1. Introduction

Our project is to look at IMDb ratings dataset from Kaggle in order to explore movie related data and ultimately see if we can make accurate predictions on the *weighted average IMDb rating.* The general breakdown of work between our team is:

- Preprocessing and EDA – Sahara Ensley
- Modeling – Josh Ting (me)
- GUI – Adam Kritz

2. Description

I will be in charge of the modeling portion of our project. My general approach is to start with a handful of regressor estimators and through a couple of judgement phases, will narrow down the model lineup to just one.

I also helped out on a part of the preprocessing to encode some of our categorical variables in *preprocessing_utils.py.*

- Worked on function to fit and transform actors, director, and writer frequencies on occurrence from training dataset. This is a proxy for quantifying the popularity of cast, writer, or director. If more than one person, we take the average of the frequencies. We also weighted the frequencies by the order of importance that the person played in the movie from the 'title_principals.csv' dataset.
- Genres: genre1, genre2, genre3 are binary encoded
    - binary_encoder_fit()
    - binary_encoder_transform()
    - binary_encoder()
- Production company: also frequency based but without order of importance weighting
    - fit_production_company_frequency()
    - transform_production_company_frequency()
- Title & Description: n words, ratio long words, ratio of vowels, ratio of interesting characters, ratio of capital letters
    - n_words()
    - ratio_long_words()
    - ratio_vowels()
    - ratio_interesting_characters()
    - ratio_capital_letters

Look at the appendix (8.2) for a log of my early phase work on this project for more details.

3. Modeling

Our objective in modeling was to create a model that fits relatively well to our dataset and would allow us to predict a satisfiable *weighted average IMDb rating*. Our high level strategy for the modeling phase was the following:

1. **Base Models**: Test out a handful of regression models using default SKlearn's parameters.
2. **Hyperparameter Tuning I**: Perform gridsearch validation on top 3 performers from step (1) using a select handful of hyperparameters.
3. **Hyperparameter Tuning II**: Select top 1 performing model from gridsearch validation for even further refined hyperparameter tuning.
4. **Retrain and Test**: Choose the best combination of hyperparameters for the best performing model from step (3) as our final model and retrain on the entire training and validation dataset then predict on test dataset.

Code Setup and Architecture

Our code for modeling is saved in ../*Final-Project-Group2/Code/* directory. There are three Python scripts, written in python3, where modeling is performed:

1. *models_helper.py*
2. *models.py*
3. *main.py*

1. *models_helper.py*

This Python file is the backbone to our modeling process. It holds the majority of objects and methods used for the modeling phase. It consists of the following 4 classes.

```
class Dataset:
class Model:
class ModelTuner(Model):
class Plotter:
```

The **Dataset** class represents our data and wraps our train, validation, and testing datasets into one object and has several methods to help with general data manipulation and returning specific objects related to our dataset. The table below shows a summary of these methods.

Method	Description
`__init__(self, train_df, test_df, val_df, random_seed, label)`	`Init method, set attributes, set numpy random seed`
`get_train_test_val_dfs(self)`	`Method to get a train, test, val as pd dataframes`
`split_features_target(self)`	`Method to split train, test, val data to features and target per dataset`
`data_as_arrays(self)`	`Method to convert data from pandas to np arrays/matrices for training`
`get_data_as_arrays(self)`	`Method to return data as np arrays`
`get_train_val_predefined_split(self)`	`Method to combine train and validation dataset into one and set predefined split for CV`

Figure X: Dataset class methods.

An instance of this object can be created in the following ways.

```
data = Dataset(df_train, df_test, df_val, random_seed, label='weighted_average_vote')
```

The **Model** class represents our model and it wraps much of the functionalities to train, calculate error, etc into one object. The table below shows a summary of these methods.

Method	Description
`__init__(self, random_seed, train_x=None, train_y=None, val_x=None, val_y=None, test_x=None, test_y=None, name=None, target_scaler=None)`	Init method for Model class
`save_model(self, filename, model)`	Method to save object model
`load_model(self, filename)`	Method to load object model
`evaluate(self, test_x=[])`	Method to evaluate model on test data
`train(self)`	Method to train model
`get_val_score(self)`	Method to get score for this estimator
`get_params(self)`	Method to get parameters for this estimator
`get_model_nickname(self)`	Method to return the model nickname assigned during object instantiation
`get_model(self)`	Method to return the model for object
`onstruct_model(self, model)`	Method to construct model (instantiate sklearn model)
`set_score(self, score)`	Method to set the score used for this model
`get_error_in_context(self, val_x=[], val_y=[])`	Method to get the error of model in context to target
`get_error(self, val_x=[], val_y=[])`	Method to get the error of model
`set_params_to_tune(self, params_dict)`	Method to set the paramaters to test for tuning the model
`set_params(self, params_dict)`	Method to set parameters to model

Figure X: Model class methods.

An instance of this object can be created in the following ways.

```
linear_model = Model(random_seed, train_X, train_Y, val_X, val_Y, test_X, test_Y,
'linear_sgd', target_scaler=ss_target)
```

The **ModelTuner(Model)** class is a child class of the **Model** class and represents an object to help tune the hyperparameters of the parent model object. The table below shows a summary of these methods.

Method	Description
`__init__(self, path, random_seed, train_x, train_y, test_x=None, test_y=None, name=None, target_scaler=None, ps=None, models_pipe=None, params=None)`	Init method for ModelTuner, child of Model

`make_directory(self)`	Helper method to make directory path if not created
`do_gridsearchcv(self, save_results=True, validation_curves=True, learning_curves=True)`	Method to perform GridSearchCV to tune hyperparameters of select models

Figure X: ModelTuner(Model) class methods.

An instance of this object can be created in the following ways.

```
gridsearchcv = ModelTuner(get_repo_root() + '/results/', random_seed, X_train_val,
Y_train_val, test_x=test_X, test_y=test_Y, name='gridsearchcv',
target_scaler=ss_target, ps=ps, models_pipe=models, params=param_grids)
```

The **Plotter** class represents a plotting object with functionality to help make the different plots we needed for modeling. The table below shows a summary of these methods.

Method	Description
`__init__(self, path, name, savename)`	Init method for Plotter
`model_comparison(self, score_dict, score, saveplot=True, show=False, alt=0)`	Method to plot a bar chart of models' avg scores
`learning_curves(self, lc_results, saveplot=True, show=False, alt=0)`	Method to plot learning curves from CV results
`validation_curves(self, vc_results, saveplot=True, show=False, alt=0)`	Method to plot validation curves from GridsearchCV results
`most_important_features(self, train_df, model, saveplot=True, show=False, alt=0)`	Method to perform analysis to get most important features from RandomForest and plot bar chart.
`s_random_bar(self, scores_dict, saveplot=True, show=False, alt=0)`	Method to plot a bar chart of models' avg scores

Figure X: Plotter class methods.

An instance of this object can be created in the following ways.

```
cv_plotter = Plotter(get_repo_root() + '/results/model_plots/', name='Tuning Model',
savename='tuning_model')
```

2. *models.py*

models_helper.py is then imported into *models.py* where the actual object instantiation and modeling is assembled.

```
from models_helper import Dataset, Model, ModelTuner, Plotter
```

The entire modeling is structured under a wrapper function in order to easily port into *main.py* later on.

```
run_modeling_wrapper(df_train, df_test, df_val, ss_target, df_test_untouched,
random_seed = 33, run_base_estimators = False, run_model_tuning = False,
fast_gridsearch = True, save_model = False, demo = False)
```

3. *main.py*

models.py is then imported into our main script *main.py* where it is integrated with the preprocessing and GUI phases of the project.

```
import models as mdl
```

The modeling wrapper has a few boolean parameters that a user can set in order to skip some parts of modeling that may take a while to execute.

```
mdl.run_modeling_wrapper(df_train, df_test, df_val, ss_target, df_test_untouched,
run_base_estimators = False, #Run base models comparison or not
run_model_tuning = False, #Run hyperparameter tuning and gridsearchcv or not
fast_gridsearch = False, #Skip most of gridsearchcv to run faster
save_model = True, #Save best model results or not
demo = True) #Demo = True, will skip all of modeling since we already have results
```

The following table outlines the different options to set in the modeling execution wrapper and what they mean for runtime.

Option	**Estimated Runtime***	**Settings**	**What it Does**
Nap Mode	2+ hours	`run_base_estimators = True` `run_model_tuning = True` `fast_gridsearch = False` `demo = False`	Run all processes from scratch: Base models, 2 phases of GridSearchCV, model evaluation, model selection, predict test data, outputs results.
Lunch Break Mode	10-15 mins	`run_base_estimators = False` `run_model_tuning = True` `fast_gridsearch = True` `demo = False`	Run with a smaller set of hyperparameters to tune with. It significantly compresses the time spent on hyperparameter tuning by only iterating over a smaller set of combinations.
Have a Coffee Mode	5 mins	`run_base_estimators = False` `run_model_tuning = False` `fast_gridsearch = False` `demo = False`	Skip model hyperparameter tuning entirely. It skips the hyperparameter tuning entirely and instead loads the already found best model pickled in the results directory
Demo Mode	0 mins	`run_base_estimators = False` `run_model_tuning = False` `fast_gridsearch = False` `demo = True`	Skips modeling entirely and is useful to focus on other portions of the project like the GUI.

Figure X: Options for choosing which parts of modeling to execute. *actual runtime depends on machine.

Model Scoring

To judge an estimator's performance, we will use the Mean Squared Error (MSE) of the predicted rating against the actual rating on the validation dataset. MSE is a measure of the error, or difference, between the predicted and expected output and can be described as,

$$MSE = 1n\ i=1n(yactual,i-ypredicted,i)2 \qquad (1)$$

where n is the number of observations in the validation dataset and i is the ith observation in n. The model with the lowest MSE will have the highest score.

Base Models

Our initial lineup of models to try are the default SKlearn's implementations of:

Model Type	SKlearn's Object	Pros	Cons
Linear Regression	SGDRegressor()[6]	Simple to implement and understand	Doesn't work well for non-linearly separable dataset
Random Forest	RandomForestRegressor()[7]	Less prone to overfitting	Large number of trees can be slow to use. Can't extrapolate on data outside range of trained data.
Gradient Boosting	GradientBoostingRegressor()[8]	Fits each subsequent tree on the residuals which can learn very well	Generally slower to fit than Random Forest and more prone to overfitting
Adaptive Boosting	AdaBoostRegressor()[9]	Uses many decision stumps and each stump gets a weighted vote	Not as robust to outliers as it tries to fit to every datapoint
K-Nearest Neighbors	KNeighborsRegresor()[10]	Simple to understand and low number of hyperparameters	Not as efficient as dataset grows

Figure X: Base SKlearn models to try.

We have detailed a few pros and cons for each model that we will have to consider in our model selection, understanding that these pros and cons are not exhaustive and can be relative depending on which models you compare to. Additionally, the next table outlines the details on each model's optimization method and loss function.

Model Type	SKlearn's Object	Optimization Method	Loss Function
Linear Regression	SGDRegressor()[6]	Stochastic Gradient Descent	Mean Squared Error
Random Forest	RandomForestRegressor()[7]	Build tree to split data	Mean Squared Error
Gradient Boosting	GradientBoostingRegressor()[8]	Build weak trees to split data and use gradient descent to fit residuals	Mean Squared Error
Adaptive Boosting	AdaBoostRegressor()[9]	Build weak trees to split data and samples with high errors are weighted higher for use in next tree	Mean Squared Error
K-Nearest Neighbors	KNeighborsRegresor()[10]	Calculate the minimum distance and take the mean for K number of neighbors	Mean Squared Error

Figure X: Summary of optimization method and loss function for each model.

The results from the initial training round found that the Random Forest and Gradient Boosting models were the top performers with the KNN in third as shown in the figure below.

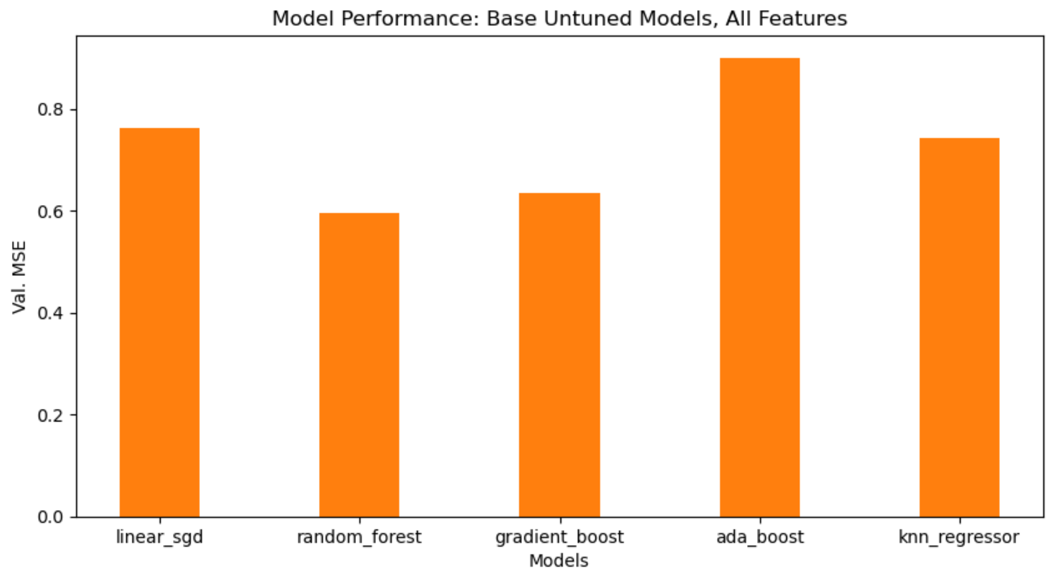


Figure X: Random Forest, Gradient Boosting, and KNN are top 3 performers.

Hyperparameter Tuning I

Hyperparameters are model parameters that are set prior to training and are not, usually, learned through training while parameters in typically machine learning lingo represents model parameters that are learned during training such as weights and biases. The following table shows the hyperparameters that we will tweak during our tuning phases.

Model Type	Hyperparameter	Hyperparameter Meaning	Dtype and Default
Random Forest	n_estimators	The number of trees in a forest.	int, default = 100
	min_sampes_split	Minimum samples required to split an internal node.	int or float, default = 2
	min_samples_leaf	Minimum samples to be at a leaf node.	int or float, default = 1
	max_features	Number of features to consider for each tree.	{'auto', 'sqrt', 'log2'} or int or float, default = 'auto
	max_depth	Max depth allowed for each tree	int, default = None
Gradient Boosting	learning_rate	Shrinks the contribution of each tree.	float, default = 0.1
	n_estimators	The number of boosting stages to use.	int, default = 100
	min_samples_split	Minimum samples required to split an internal node.	int or float, default = 2
K-Nearest Neighbors	n_neighbors	Number of neighbors to use.	int, default = 5
	p	p = 1 (Manhattan Distance), p = 2 (Euclidean Distance)	int, default = 2

Figure X: Hyperparameter definitions. Source: SKlearn model documentations.

In our first hyperparameter tuning phase, we selected our top 3 performing estimators and adjusted a few hyperparameters to see if we can improve upon their base scores. For this phase, we used SKlearn's *GridSearchCV()*[11] to iterate through each combination of hyperparameters for each model. We will perform a K-Fold cross validation phase after training on each combination where k=1 and the validation set used will be our predefined validation dataset.

Each model's hyperparameter grids for this phase and their best performing hyperparameters are shown in the table below.

Model Type	**Hyperparameter Grids**	**Best Performing Hyperparameters**

Random Forest	n_estimators: [100, 200, 400]
	
	min_sampes_split: [2, 4, 8, 16]
		n_estimators = 400
		
		min_sampes_split = 2

Gradient Boosting	learning_rate: [0.01, 0.1]
	
	n_estimators: [100, 200, 400]
	
	min_samples_split: [2, 4, 8, 16]
		learning_rate = 0.1
		
		n_estimators = 400
		
		min_samples_split = 2

K-Nearest Neighbors	n_neighbors: [3, 5, 7, 9]
	
	p: [1, 2]
		n_neighbors = 9
		
		p = 1

Figure X: Hyperparameter tuning I.

The following three figures show the validation curves for each of our models where we are plotting MSE on the y-axis and different hyperparameters that were tried on the x-axis. This is a useful visualization to see how a different hyperparameter value changes the model's MSE.

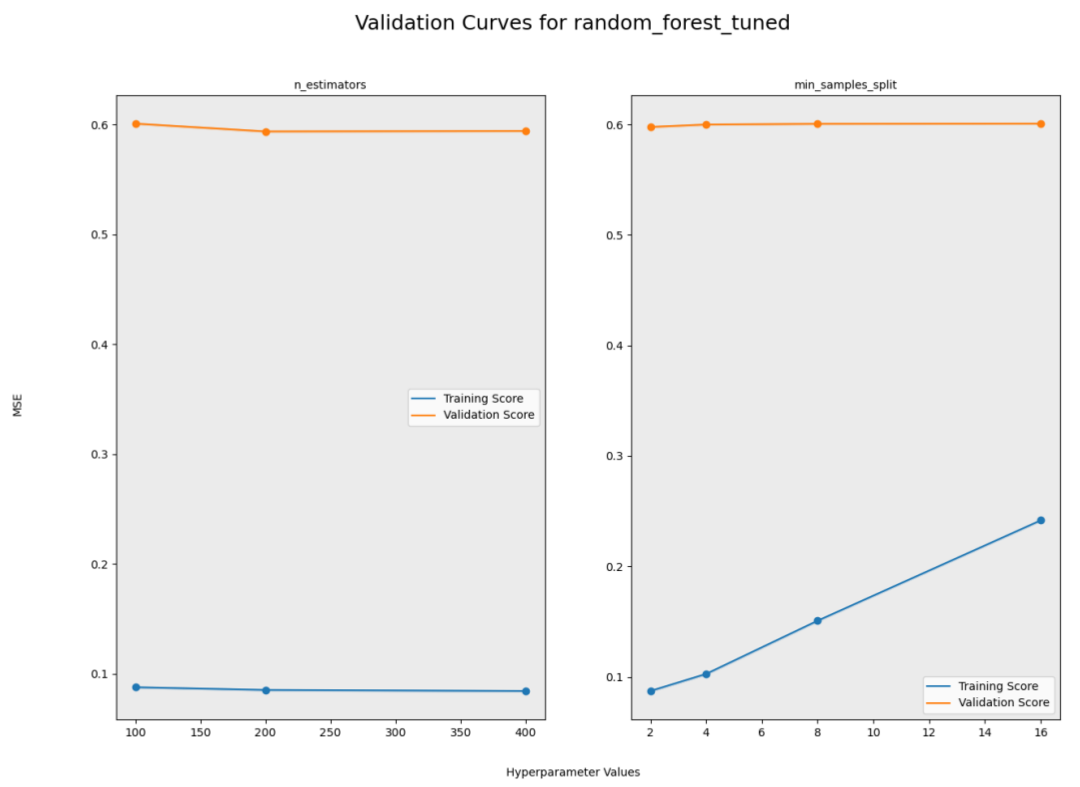


Figure X: Validation curves[12] for Random Forest model.

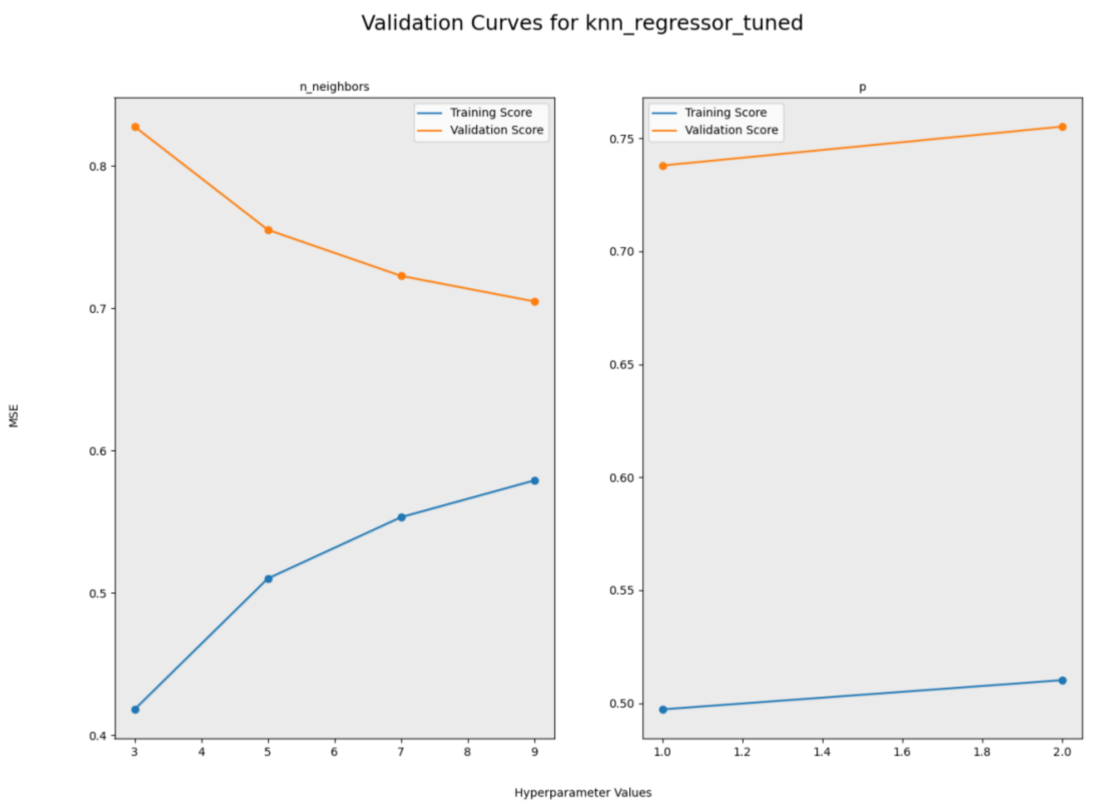


Figure X: Validation curves[12] for KNN model.

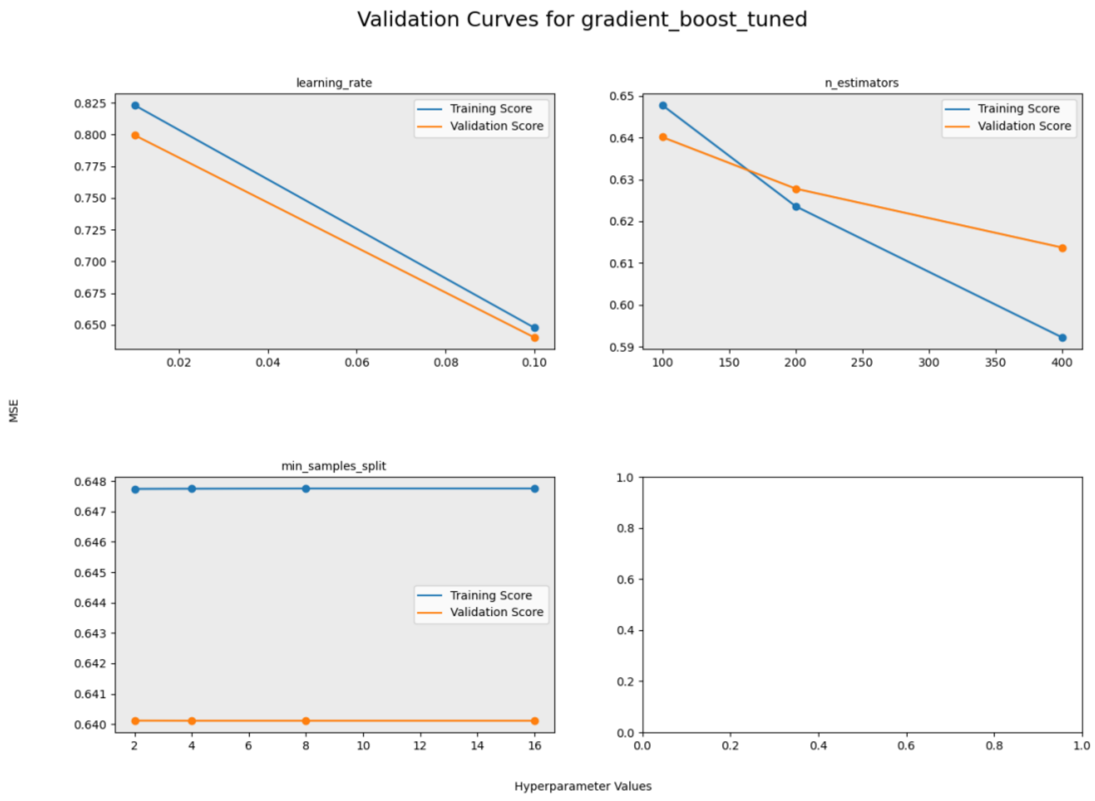


Figure X: Validation curves[12] for Gradient Boosting model.

It is interesting to note that generally across all three models, the training score is lower, quite considerable so in some cases, than the validation scores. This can be a sign of overfitting where our learned models have high variance and are not generalizing as well. We will try to add some regularization parameters in our next hyperparameter tuning phase for our best model.

Lastly, the following figure shows the best performing model after our first phase of hyperparameter tuning. Our best model is still the Random Forest. In the next phase, we will attempt to further refine the tuning for our Random Forest model and will keep overfitting in mind.

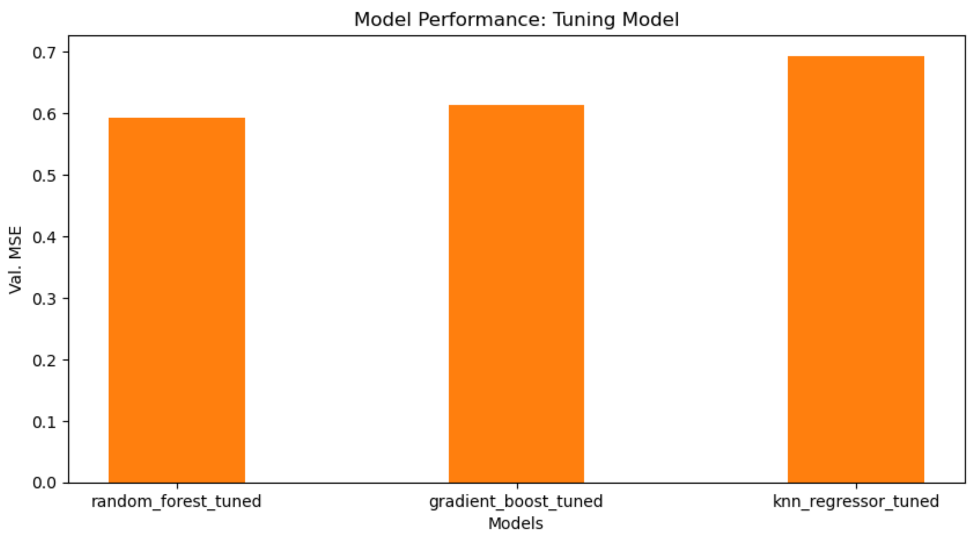


Figure X: MSE comparison of best hyperparameters for each model after tuning phase I.

Hyperparameter Tuning II

In this phase, we will now focus solely on the Random Forest model in an attempt to fine tune it more and reduce overfitting. Our new set of hyperparameters to use are:

Model Type	Hyperparameter Grids	Best Performing Hyperparameters

Random Forest	n_estimators: [200, 300, 400, 500]
	min_samples_leaf: [2, 4, 8, 12]
	max_feature: [0.3, 0.5, 0.7]
	max_depth: [5, 10, 15, 25]
		n_estimators = 500
		min_sampes_leaf = 2
		max_feature = 0.7
		max_depth = 25

Figure X: Hyperparameter tuning II.

We decided to try increasing the number of trees in our forest, increase the minimum samples allowed at a leaf, and reduce the maximum number of features used when building a tree in order to add regularization to our model and address overfitting. The next two figures show the validation curves and learning curves for our Random Forest model.

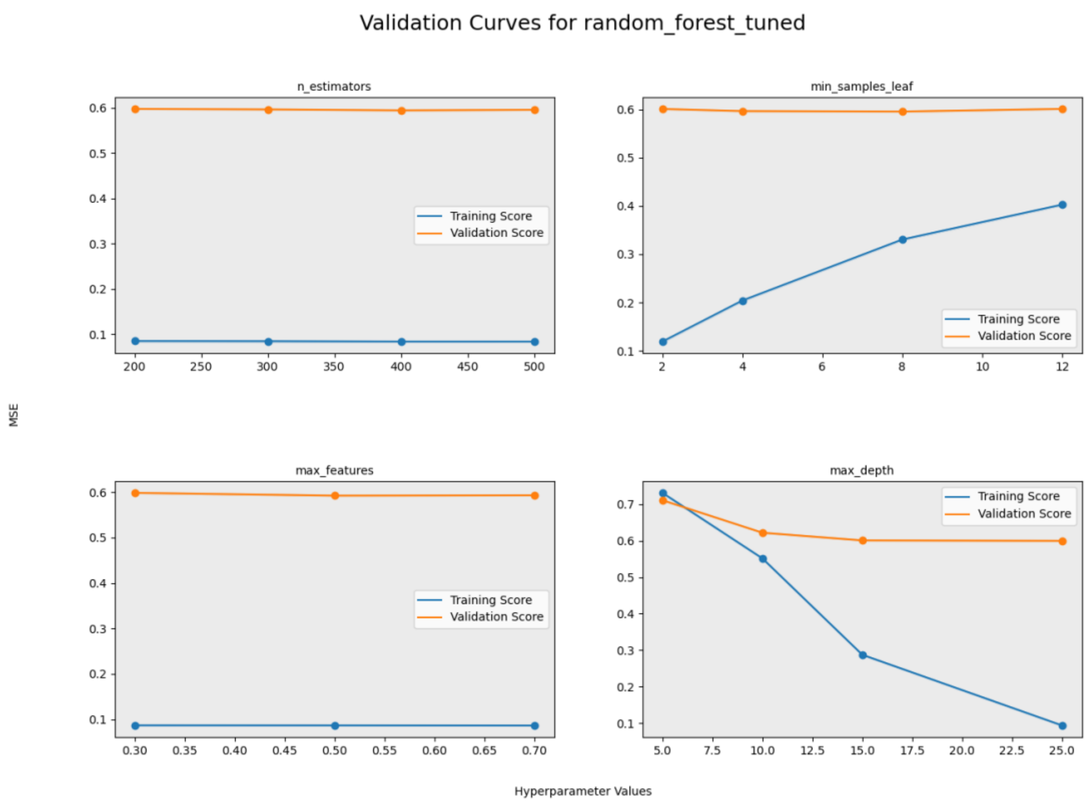


Figure X: Validation curves[12] for Random Forest model.

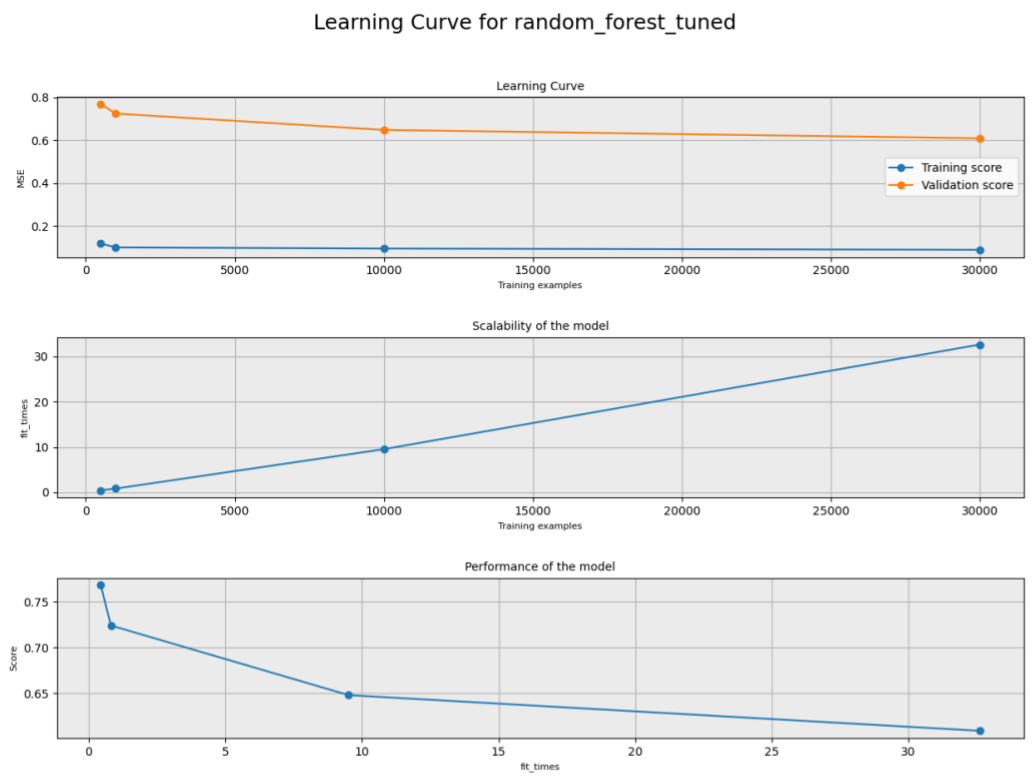


Figure X: Learning curves[13] for Random Forest model.

It is interesting to note that only max depth at around 6 levels is where the training and validation scores are roughly equal while the other hyperparameters did not address overfitting that heavily. We also plotted the learning, scalability, and performance curves of our model. Note that our MSE seems to slowly plateau with additional training data. The time complexity of our model seems to be roughly in the order of O(n) which is not bad. The performance curve shows that the model's MSE decreases with longer fit times which can generally mean a more complex model or fitting more training data.

Retrain and Test

Lastly, we retrained our model on the combined training and validation dataset and the relative feature importance of each feature can be visualized in the figure below.

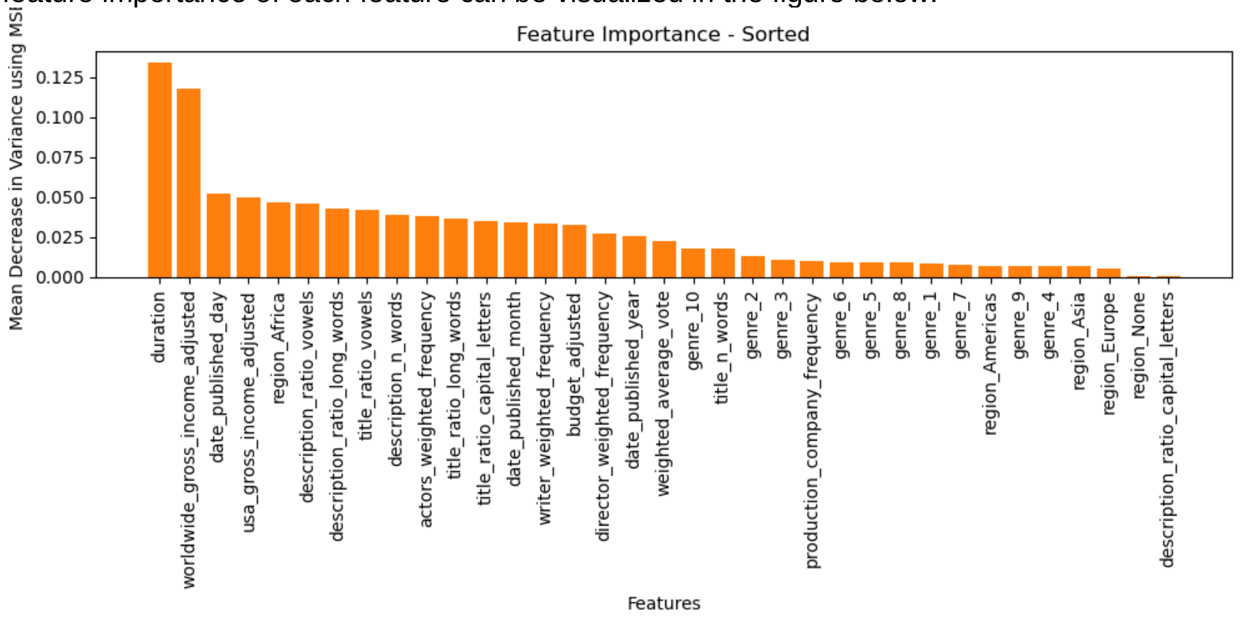


Figure X: Feature importance of our Random Forest model.

Some interesting things to note is that *date_published_day* actually has a relatively high feature importance, likely due to being somewhat of a proxy for days of the week. *Region_Africa* has a much higher feature importance compared to the other regions. On the other hand, *budget_adjusted* actually is in the middle of the pack for feature importance and genres, at least the way we encoded them, did not have as high relative feature importances.

Finally, we then proceeded to test on our testing dataset where the results are discussed in the next section.

4. Results

Testing and Model Evaluation

Our final MSE on the testing dataset is 0.602 on the scaled target label and is 0.869 once we reversed the scaling performed on the target label. The MSE by themselves are not the best to explain to humans how well our model is performing. We wanted to add context into this score that can explain truly how good our model is. We will do this in two ways:

1. Calculate the RMSE for the inverse scaled target label MSE which will bring us back into the same scale as the weighted average IMDb rating by reversing the squared operation in the MSE. We can think of the RMSE being used in a setting such as, Prediction Rating RMSE, in a one-to-one comparison against the prediction rating. Although not exactly apples-to-apples as using pure average error, this can still give us some idea of how good our model is that is intuitive for humans to understand. Our RMSE for the inverse scaled target label is 0.931. **This means that on (root mean squared) average, our error between the actual and predicted is 0.931. In the rating's range of 1-10, this is not a bad error although also not amazing.**

2. We can judge the MSE of our model against a model that randomly generates values between 1 and 10, the range of the weighted average IMDb rating. We will call this model a *Random Model* and the Random Forest model *Our Model*. The *Random Model* can be generated like so:

```
test_Y_random = np.random.uniform(1, 10, size=test_Y.shape)
```

The MSE of *Our Model* vs the *Random Model* is as follows:

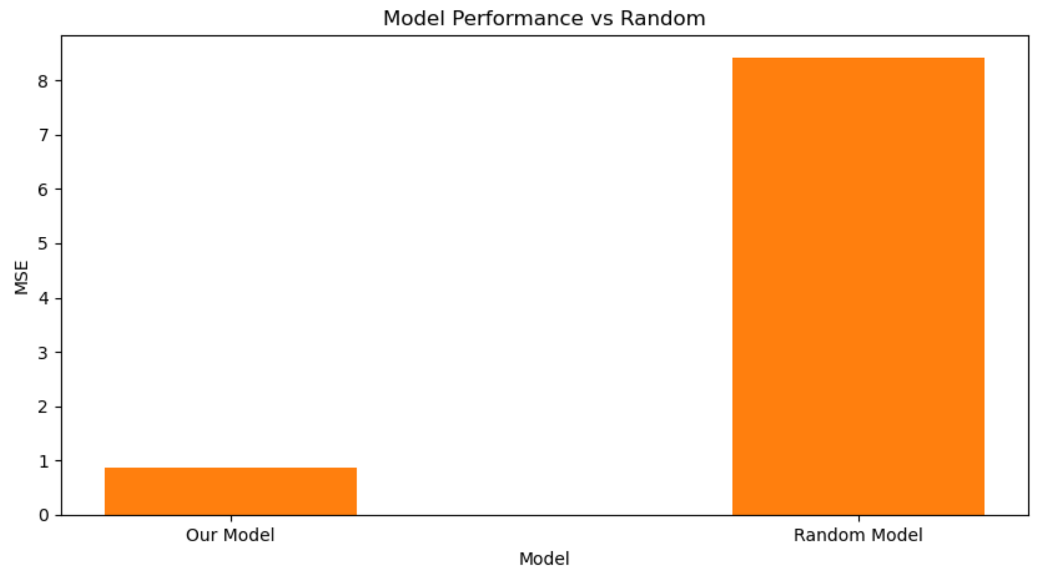


Figure X: *Our Model* (Tuned Random Forest Model) vs *Random Model* (Random Generating Model).

The MSE of our tuned model is almost a magnitude lower than the MSE for the Random Generated Model. Although this seems significant at first glance, let's perform statistical testing to confirm that the two predicted rating distributions are truly statistically different.

We used a *2-Sample T-Test* to check if the means of the two predicted rating distributions are different. Before we can do that, we need to first confirm that our two distributions are approximately normal and have approximately equal variances in order to use the T-Test.

- We will first performed the *Shapiro-Wilk Test*[14] on each distribution to confirm normality where H0 : Data drawn are from normal distribution with an   = 0.05.
- Then we will performed a *Bartlett Test*[15] on both distributions to confirm equal variances where H0 : All input samples have same variance with an = 0.05.

The table below shows our results from our 2 tests.

Test	**p-value**	**Action**
Shapiro-Wilk Test on Our Model Ratings	4.75e-40	Reject H0
Shapiro-Wilk Test on Random Model Ratings	0	Reject H0
Bartlett Test	0	Reject H0

Figure X: *Hypothesis testing results for normality and equal variance*.

Since all our p-values were below our   value, we rejected the null hypotheses for all three tests and our two distributions are not normal and do not have equal variances. Therefore, we cannot use a *2-Sample T-Test.* Instead, we will use:

- The *Mann Whitney U Test*[16] which is the non-parametric test that checks differences in medians or difference in locations between 2 distributions where H0 : The distribution under sample x is the sample y with an = 0.05.

Test	**p-value**	**Action**
Mann-Whitney U Test	3.80e-29	Reject H0

Figure X: *Mann-Whitney U Test results for independence*.

With a low p-value we can reject our null hypothesis and confirm that the two medians of our predicted rating and the random generating rating distributions are different.

We were able to use the two steps outlined above to gain some understanding of how well our model performs in the context of the problem and against just randomly guessing. The table below shows a sample of 20 test ratings and the predicted ratings from *Our Model* and the *Random Model.* In this sample, *Our Model* has a smaller error 80% of the time (84% on the entire testing dataset).

Actual Rating	*Our Model* Predicted Rating	*Random Model* Predicted Rating	*Our Model* Error	*Random Model* Error	Smaller Error

4.7	5.3	5.4	0.6	0.1	RM
7.2	6.2	9.7	-1.0	3.5	OM
6.8	6.7	4.5	-0.1	-2.2	OM
6.4	6.2	1.7	-0.2	-4.5	OM
6.5	5.6	4.2	-0.9	-1.4	OM
6.3	6.0	2.5	-0.3	-3.5	OM
5.7	6.8	9.8	1.1	3.1	OM
7.2	6.0	8.9	-1.2	2.9	OM
4.4	6.3	5.4	1.9	-0.8	RM
6.8	6.4	4.6	-0.4	-1.8	OM
5.6	6.1	5.1	0.5	-1.0	OM
5.2	5.4	7.5	0.2	2.1	OM
8.2	6.9	3.2	-1.3	-3.7	OM
7.4	6.4	6.6	-1.0	0.2	RM
5.3	5.9	2.3	0.6	-3.6	OM
5.1	6.3	2.8	1.2	-3.5	OM
6.2	7.0	1.7	0.8	-5.3	OM
5.8	5.9	9.6	0.1	3.7	OM
7.8	6.0	1.5	-1.8	-4.5	OM
7.3	6.4	6.4	-0.9	0.0	RM

Figure X: Sample of 20 ratings from test data showing the Actual, Predicted and a Randomly generated rating.

An interactive prediction game pitting your guess against *Our Model*'s prediction can be found in the GUI.

5. Conclusions

I learned a lot from this project, specifically how to best implement machine learning models and working with other data developers to integrate our code seamlessly.

Areas of improvement include better encoding methods for some of our features that had long text data, reduce the dimensionality of features used even further, try a VotingRegressor(), find ways to reduce the time complexity of performing our GridSearchCV and hyperparameter tuning.

6. Percentage of Code

Roughly (205/1237) * 100 = 16.5% from internet.

7. References

1.
2.
3.
4.
5.
6. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html
7. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
8. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html
9. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html
10. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html
11. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
12. https://scikit-learn.org/stable/auto_examples/model_selection/plot_validation_curve.html#sphx-glr-auto-examples-model-selection-plot-validation-curve-py
13. https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html#sphx-glr-auto-examples-model-selection-plot-learning-curve-py
14. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html
15. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bartlett.html#scipy.stats.bartlett
16. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html

8. Appendix

8.1 Random Seed

Random seed used in the modeling phase was *33*.

8.2 Log

- 11/4/2021
    - Inspected dataset: looks like we will have high cardinality with many categorical data. Did research into what we can do to reduce the cardinality and reduce dimensionality since just doing OHE will face sparse dataset and have Curse of Dimensionality and may easily overfit our models.
    - Options:
        - Group and cluster our categorical features that have many classes
        - Continue with OHE and use strong L1/L2 regularization
        - Catboost
        - MCA/FAMD
        - Frequency thresholds
        - Grouping / clustering
    - Thinking about doing MCA/FAMD or clustering
- 11/10/21
    - Started pipeline for models with models.py and models_helper.py
    - Did feature selection with team on: https://docs.google.com/spreadsheets/d/1qrFCjBWOn3emAx8xtMGC3bpg0CQsx3LMaK1O56ReDO0/edit?usp=sharing
    - Will use Random Forest to check for Feature importance
        - Then do linear regression with most important features as baseline
        - Then do RandomForest & GradBoost & AdaBoost then tune/cross validate and use best model
- 11/12/21 & 11/13/21
    - Helped out on pre-processing, specifically, worked on encoding of categorical features and feature engineering/extraction.
    - Worked on function to fit and transform actors, director, and writer frequencies on occurrence from training dataset. This is a proxy for quantifying the popularity of cast, writer, or director. If more than one person, we take the average of the frequencies. We also weighted the frequencies by the order of importance that the person played in the movie from the 'title_principals.csv' dataset.
        - The weight can be calculated as linear model where order of 1 is most important with an assigned weighting of 10/10.
        - Order of 2 is assigned weighting of 9/10.
        - Order of 10 is assigned weighting of 1/10.
        - We can calculate the function based on input 'order' to output the desired weighting.

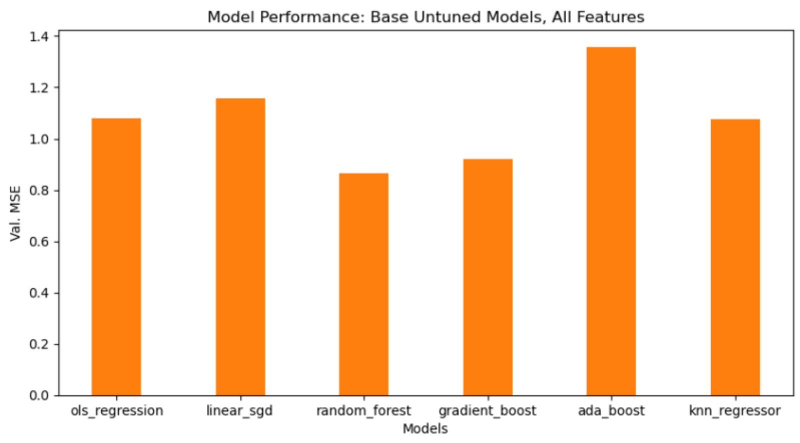
        - This is performed in the solve_linear_transformation() function in preprocessing_utils.py where:
            - order * m + b = weight multiplier, where m and b's are slope and intercept of our linear transformation
            - ie: 1m + b = 10/10, 2m + b = 9/10, 3m + b = 8/10, ..., 10m + b = 1/10
    - Next up is working on the encodings for
        - Genres: genre1, genre2, genre3 will be binary encoded
        - Production company: also frequency based but without order of importance weighting
        - Title: n words, ratio long words, ratio of vowels
        - Description: n words, ratio long words, ratio of vowels, ratio of punctuation, ratio of capital letters, ratio of capital letters after first word

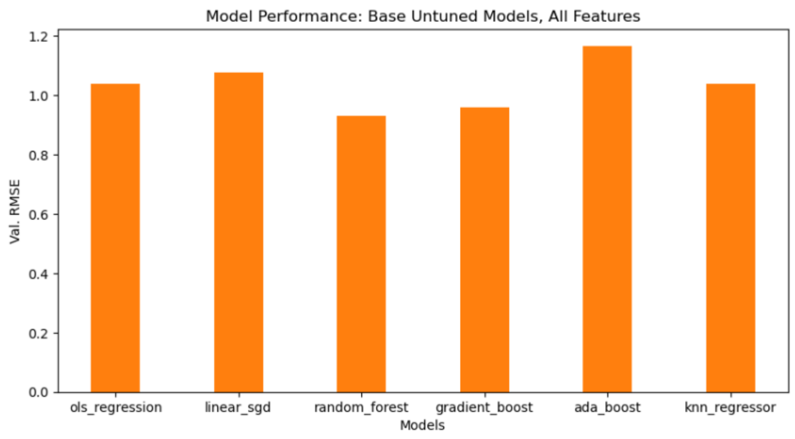
- 11/14/21
    - Finished categorical encoding for:
        - Genres: genre1, genre2, genre3 are binary encoded
            - binary_encoder_fit()
            - binary_encoder_transform()
            - binary_encoder()
        - Production company: also frequency based but without order of importance weighting
            - fit_production_company_frequency()
            - transform_production_company_frequency()
        - Title & Description: n words, ratio long words, ratio of vowels, ratio of interesting characters, ratio of capital letters
            - n_words()
            - ratio_long_words()
            - ratio_vowels()
            - ratio_interesting_characters()
            - ratio_capital_letters

    - From preprocessing_utils.py, I wrote approximately 205 lines of actual code (excluding comments). About 36 lines of those I had to use Google to find solutions or code templates that I repurposed for my needs.
    - Next up is to work on actual modeling.
- 11/15/21
    - Started working on modeling part:
        - Created architecture and pipeline to handle modeling part and connect with rest of project modules
        - Created basic random forest regressor to check most important features
    - Changed \`genre\` encoding from having separate \`genre1\`, \`genre2\`, and \`genre3\` binary encoded labels to just using \`genre\` where each label is given for each unique

tuple of `(genre1, genre2, genre3)` where contents inside tuple are sorted to take different ordering of same 3 genres in different observations into account.

- 11/17/21
  - Have base models trained and validation scores calculated
  - Used LinearRegression(), SGDRegressor(), RandomForestRegressor(), GradientBoostingRegressor(), AdaBoostingRegressor(), KNeighborsRegressor()
  - Here were their MSE and RMSE validation scores:





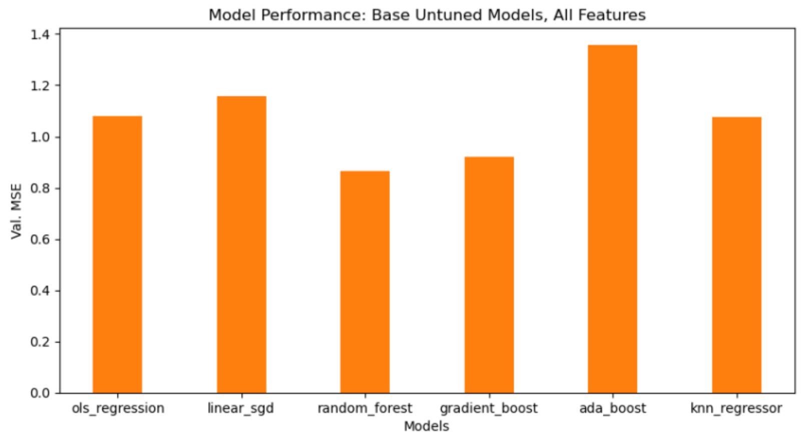
  - For GridSearch CV, looks like RandomForest and GradientBoosting have the most promise. KNN might be worth a try too to keep.

Overall modeling process:

1) Problem space:
   a. Problem type is regression where we are trying to predict a continuous value in 'weighted_avg_vote'
   b. Will be using Mean Squared Error as score to evaluate model's performance
2) Types of models available and what we are going to use in our modeling

a. Will be using models from SKlearn to prevent recreating code and try to tune them
b. Models to try out:
    i. SGDRegressor()
    ii. RandomForestRegressor()
    iii. GradientBoostingRegressor()
    iv. AdaBoostingRegressor(),
    v. KNeighborsRegressor()

3) Base Models and their performance



Random Forest and Gradient Boosting seem to be the best performers out of the box with default SKlearn parameters. KNN is the next best performer. We will use these 3 in our hyperparameter tuning and cross validation phase.

4) Hyperparameter tuning and validation
    a. Strategy is to use grid search on a set number of parameters per model and test on validation set
    b. Parameters and their values to test are:
        i.
    c. Results are:
5) Selecting best model
    a. Selected best model by taking best model and parameters from hyperparameter tuning and validation
    b. Our best performing model is:
6) Best model evaluation and prediction
    a. The prediction score on testing data is:
    b. The score means … (provide context)