear and smooth color gradient that flows across the entire structure. It transitions seamlessly from dark purple (bottom right) through teal (center) to bright yellow (top left). This indicates the Listening_Time_minutes is continuous and changes progressively across the dataset.

# Two high-cardinality categorical features

## Episode_Title

Text

| | |
|---|---|
| **Distinct** | 100 |
| **Distinct (%)** | < 0.1% |
| **Missing** | 0 |
| **Missing (%)** | 0.0% |
| **Memory size** | 5.7 MiB |


The first one is Episode_Title that has 100 categories.

# Two high-cardinality categorical features

## Podcast_Name

Text

| | |
|---|---|
| **Distinct** | 48 |
| **Distinct (%)** | < 0.1% |
| **Missing** | 0 |
| **Missing (%)** | 0.0% |
| **Memory size** | 5.7 MiB |


The second is Podcast_Name which has 48 categories

# Other categorical features (have small number of categories)

## Publication_Time

Text

| | |
|---|---|
| **Distinct** | 4 |
| **Distinct (%)** | < 0.1% |
| **Missing** | 0 |
| **Missing (%)** | 0.0% |
| **Memory size** | 5.7 MiB |

## Episode_Sentiment

Text

| | |
|---|---|
| **Distinct** | 3 |
| **Distinct (%)** | < 0.1% |
| **Missing** | 0 |
| **Missing (%)** | 0.0% |
| **Memory size** | 5.7 MiB |

## Genre

Text

| | |
|---|---|
| **Distinct** | 10 |
| **Distinct (%)** | < 0.1% |
| **Missing** | 0 |
| **Missing (%)** | 0.0% |
| **Memory size** | 5.7 MiB |

## Publication_Day

Text

| | |
|---|---|
| **Distinct** | 7 |
| **Distinct (%)** | < 0.1% |
| **Missing** | 0 |
| **Missing (%)** | 0.0% |
| **Memory size** | 5.7 MiB |

# No grouped categories

Overview | Words | Characters

| Value | Count | Frequency (%) |
|---|---|---|
| tech | 42396 | 2.8% |
| matters | 42034 | 2.7% |
| business | 36492 | 2.4% |
| sports | 36244 | 2.4% |
| crime | 33747 | 2.2% |
| digest | 29562 | 1.9% |
| | 28325 | 1.8% |
| living | 26884 | 1.8% |
| talks | 22847 | 1.5% |
| news | 22818 | 1.5% |
| Other values (78) | 1210436 | 79.0% |

Overview | Words | Characters

| Value | Count | Frequency (%) |
|---|---|---|
| neutral | 251291 | 33.5% |
| negative | 250116 | 33.3% |
| positive | 248593 | 33.1% |

Based on the Frequency of each category among all categorical features it is noticable that the data between these categories was splitted equally, therefore no rare categories are present in the data set as well as none of them were grouped.

Section 3

# Feature Engineering

# Basic Processing - 1

```
missing_values_columns = ["Episode_Length_minutes", "Number_of_Ads"]
for col in missing_values_columns:
    train[f'{col}_IS_IMPUTED'] = train[col].isnull().astype(int)
    test[f'{col}_IS_IMPUTED'] = test[col].isnull().astype(int)
    med = train[col].median()
    train[col] = train[col].fillna(med)
    test[col]  = test[col].fillna(med)


train['Guest_Popularity_percentage'] = train['Guest_Popularity_percentage'].fillna(0.0)
test['Guest_Popularity_percentage'] = test['Guest_Popularity_percentage'].fillna(0.0)
train['Host_Popularity_percentage'] = train['Host_Popularity_percentage'].fillna(0.0)
test['Host_Popularity_percentage']  = test['Host_Popularity_percentage'].fillna(0.0)


train['Episode_Title'] = train['Episode_Title'].fillna('missing').astype(str)
test['Episode_Title']  = test['Episode_Title'].fillna('missing').astype(str)
```

For numerical features Episode_Length_minutes and Number_of_Ads, we fill missing values with the median (a good choice as it's robust to outliers). Crucially, we also add a binary flag column Episode_Length_minutes_IS_IMPUTED to let the model know which rows were imputed.

For popularity features, NaN is filled with 0.0. This is a logical assumption, implying that a missing popularity score is equivalent to zero popularity or no guest.

Episode_Title NaN values are filled with the string "missing". This ensures the TF-IDF vectorizer has a value to process, and "missing" will be treated as its own unique token.

# Derived Features - 2

```python
eps = 1e-6
for df in (train, test):
    df['Popularity_Combined'] = (df.get( key: 'Host_Popularity_percentage',  default: 0.0) +
    df.get( key: 'Guest_Popularity_percentage',  default: 0.0))
    df['Ads_per_Minute'] = df['Number_of_Ads'] / (df['Episode_Length_minutes'] + eps)
    df['Ads_per_Minute'] = df['Ads_per_Minute'].fillna(0.0)
    df['Len_div_ads'] = df['Episode_Length_minutes'] / (df['Number_of_Ads'] + 1.0)
    df['Has_Guest'] = (df.get( key: 'Guest_Popularity_percentage',  default: 0.0) > 0).astype(int)
```

Popularity_Combined: A simple interaction feature that sums guest and host popularity. This provides a single, unified "popularity" signal for the episode.
Ads_per_Minute: Normalizes ad count by episode length. eps (a very small number) is added to prevent division by zero.
Len_div_ads: The inverse of density, calculating "minutes of content per ad." + 1.0 is used to avoid division by zero for episodes with 0 ads.
A binary feature that discretizes Guest_Popularity_percentage. It simplifies the signal from "how popular is the guest?" to "is there a guest?" (assuming any popularity score > 0 means a guest is present).

# Derived Features - 2

```python
def extract_episode_number_safe(title):
    if pd.isna(title):
        return 0
    m = re.search(r'(\d{1,5})\b(?!.*\d)', str(title))
    return int(m.group(1)) if m else 0

train['Episode_Number'] = train['Episode_Title'].apply(extract_episode_number_safe)
test['Episode_Number']  = test['Episode_Title'].apply(extract_episode_number_safe)
```

Episode_Number: A smart regex is used to extract what is likely the episode number. The regex r'(\d{1,5})\b(?!.*\d)' specifically looks for the last sequence of 1-5 digits in the title, which is a robust way to avoid capturing other numbers (like "Part 2" in a title).

Ads_Groups (Binning): Number_of_Ads is binned into categorical groups: '0-1', '1-2', and '2+'. This is a powerful technique that captures potential non-linear effects. For example, the difference in listening time between 0 and 1 ad might be huge, while the difference between 10 and 11 ads might be negligible.

# Category Processing - 3

This section converts categorical string features into integers so they can be combined into a single numerical matrix with the TF-IDF features.

```
from sklearn.preprocessing import LabelEncoder
categorical_small = ['Genre','Publication_Day','Publication_Time','Episode_Sentiment','Ads_Groups']
for col in categorical_small:
    if col in train.columns:
        le = LabelEncoder()
        all_values = pd.concat( objs: [train[col].astype(str), test[col].astype(str)], axis=0)
        le.fit(all_values)
        train[col + '_le'] = le.transform(train[col].astype(str))
        test[col + '_le']  = le.transform(test[col].astype(str))
```

We are using LabelEncoder for all small categorical columns (such as Genre, Publication Day, and the new Ads Publication_Day, and the new Ads_Groups). It assigns a unique integer to each category in the way that:  '0-1' -> 0, '1-2' -> 1, '2+' -> 2.

# TF-IDF - 4

Models like LightGBM only work with numerical data. The Episode_Title column is text, but it contains valuable predictive information. Therefore, we use TF-IDF as a method to translate that text. It converts the titles into a set of numerical features, allowing the model to "read" the words and use the important information from the title to make predictions.

```
tfidf = TfidfVectorizer(max_features=3000, ngram_range=(1,2), min_df=3)
all_titles = pd.concat( objs: [train['Episode_Title'], test['Episode_Title']], axis=0)
tfidf.fit(all_titles)
X_tfidf_train = tfidf.transform(train['Episode_Title'])
X_tfidf_test  = tfidf.transform(test['Episode_Title'])
```

- TfidfVectorizer: This creates features based on word importance.
- max_features=3000: It limits the vocabulary to the 3,000 most frequent/important words.
- ngram_range=(1,2): It considers both single words (1-grams, e.g., "science") and two-word phrases (2-grams, e.g., "data science"), which capture more context.
- min_df=3: It ignores words that appear in fewer than 3 titles, filtering out noise and typos.

# Combining Features - 5

Finally, all the engineered features are combined into a single sparse matrix, which is highly efficient for LightGBM.

```
X_tab_train_sparse = sparse.csr_matrix(X_tab_train.values)
X_tab_test_sparse  = sparse.csr_matrix(X_tab_test.values)

X_train_full = sparse.hstack([X_tab_train_sparse, X_tfidf_train]).tocsr()
X_test_full  = sparse.hstack([X_tab_test_sparse, X_tfidf_test]).tocsr()
```

All the numerical and label-encoded features (Sections 1-3) are put into X_tab_train_sparse. The text features (Section 4) are in X_tfidf_train. sparse.hstack stacks these two matrices side-by-side to create one wide matrix (X_train_full) that contains all the information.

Section 4

# Model Performance

# Models tested

1. CatBoostRegression
2. LightGBMRegression – gave the lowest RMSE
3. XGBoostRegression

# LightGBM Overview

The model was configured as a LGBMRegressor with 10,000 boosting iterations, a learning rate of 0.05. Additional parameters, such as num_leaves, bagging_fraction, bagging_freq, feature_fraction, min_child_samples, and regularization terms (lambda_l1, lambda_l2), were tuned with Bayesian optimization. The model was trained using 20-fold cross-validation, ensuring that every data point contributed to both training and validation. Early stopping was applied with a patience of 100 rounds based on the RMSE to terminate training once no further improvement in performance was observed.

# K-fold cross-validation

```
bins = int(np.floor(1 + np.log2(X_train_full.shape[0])))
y_binned = pd.cut(y, bins=bins, labels=False)
kf = KFold(n_splits=20, shuffle=True, random_state=0)

rmse_scores = []
test_pred = np.zeros(X_test_full.shape[0])

for fold, (train_idx, val_idx) in enumerate(kf.split(X_train_full, y_binned)):
    X_train, X_val = X_train_full[train_idx], X_train_full[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]
```

The dataset was randomly shuffled and divided into 20 equal parts. In each iteration, 19 folds were used for training and 1 for validation.
The `KFold` function generates the indices for training and validation samples (`train_idx`, `val_idx`), which are then used to split the feature matrix (`X_train_full`) and target values (`y`).
This process is repeated 20 times so that every data point is used for validation exactly once.

# Model's hyperparameters produced with optimization

```
model = lgb.LGBMRegressor(
        random_state=RANDOM_STATE,
        n_estimators=10000,
        learning_rate=0.05,
        num_leaves=198,
        n_jobs=-1,
        bagging_fraction= 0.5550229971836489,
        bagging_freq=7,
        feature_fraction= 0.9450630242019847,
        lambda_l1=0.002477866490667525,
        lambda_l2=2.3052514525674056e-06,
        min_child_samples=20,
)
```

# Model training

```
model.fit(
    X_train, y_train,
    eval_set=[(X_val, y_val)],
    eval_metric='rmse',
    callbacks=[lgb.early_stopping(stopping_rounds=100)]
)

preds_valid = model.predict(X_val)
rmse = np.sqrt(mean_squared_error(y_val, preds_valid))
rmse_scores.append(rmse)
print(f"Fold {fold + 1} RMSE: {rmse:.4f}")

# accumulate test predictions
test_pred += model.predict(X_test_full, num_iteration=model.best_iteration_) / kf.n_splits
```

The model is trained on the previously spllited training data `X_train` and `y_train`, while monitoring its performance on the validation set `X_val` and `y_val` using the RMSE metric. After training, predictions are made on the validation set, and the RMSE for the current fold is calculated and stored in `rmse_scores`. Finally, the model generates predictions on the test set `X_test_full` using the best iteration determined by early stopping. These test predictions are accumulated and averaged over all folds.

# Obstacles and solution

The main obstacle during development was running out of memory while executing large-scale code. This issue was resolved by leveraging RAPIDS and CUDA, which enabled GPU-accelerated computation, significantly reducing memory usage and speeding up processing.

# Conclusions

1. Feature engineering can have a greater impact on model performance than raw data alone. Creating derived features, grouping variables, or flagging missing values helps capture predictive relationships and non-linear effects.
2. There is no single best model for every problem; while LightGBM performs well on engineered features and sparse data, the optimal choice depends on the dataset, features, and task. Experimentation with different models is essential to find the most effective solution.

# Appendix

Github repo: https://github.com/maxrub04/Audio-Engagement-Challenge

Kaggle Challenge: https://www.kaggle.com/competitions/audio-engagement-challenge/overview

# Thank you!