Tyler Gourley
26 February 2025

ECEN 471 Project 2
Sanguine Synergy: Modeling Team Chemistry in Soccer

1 Introduction and Problem Formulation

1.1 Background

In soccer, success on the field is not solely determined by the individual talent of players but also by their ability to function as a cohesive unit. The concept of **team chemistry**—the interplay of familiarity, experience, and tactical synchronization among players—remains an elusive yet critical factor in predicting team performance. While traditional analytics focus on player statistics, formations, and tactical strategies, the impact of team synergy has been largely unquantifiable… until now.

This paper seeks to bridge that gap by developing a machine learning model that captures the impact of team chemistry on match outcomes. Using a dataset of playing time distributions, we construct a framework to analyze how different lineups influence goal differential (a fundamental metric of success). We will aim to solve this **regression problem** by employing techniques such as linear regression, polynomial regression, and regularization. In doing so, we explore how team chemistry among players contributes to the ultimate goal of every soccer team, coach, manager and player: winning matches.

2.2 Impact

The impact of a highly accurate model describing the effect of team chemistry would alter the landscape of soccer forever, especially if that model indicated that strong team chemistry leads to an increased probability for success. Unfortunately, using the preliminary methods that are described in this paper, we will not be able to say for certain how much impact team chemistry truly has on winning games. The goal of this paper—however—is to create a model that can accurately predict goal differential based on a team's previous playing experience together.

Even so, football coaches could use the trained model created in this paper to get a preliminary estimate as to their team's expected performance (against any random opponent) based solely on their current team synergy.

2.3 Problem Structure

The inputs to the machine learning models are 55 feature vectors where each element is quantified by the number of games two players have played together in their professional careers. Given there are 11 starters on a soccer team and are concerned about two-player combinations, this size of the input feature vector can be verified easily: ${}_{11}C_2 = 55$.

The output of the model is goal differential—represented with a single integer. That is to be calculated as the difference of goals scored and goals allowed. For this paper, the metric for goal differential will only include goals that are scored during the 90-minute period (including extra time). It will NOT include goals scored from shoot-outs or any other tie-breaking procedure. So, a match that comes down to shoot-out will always have a goal differential of 0 regardless of if a team emerges victorious due to scoring more goals in the shoot-out.

I expect that teams with high team chemistry will be more likely to perform better. Because I believe there is a strong relationship between team chemistry and team success, I am confident a model can be made that drastically outperforms a simple baseline model.

2.3 Additional Depth

This project builds upon the dataset from Project 1. In order to go above and beyond, I will be increasing the size of the dataset. So instead of using data from just 2008, I will also be using data from 2009 as well. This more than doubles the size of the dataset (increasing from 27,014 to 55,780 data points).

I expect that linear and logistic regression models will perform much worse than the kNN and Decision Tree models. The reason for this is that the input data (minutes played together for specific player combinations) does not necessarily have a linear relationship when compared to game outcomes. Because of this, I think the kNN and Decision Tree algorithms will do better at finding the nonlinear relationships in the data.

2 Methods and Experiments

2.1 Setup

2.1.1 Source of the Dataset

The data used in this project comes from an SQLite database that was assembled in Dr. Harrison's research lab that contains over 20 years of playing time data. Originally, the data was scraped from the ESPN website. In my responsibilities as a researcher, I was the one to compile the raw playing time data into the .csv files that will be used as the dataset for this project (the dataset .csv files can be found at this link). We will be only analyzing data from 2008 and 2009 which contains data for 27,890 soccer matches—which corresponds to twice as many inputs (55,780) because each match has data for both teams that participated in the match.

2.1.2 Test/Train Split

First, we separated the data into a training dataset and test dataset. The nature of the dataset is that each match has 2 inputs associated with it (one for each team). So, in order to avoid data leakage into the test set, the data was first grouped by match before performing a test/train split. We do this by using the GroupShuffleSplit() method splitting on match id (see documentation for sklearn.model_selection.GroupShuffleSplit). By stratifying the random split by match id, we are able to ensure that inputs from the same match are both found in the same set (either both in the test set or both in the training set). We used a 90/10 split—90% for training data and 10% for test data.

2.1.3 Estimating Enew

We will use the holdout validation method to estimate the error on data the model has never seen before (E_{new}). This decision was made to decrease the computation time of estimating E_{new}—holdout validation is less computationally expensive than cross-validation. Because we have a large dataset, we expect this method will provide accurate estimates for E_{new}.

When creating the holdout validation set, we also stratified the data by match id in order to prevent data leakage (see section 2.2). We decided to use 20% of the *original* dataset as the validation set. Because we had already split the data into train and test, we were sure to take that into account and split the data accordingly. We normalized the train/validation split to the 20/90 ≈ 0.22 (~22.2% validate and ~77.8% training). This left the final allocation of the data as 70% training data, 20% validation data, and 10% test data.

We will set the random seed to be 999 for all functions that use randomness. Additionally, we will be using the scikit-learn Standard Scalar (see sklearn.preprocessing.StandardScaler) to scale the inputs. We made sure to scale the inputs after the data was split into test, train, and validation sets to avoid data leakage.

2.1.4 Error Function

The nature of our output data (goal differential) is numerical. Thus, we will be using regression models. The error function we will be optimizing for is mean squared error (MSE).

2.1.5 Baseline Model

We will use Scikit-learn Dummy Regressor (see sklearn.dummy.DummyRegressor package) as our baseline model to which we will compare the machine learning models we create. This baseline model will always predict a goal differential of 0. The strategy of the baseline model was set to always estimate the statistical mean of the training set. This was accomplished by setting the strategy parameter to 'mean' (strategy='mean'). All other parameters were left to the default values.

Using the holdout validation strategy on the dummy regressor model, we get an estimate for MSE on new data of $E_{new} \approx 3.0929$. This will be the baseline model that we will use to evaluate the performance of the other models we create.

2.2 Models without Regularization

2.2.1 Linear regression model

For linear regression, we will use the scikit-learn Linear Regression model (see documentation for sklearn.linear_model.LinearRegression). We used the default value for all the model parameters. This model produced an estimated E_{new} of 3.088, using holdout validation. Finally, we trained a linear regression model on the full dataset (comprising both the training set and validation set, but not including the test set).

2.2.2 Linear Regression model with polynomial features

Now we move to training a Linear Regression model with *polynomial features*. We will use the (see sklearn.preprocessing.PolynomialFeatures). Our plan was to sweep through the following list of hyperparameters for degree: [2, 3, 4, 5, 9]. Unfortunately, we ran into a runtime error: "Your session crashed after using all available RAM" when fitting our data to a model with a degree of 3 or higher. Google Colab notebooks limit the RAM available in a session to 12.7 GB. Because of this, we were only able to train a model of a degree 2 polynomial.

The reason that we used up all the available RAM is that our data balloons in size when using the scikit-learn PolynomialFeatures() function. Modeling a higher degree polynomial requires combinatorially increasing features. In general, with n original features and degree d, you get roughly ${}_{(n+d)}C_d$ new features. For our dataset where we have 55 features per sample, to model a degree-3 polynomial, our dataset would have ${}_{58}C_3$ = 30,856 features per sample (over 560 times larger than our original dataset). To solve this problem, we would need to in some way limit the number of features of our dataset. I have come up with some ideas to consolidate the features but still describe the data in a useful manner: use average and standard deviation of the features [2 features], take quartile scores (median, upper and lower quartiles, and maximum and minimum) [5 features], randomly drop 60% of the features (similar to dropout in a neural network) [33 features], and distribute the features in buckets based position (forwards, midfielders, and defenders) and take the average of each [3 features]. If I had more time, I would try all of these different data preprocessing strategies. However, since this project is not focused on data augmentation (manipulating the dataset in preparation for training a model), I will leave this as a consideration for future refinement, which I will likely do as a part of my research responsibilities.

The degree-2 polynomial model had a MSE of 3.269, determined from the validation set.

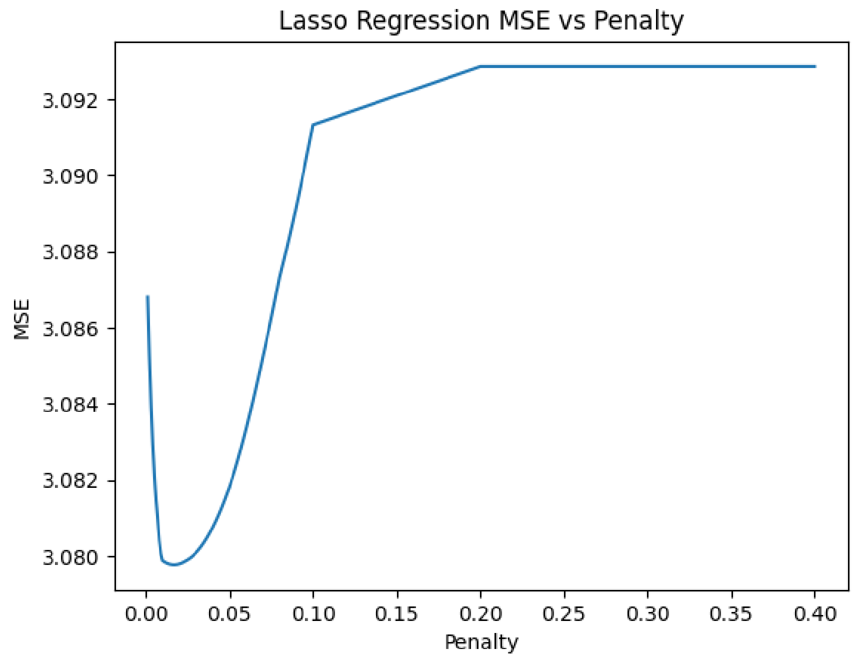
Finally, we trained a with a polynomial degree of 2 (leaving all other parameters as the default values) on the full train/holdout (not test) dataset.

2.3 Models with Regularization

2.3.1 Linear regression model with regularization

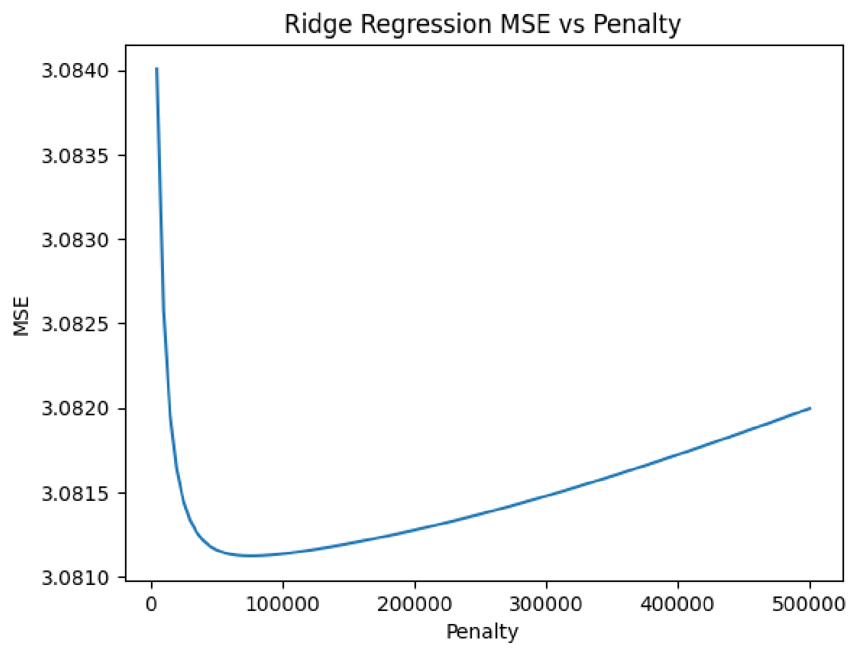
We will use 3 different Regularization methods in an attempt to improve performance on our linear regression model: Lasso (L1), Ridge (L2), and Elastic Net. For each method, we will perform a hyperparameter sweep of the alpha term (penalizing the model's coefficient weights for being too large and overfitting). We will the holdout set to estimate E_{new} of each hyperparameter and determine the most optimal parameter by choosing the one with the lowest error on the holdout data.

For Lasso regression, we used the scikit-learn Lasso() method. We swept the values for the alpha hyperparameter in the interval [0.001, 0.1] with a step size of 0.001. We left all the other parameters as their default values. From this, we found that the optimal value for alpha is 0.017, with an MSE on new data of 3.0797.



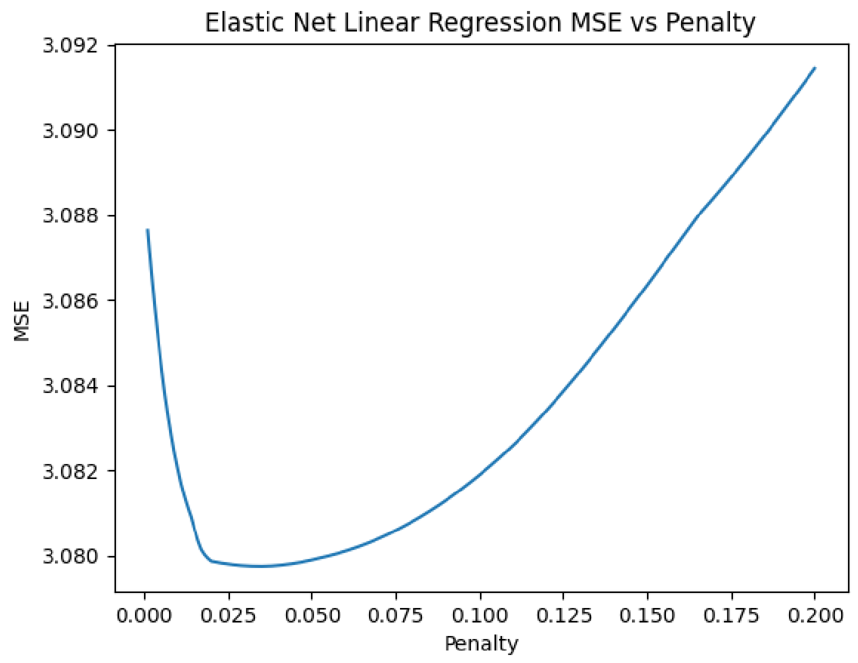
2.3.1A – Lasso Regression hyperparameter sweep of alpha

For Ridge regression, we used the scikit-learn Ridge() function. We swept the values for alpha in the interval [5,000, 500,000] with a step size of 1. We left all the other hyperparameters to be the default values. We found that the optimal value for alpha is 75000, with an MSE on the validation set being 3.0811.



2.3.1B – Lasso Regression hyperparameter sweep of alpha

For the Elastic Net method of regularization, we used the scikit-learn ElasticNet() function. We swept the values for alpha in the interval [0.001, 0.2] with a step size of 0.001. We left all the other hyperparameters to be the default values. We found that the optimal value for alpha is 0.035. Using the holdout validation strategy, we determined the estimated E_{new} for this model to be 3.0797.



2.3.1C – Elastic Net Regression hyperparameter sweep of alpha

2.3.2 Linear regression model with polynomial features and regularization

We will use the same 3 Regularization methods—Lasso (L1), Ridge (L2), and Elastic Net—to improve performance on our polynomial model. Same as before, we will perform a hyperparameter sweep of the alpha parameter for each method. Like above when we trained the polynomial model without regularization, we are unable to train a polynomial with a degree larger than 2 because of the large number of features in our data set (for further details, see the discussion in section 2.2.2). For this reason, we will only sweep through the alpha parameters and plot the resulting graphs.

For Lasso regression, we swept the values for the alpha hyperparameter in the interval [0.05, 5] with a step size of 0.05. All the other parameters were left at their default values. From this, we found that the optimal value for alpha is 0.05, with an MSE on new data of 3.0796. Since the optimal value is at the lower bound of the parameters we tested (and we cannot go any lower unless we run into runtime errors where the algorithm does not converge), we can conclude that there is likely no further improvement we can find without just doing a model with no regularization.

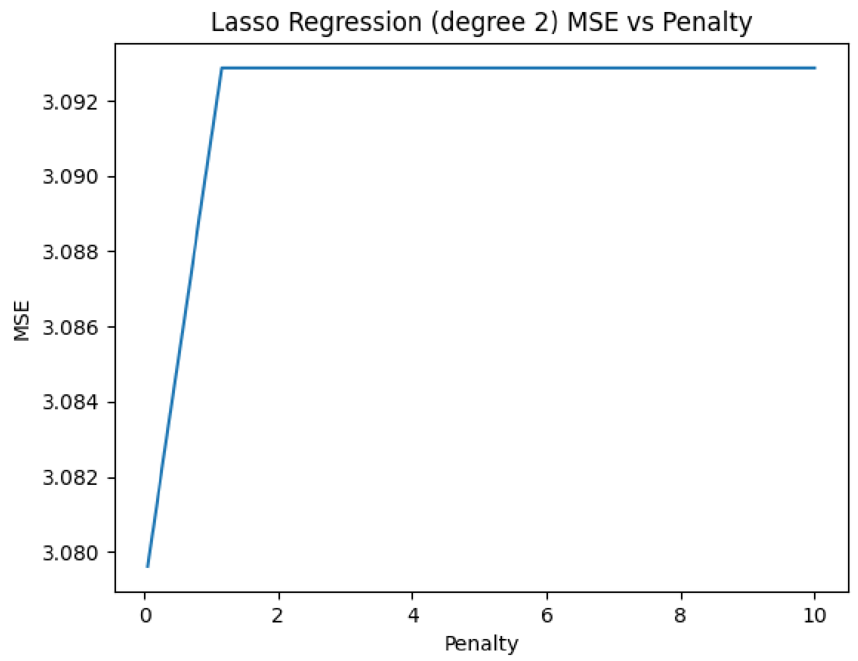


Figure 2.3.2A – Lasso Regression (Polynomial features) hyperparameter sweep of alpha

For Ridge regression, we swept the values for alpha in the interval [800,000, 900,000] with a step size of 1,000. We left all the other hyperparameters to be the default values. We found that the optimal value for alpha is 877,000, with an MSE on the validation set being 3.0821.

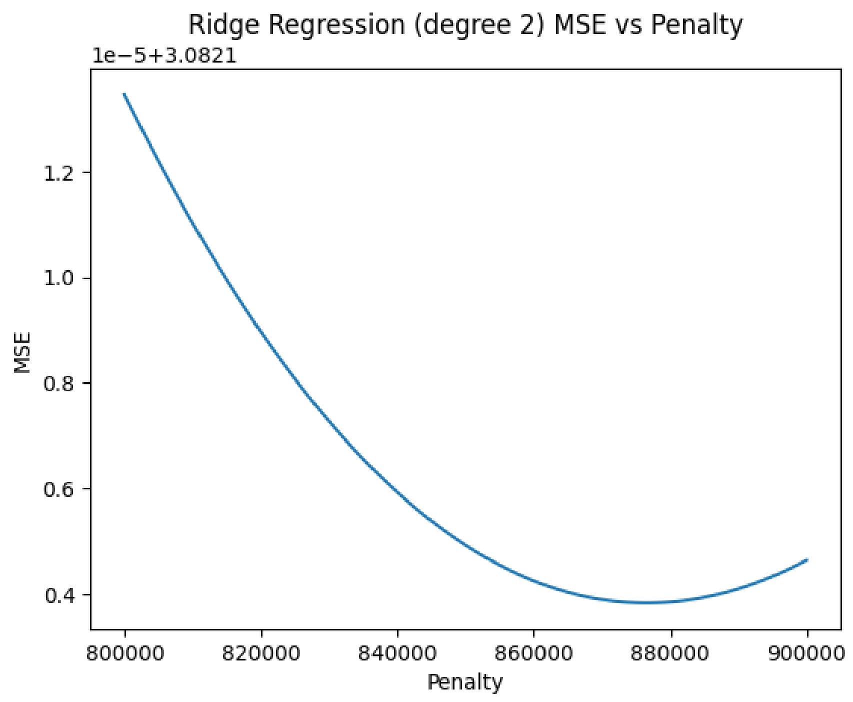


Figure 2.3.2B – Ridge Regression (Polynomial features) hyperparameter sweep of alpha

For the Elastic Net method of regularization, we. We swept the values for alpha in the interval [0.01, 0.2] with a step size of 0.035. We left all the other hyperparameters to be the default values. We found that the optimal value for alpha was 0.045. Using the holdout validation strategy, we determined the estimated E_{new} for this model to be 3.0778.

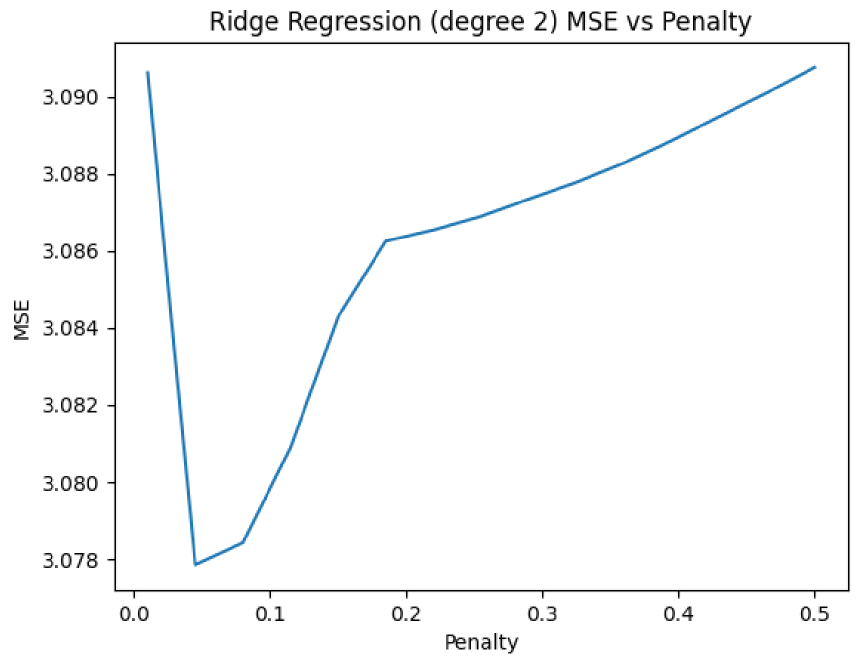


Figure 2.3.2C – Elastic Net Regression (Polynomial features) hyperparameter sweep of alpha

2.4 Compare Final Models

2.4.1 Test all models on Test dataset

For the 9 final models we have trained, we used the test dataset to evaluate each model's performance. Here are the results:

Model	E_{test} (MSE)
Baseline	3.0613
Linear Regression	3.0514
Polynomial (degree=2)	3.1702
Lasso	3.0507
Ridge	3.0492
Elastic Net	3.0506
Lasso Polynomial (degree=2)	3.0486
Ridge Polynomial (degree=2)	3.0469
Elastic Net Polynomial (degree=2)	3.0448

Table 2.4.1A – Table of final performances (E_{test}) for each of the final models

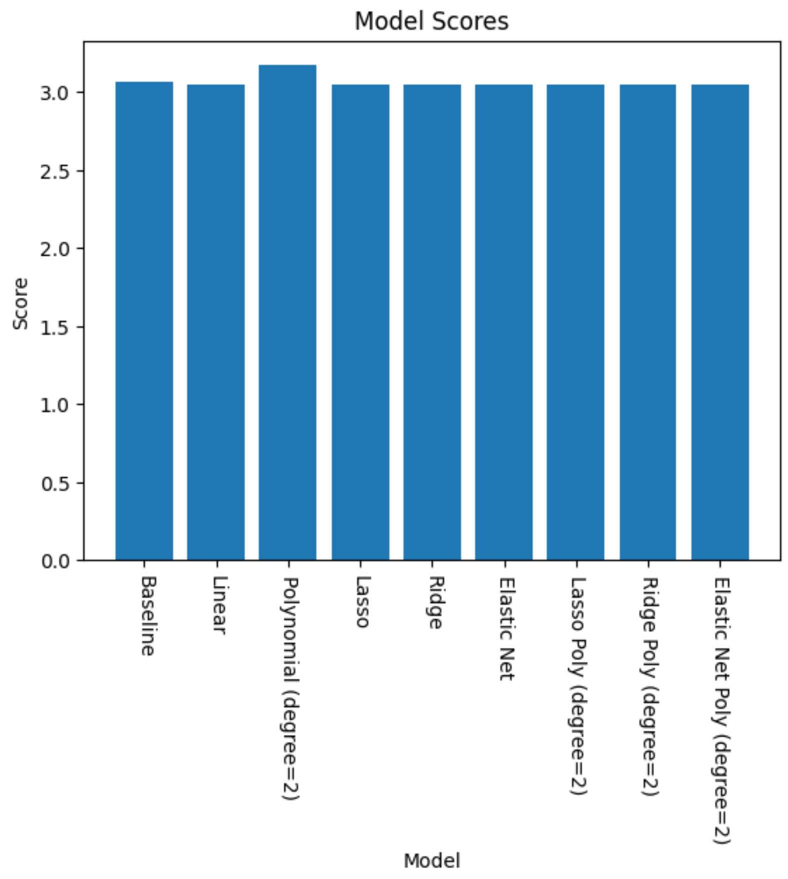


Figure 2.4.1A – Plot of E_{test} for each of the final models

3 Discussion and Conclusion

3.1 Analyze Performance Metrics

None of the models outperformed the baseline model. There was also no statistically significant difference between any of the models that used regularization and those that did not. In fact, there was no statistically significant difference between any of the models at all.

Because our data was not fit for the use of linear/polynomial models, it is difficult to say what regularization technique worked the best—they all failed equally. Similarly, we cannot tell for certain which models would be most robust to outliers since the data does not fit a linear relationship.

3.2 Discussion and Final Conclusion

Unfortunately, I do not have much confidence in any of the models that were created in this Project. Because there is not a significant improvement to the error of the models I created compared to the baseline model, I would say that there are a lot of limitations. I would not put any of these models into production in the real world—they do not perform better than guessing 0 every time. I have concluded that linear (and polynomial) models do not do well to model my data since there is not a linear relationship between the inputs and the labels.

Compared to the models that were created from Project 1, the linear models (from this Project) perform significantly worse. Up to this point, the best model was the Decision Tree from Project 1. In the future, I want to use bagging techniques (such as Decision Forests) and boosting methods to improve the best model we have got so far, the Decision Tree model.