# DA Project (Kaggle Contest)

**Shashank Satish Adsule - DA25M005**

20 November 2025

Table of Contents

## 1. Introduction

This Report contains the mythology and steps to solve Kaggle contest (2025)

Github repo: https://github.com/shashank-adsule/DA_project_contest

- Code Files:
  - "gen_embedding_v1.ipynb", "gen_embedding_v2.ipynb", "train_combine.ipynb"," test_combine.ipynb"," train_split.ipynb"," test_split.ipynb"
- Dataset:
  - Train_data.json,
  - Test_data.json
  - metric_name_embeddings.npy
  - defination_embedding.pkl
- Models:
  - Linear Regression
  - Ridge Regression
  - XGBRegressor
  - XGBRFRegressor
  - RandomForestRegressor
  - MLPRegressor
- Custom python Modules:
  - create_syntetic_data.py
  - metric_embeding_mapping.py

For navigating through code files in repo refer table below

| Sr no. | File Name | File Directory |
|---|---|---|
| 1 | *"train_data.json","test_data.json", "metric_name_embeddings.npy", "defination_embedding.pkl"* | .\dataset* |
| 2 | *"create_syntetic_data.py", "metric_embeding_mapping.py"* | .\code* |
| 3 | *Models pkl file* | .\assests\models* |
| 4 | *"gen_embedding_v1.ipynb", "gen_embedding_v2.ipynb",* | .\genrate_emb* |
| 5 | *"train_combine.ipynb"," test_combine.ipynb"," train_split.ipynb"," test_split.ipynb"* | .\final_code \version 1*, .\final_code \version 2* |
| 6 | *CSV outputs* | .\outputs* |
| 7 | *Report.pdf* | .\Report.pdf |

## 2. Dataset analysis

The dataset contains class definition as “metric_names and texts like user_prompt, response, system_prompt , each associated with a discrete score label. The natural score distribution is strongly skewed, with many samples Peaking around (9-10). Score of particular text set is similary between text set and metric definition which is in “metric_name_embedding.npy” file. It provides the mapping between the metric_name and it’s actual definition.

- Both train_data.json and test_data.json contains [“metric_name”,”user_prompt”,”response”,”system_prompt”] features.
- [“score”] is target/label for this dataset in train_data.json

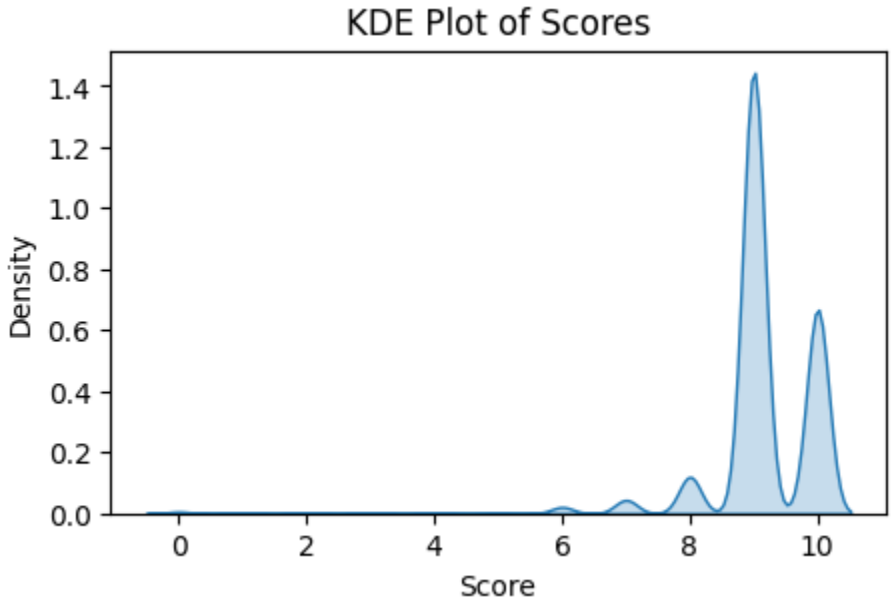


Fig [1]. Distribution of score in training data.

## 3. Data Loading and preprocessing

First before loading data for training and testing we have covert the metric_name definition file to useable mapping dict() which we can use for easy mapping in pandas dataframe. To so I have used “metric_embeding_mapping.py” for cheating a .pkl file for mapping.

Steps followed for Data Loading and preprocessing:

- Loaded training data from JSON format into a Pandas DataFrame.
- Applied a custom text preprocessing function to normalize prompts and responses:
    - converted to lowercase
    - removed punctuation and digits
    - collapsed multiple spaces
    - stripped whitespace
- Replaced all missing text fields with empty strings for consistency.

- Loaded a precomputed dictionary of metric embeddings and mapped each “metric_name” to its corresponding embedding.
- Converted the “score” column to float32 to ensure numerical efficiency.

Final dataset contains clean, standardized text fields and rich embedding features ready for model input.

## 4. Embedding Generation

To convert text to embedding 3 different sentence transformer models were used from “huggingface” repository [“google/embeddinggemma-300m”, “all-mpnet-base-v2”,”intfloat/e5-base-v2”].

A dedicated embedding pipeline (in gen_embedding_v1 and gen_embedding_v2 notebooks) is used to convert each metric name into a dense numerical vector.

Steps involved in embedding generation

- Load and Preprocess data as in steps given  above part.
- Text sets are first standardized through light text-cleaning (lower-casing, trimming, and normalization) to ensure consistent model input.
- A pretrained language model encodes each cleaned Text set into a fixed-dimension embedding that captures its semantic meaning and contextual relationships.
- These embeddings represent how similar or different various evaluation metrics are, providing an additional semantic signal during model training.
- All generated embeddings are stored in a np.ndarray  and serialized to disk using .parquet file extension for efficient reuse.
- During dataset construction, each metric_name and text sets are mapped to its corresponding embedding vector and stored in the new embeddings column.

These embedding vectors are then used as part of the model’s input features, enabling richer learning beyond raw text fields.

Each type of embedding is store in different format like for “gen_embeddding_v1.py” the all the text components are concatenated as single text then pass through the sentence transformer model. To create single embedding file for whole text data. For “gen_embeddding_v2.py” create separate embedding for each text set and save different. parquet file of each text column component.

The model which performs better on this data set was ”intfloat/e5-base-v2” which I have use for final evaluation

## 5. Data Augmentation

To improve the diversity and robustness of the training dataset, several augmentation strategies were applied to the original metric–text embedding pairs. First, a reduced subset of the dataset was selected, and three types of negative examples were generated from it:

- **shuffle-based negatives**, where text embeddings were randomly permuted to break alignment with metric embeddings;
- **noise-corrupted negatives**, created by adding Gaussian noise to text embeddings
- **metric-swap negatives**, where metric embeddings were randomly reassigned to mismatched text samples.

These negative samples were assigned low or random score values to represent weak or incorrect metric–text alignment.

To address the natural sparsity in mid-range score values (3–6), an additional set of mid-score synthetic samples was created using noise-corrupted embeddings. Furthermore, a positive augmentation was added by generating a separate group of synthetic high-score examples (scores 9–10), created through independent random permutations of metrics and text.

All augmented samples were combined with the original dataset to produce a large, diverse training set that captures both realistic and artificially constructed variations. This enriched dataset supports better generalization and helps the model learn stronger discriminative patterns across the entire score spectrum.

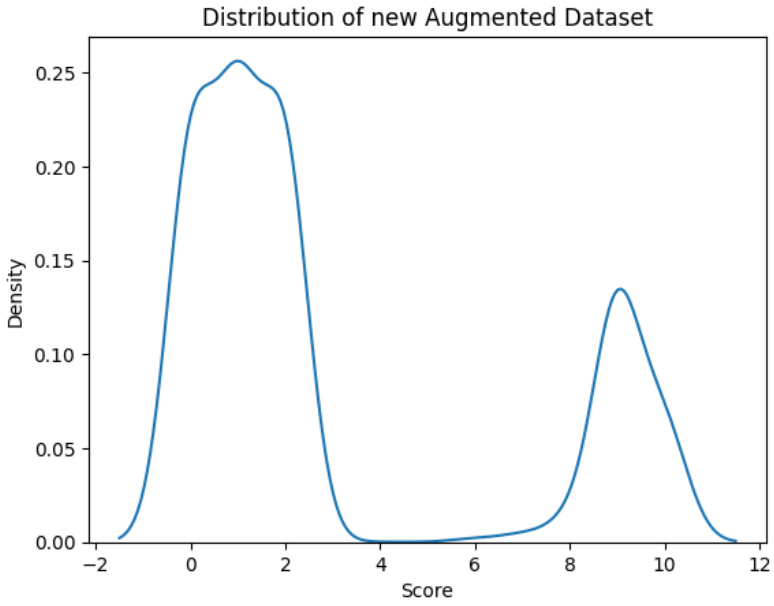


Fig [2]. Distribution of data after Augmentation

After doing Augmentation the size of new train set was around (20000-25000) rows and (3076) [embeddings vectors].

## 6. Feature Engineering

The feature engineering process combines multiple embedding-based similarity signals to create a rich numerical representation for supervised learning. First, the **cosine similarity** between each metric–text embedding pair is computed, providing a scalar measure of semantic alignment. Additional element-wise interactions are then generated, including the **absolute difference** between embeddings (capturing how far the two vectors deviate) and their **element-wise product** (highlighting shared activation patterns). These interaction features help the model capture subtle relationships between metric definitions and text responses.

To preserve the full contextual information, the original metric and text embeddings are also **concatenated** into a single vector. Finally, all features "raw embeddings", "abs-difference", "element-wise product", and "cosine similarity" are stacked to form the final input matrix.

This results in a comprehensive 3073-dimensional feature vector per sample, combining both linear and non-linear relationships. Such enriched representations help downstream models learn stronger discriminative patterns and improve prediction accuracy by leveraging multiple views of similarity and interaction between the metric and text embeddings.

## 7. Model Training

The model training pipeline is designed to compare a diverse set of regression algorithms, each model bringing a different inductive bias and learning capability to the score-prediction task. This modular setup enables systematic experimentation by selecting any model from the predefined list and training it under a unified workflow. By examining the performance of diverse models, the training setup helped me to identify the most suitable regression approach for the high-dimensional, engineered feature space constructed earlier.

The simplest models in the collection were "LinearRegression" and "Ridge Regression" serve as interpretability baselines and help assess whether the relationship between metric–text embedding features and score values is predominantly linear. LinearRegression provides an unconstrained least-squares fit, while Ridge introduces L2 regularization to control coefficient magnitude and reduce overfitting. Although computationally lightweight, these models generally struggle with the non-linearity present in semantic embedding features but remain useful as diagnostic tools and lower performance bounds.

A second group of models consists of tree-based ensemble methods, namely RandomForestRegressor, XGBRegressor, and XGBRFRegressor. These models capture complex non-linear interactions between features without requiring explicit feature scaling. XGBRegressor employs gradient boosting to iteratively refine predictions through additive tree updates, making it highly expressive and able to model subtle interactions. XGBRFRegressor combines Random Forest bagging with XGBoost's efficient histogram-based split algorithm, offering a robust and computationally efficient alternative. RandomForestRegressor, meanwhile, leverages bootstrap

sampling and deep decision trees to learn hierarchical decision patterns. These ensemble models are well-suited for the mixed interaction-based features generated earlier.

The final candidate, MLPRegressor, provides a neural-network‑based approach that can learn smooth, continuous relationships in high-dimensional spaces. The selected architecture uses two hidden layers (256 → 128), the Adam optimizer, early stopping, and a large batch size to stabilize training. Although neural networks require careful hyperparameter tuning, they can outperform classical models when sufficient data and well-structured features are available.

After selecting a model for experimentation, the training workflow fits the model to x_train, generates predictions for x_test, and calculates key evaluation metrics including MSE, $R^2$, and RMSE.

## Model Overview Table

| Model Name | Model Type | Hyperparameters | Strengths | Limitations |
|---|---|---|---|---|
| LinearRegression | Linear Model | — | Fast, simple, interpretable | Cannot model non-linear relationships |
| Ridge Regression | Linear + L2 Regularization | solver = sag; max_iter = 5000 | Reduces overfitting, stable gradients | Still linear; limited expressive capacity |
| XGBRegressor | Gradient Boosting Trees | n_estimators = 400; learning_rate = 0.05; max_depth = 8; subsample = 0.8; colsample_bytree = 0.8 | High accuracy, strong non-linear modeling | Slower training, sensitive to hyperparameters |
| XGBRFRegressor | Random Forest + XGBoost Hybrid | n_estimators = 300; max_depth = 8; subsample = 0.8; colsample_bytree = 0.8 | Robust, efficient, stable generalization | Less expressive than full gradient boosting |
| RandomForestRegressor | Bagged Decision Trees | n_estimators = 200; max_depth = 20; n_jobs = -1 | Good for non-linear data; no scaling required | May overfit; large memory usage |
| MLPRegressor | Neural Network | hidden_layer_sizes = (256,128); max_iter = 250; batch_size = 2048; solver = Adam; | Learns complex high-dimensional patterns | Needs scaling; tuning required; sensitive initialization |

| | | | | |
|---|---|---|---|---|
| | | early_stopping = True | | |

After complete Training the trained model were saved using "joblib" package to use later for test set. These models were saved in .pkl format.

## 8. Evaluation

To assess the performance of the trained regression model, three widely used evaluation metrics are computed:

- **Mean Squared Error (MSE)**: it measures the average squared difference between the predicted and true values, making it sensitive to large errors and useful for capturing overall prediction quality.
- **$R^2$ Score:** it evaluates how much of the variance in the target variable is explained by the model; values closer to 1 indicate a strong fit, while values near 0 suggest weak or no explanatory power.
- **Root Mean Squared Error (RMSE)**: provides an interpretable error measure in the same units as the score itself, capturing how far, on average, the model's outputs deviate from the true scores.

These metrics together provide a balanced understanding of both error magnitude and model fidelity.

Because the task involves predicting discrete score values between 0 and 10, an additional post-processing step is applied where predictions are rounded and clipped to remain within the valid range.

### Model Evaluation table

| Model | MSE (train) | $R^2$ score (train) | RMSE (train) | RMSE (test) |
|---|---|---|---|---|
| LinearRegression | 11.9923 | 0.0797 | 3.4765 | 4.352 |
| Ridge Regression | 11.9683 | 0.0815 | 3.4734 | 4.332 |
| XGBRegressor | 0.7477 | 0.1014 | 0.9018 | 3.343 |
| XGBRFRegressor | 0.7736 | 0.0653 | 0.9190 | 4.059 |
| RandomForestRegressor | 0.7865 | 0.0498 | 0.9284 | 4.626 |
| MLPRegressor | 7.4069 | 0.4315 | 2.7144 | 2.959 |

## 9. Conclusion

Among all tested models, neural and tree-based approaches performed noticeably better than linear baselines, with the **MLPRegressor** achieving the strongest generalization on the test set. Overall, the workflow proved effective for embedding-driven regression tasks, providing a solid foundation for future enhancements such as deeper neural architectures, improved augmentation strategies, and advanced embedding models.

Another important observation is the influence of **highly skewed target distributions**, which limited the models' ability to learn stable patterns across the full value range. Although neural and tree-based models handled nonlinearity better than linear baselines, the extreme imbalance caused them to focus disproportionately on dense regions of the distribution, resulting in weaker performance on rare or high-magnitude cases. This suggests that the models were not fully capturing the underlying relationships for the tail values.