SYDE 522 Final Project: Predicting Running Calorie Expenditure

Erick Yan
20832381
e5yan@uwaterloo.ca

Maxwell Mastali
20880914
m2mastal@uwaterloo.ca

Abstract

This report investigates the preprocessing, model selection, and model performance evaluation steps we took to predict calories burned while running. Since we constructed our own dataset, we first performed data cleaning to remove null values and outliers, followed by feature scaling to standardize the dataset. After selecting key features, we selected and evaluated various models including regression, support vector regression (SVR), and the multi-layer perceptron (MLP). The results showed that the SVR model with a linear kernel outperformed the other models with a testing RMSE of 10.2, revealing that our dataset's underlying relationships are relatively linear. Due to our dataset's linearity, complex models such as high degree polynomial regression models and the MLP model resulted in a higher RMSE and overfitting. These findings highlight the importance of selecting the appropriate model based on the nature of the data.

1 Introduction

As avid part-time, amateur athletes, this study was largely motivated by our insatiable curiosity on whether we can accurately predict the amount of calories burned from running. The first question we sought to answer was what contributes to calorie expenditure? Most fitness articles will state that body weight, gender, workout intensity, workout environment (ex. road grade), and climate play the most pivotal roles in determining the amount of calories burned [1]. On a granular level, our body is equipped with a basal metabolic rate to perform critical functions such as fighting diseases or processing nutrients by breaking down energy releasing molecules. During exercise, our body specifically looks to break down adenosine triphosphate (ATP) to perform skeletal muscle contractions, which is the main factor that increases caloric expenditure [2]. ATP is a molecule which the body uses to perform cellular work. The next question is how is caloric expenditure measured?

2 Description of Task and Data

2.1 Modern Measurement Methods

To understand how caloric expenditure is measured, we have to venture yet again into the biology of the human body. A person mainly depends on energy substrates, or nutrient sources, such as carbohydrates and fat to release energy and perform bodily functions. However, the amount of energy available in these energy substrates alone isn't what dictates energy expenditure; it's the amount available relative to oxygen consumption. This is why a person with a higher VO_2 max (maximum oxygen consumption) can burn more calories and a measurement of one's respiratory exchange ratio (RER) is required to accurately capture their caloric expenditure.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

The most common way to measure how much calories are burned during exercise is to monitor the CO_2 release. Exercise intensity through the measure of heart rate is also a reliable metric, specifically in the form of power. For mainly aerobic exercises where oxygen consumption can reach steady state, there is a linear relationship between heart rate intensity and oxygen consumption, which can then be used to derive caloric expenditure. Thorough research on this method has been performed through the use of cycle ergometers which have the ability to accurately quantify and adjust power exerted by an athlete. State of the art caloric expenditure measurement methods involve consuming doubly labeled water instead of requiring an exhalation hood. The labeled hydrogen from the water is passed through the body but some of the labeled oxygen is released as CO_2. Through urine sampling, we can gather the ratio of labeled oxygen to labeled hydrogen to derive the amount of oxygen consumed, and therefore the amount of calories burned [3]. Although these methods are all very exorbitant, some costing as much as $600 per test, they've unraveled groundbreaking, controversial theories like how the total energy expenditure discrepancy between someone that exercised and someone that didn't is minimal.

For the average athlete who's not looking to break the bank for an accurate caloric snapshot, we only care for an estimate of our caloric expenditure during exercise. The meteoric rise of fitness devices in the 21st century from reputable brands like Garmin and Apple have provided athletes with a snapshot of their workout quality through metrics like Active and Resting Calories. These devices are often equipped with optical photoplethysmography sensors, which can measure heart rate and blood oxygen levels [4], accelerometers, altimeters, barometers, and compasses, just to name a few. Garmin has stated that their algorithm relies on various factors to determine an individual's Active Metabolic Rate such as individual characteristics (gender, body weight, height, etc.), movement patterns, and heart rate intensity, but the specifics are proprietary information [5]. The same story goes for Apple watches. Although both Garmin and Apple boast accurate measurements of caloric expenditure, it's mostly the case when compared with other fitness brands. Inevitably, at the cost of affordability, the sensors come with several vulnerabilities that can lead to inaccuracies. Nonetheless, most athletes, including ourselves, continue to use these devices and this study aims to capture the high level of accuracy achieved by these fitness brands.

2.2 Data Collection

As amateur runners, we've amassed hundreds of running data instances in the past 2 years and decided to leverage our own runs as the model's dataset. By creating a Python script using Strava API to extract our run data, we performed some initial processing on both datasets to align on columns and their unit types. Unfortunately, we had to relinquish more granular metrics available from my Garmin watch, such as Run Cadence and Power, because the Apple watch didn't provide it to Strava. The final dataset details are shown in the following table.

3 Preprocessing Rationale and Methodology

3.1 Data Cleaning

Since we developed our own dataset, we needed to clean the data before analyzing it. Some steps we took included removing instances with outliers and null entries. Our goal with removing outliers was to retain as much valid data as possible and discard any data that might be misleading with null fields. By observing our data, we found that only a relatively small number of instances have data with null fields, so we decided that it was acceptable to completely drop those ill-defined instances. However, had the distribution of null fields been different across the instances, we may have had other approaches such as estimating the null field or removing the feature across the dataset if needed. Once we had well-defined data, the next step was to remove outliers.

Various methods to remove outliers were considered, one being using the z-score and another being using the interquartile range. We chose to use their z-score since it's more sensitive to extreme outliers and since it provides a clear and standardized metric of how far each point deviates from the mean. Before applying the z-score method, we ensured not to apply it to the "distance" and "moving_time" columns since those can still have extreme values but still be valid data that we do not want to discard. Next, we calculated the z-score for each datapoint and only kept the runs where all its associated features are within a magnitude of the z-score standard of 3. The standard of 3 is often used to find outliers, and we tested various cut-off z-scores and confirmed that 3 was the best for our data.

Features	Description
Distance	The total distance covered during the activity, measured in meters.
Average Heart Rate	The average heart rate of the individual throughout the activity, measured in beats per minute (bpm).
Max Heart Rate	The maximum heart rate of the individual achieved from the activity, measured in beats per minute (bpm).
Average Speed	The average speed maintained during the activity, measured in meters per second.
Max Speed	The max speed achieved during the activity, measured in meters per second.
Average Grade Adjusted Speed	The grade-adjusted speed maintained during the activity, measured in meters per second.
Total Elevation	The total elevation gained during the activity, measured in meters.
Elapsed Time	The total time taken from the start to the end of the activity, measured in seconds.
Moving Time	The total time spent actively moving during the activity, measured in seconds.
Elevation Low	The lowest elevation point recorded during the activity, measured in meters.
Elevation High	The highest elevation point recorded during the activity, measured in meters.
Calories	The total amount of energy burned during the activity, measured in calories.

Table 1: Dataset Features

3.2 Feature Scaling

We considered feature scaling our data to ensure that one feature does not dominate other features due to a difference in their scales. In order to do this, we standardized our data so that each column has a mean of 0 with a standard deviation of 1. There are other feature scaling methods we considered, but standardizing our data in this way was optimal since it works with our assumption of normally distributed data, gives the same scale to all columns and ensures one does not dominate due to scale, and is easy to compare.

3.3 Feature Selection

Feature selection was implemented as a means of minimizing the redundancy across our feature set. Principal Component Analysis was initially considered, but since the dataset only contained 12 features, we decided that dimensionality reduction would be excessive. We chose Recursive Feature Elimination (RFE) as our primary feature selection method. RFE is known to be computationally expensive, especially on large datasets, which would be excellent for our relatively small dataset. While RFE is typically advised for datasets with intricate feature interactions [6], we were cautious about prematurely discarding features that our domain expertise suggested might be interconnected. More specifically, we wanted to leverage RFE to filter for a feature set that most effectively interacted with each other and determine the relative importance of each feature through model-driven results. RFE performs very well in the sense that it is one of the most optimal feature selection methods that removes less significant features and considers feature interactions unlike the foundational filtering and wrapper methods.

To implement RFE, an estimator model is required to evaluate the importance of each feature. We decided to test a list of models to gain a comprehensive understanding of feature importance across different algorithmic approaches [7]. The way this process was facilitated was through sklearn's Pipeline function in which we'd use a base model as both the RFE estimator and the final evaluator. The base models tested were Random Forest, Gradient Boosting, and Linear Regression. Random Forest can provide robust feature importance rankings with strong performance in nonlinear relationships among the variables, while Gradient Boosting offers a different perspective from sequential error minimization with high-dimensional datasets. Linear Regression was included to serve as a baseline, linear model to provide a simple, interpretable approach in contrast to the previous

two. We also varied the number of features that were selected after the estimator evaluated the relative importance of the feature set. After testing each algorithm, we can conclude on the most effective estimator to finally eliminate redundant features.

4 Model Selection Rationale and Methodology

4.1 Regression

We selected regression as a baseline algorithm to compare with our more complex models. Regression is useful since we can capture both linear and non-linear relationships in our data based on the degree we use. The degree has a large impact on the model's ability to fit to the data, where lower-degrees result in a stiff model and high-degrees allow for more flexibility in adapting to complex datasets. However if it's used on a simple dataset, overfitting may occur. We will first vary the degrees with the following values: [3, 5, 8, 10, 12]. This will cover a wide range of degrees, and then we can focus on where we see good results. For each degree, we will first use PolynomialFeatures() from sklearn.preprocessing to fit the data to the number of degrees.

In addition to varying the degree, we will also vary the regularization term λ. This dictates the importance the model gives to keeping weights small. For example, a large regularization term means that there is a large penalty for large weights. We will experiment with a wide range of regularizations by iterating through np.exp(np.linspace(-20, 5, 50)). For each lmbd (λ), we will create a model based on it using Ridge(alpha=lmbd). Our high-level approach to regression will be to vary the degree our data is fitted to. For each degree, we will create a model with each of our regularization values and determine the model's training and validation RMSE. For each degree, we will find the combination with the lowest validation RMSE to create the best model for the degree to test on our testing data. After iterating through each degree, we will then select the specific combination of degrees and regularization that has the lowest testing RMSE to create our best model.

4.2 Support Vector Regression

Support Vector Regression (SVR) was used as another model to work on our dataset. We chose it as it works well with non-linear relationships, which seems to exist in our dataset from our work with the regression model. Additionally, it performs well with a high number of features due to its unique kernel trick, and this is beneficial for us since our dataset contains 7 features.

The first SVR parameter we will tune is the kernel used. The kernel is a function that transforms the input data into a higher-dimensional space where it may be easier to decipher a relationship between the input and the target. There are various possible functions and kernels to use, but we will cover the most widespread functions, which are both the linear and Gaussian Radial Basis function kernel. The linear kernel assumes the data can be separated linearly, while the Gaussian Radial Basis function kernel will aim to separate the data into as many dimensions as it deems necessary.

While optimizing our linear kernel SVR model, we will tune and optimize the regularization C and epsilon ϵ. As we work with SVR, we will need to optimize the regularization C, which defines the importance the model gives to staying within the margin specified. A large regularization C leads to the model trying to stay within the margin and fit the data perfectly, but this can lead to overfitting. We will experiment with various values for C, iterating through np.logspace(-3, 3, 10) to find the optimal regularization. At the same time, we will be varying epsilon ϵ, which defines the margin for allowed error before penalizing our model. A large epsilon ϵ will lead to a wide margin for allowed error and result in a relatively simple model that may be prone to underfitting and not being able to capture the dataset's relationships. On the other hand, a small epsilon ϵ will lead to a tight margin for allowed error which could lead to overfitting. It's a balancing act to have a proper regularization C while maintaining a reasonable margin of error epsilon ϵ. For the SVR model using a linear kernel, we will vary the regularization C and epsilon ϵ to find the optimal combination that works the best on our validation data. We will then run this model on our testing data.

We will also examine how using the Gaussian Radial Basis Function as our kernel can change our SVR model. We will again optimize the regularization λ, which represents the same theory as for the linear kernel but now has a symbol λ instead of C. We will also optimize for gamma γ, which scales the importance of training points in defining the model, as shown by the equation below.

$$k(x_1, x_2) = e^{-\gamma \|x_1 - x_2\|^2}$$

As gamma γ increases, the model becomes broad and may lead to a simple model and thus underfit. However, as gamma γ decreases, the model becomes more narrow with finer details, and this can lead to overfitting by the model being too sensitive to noise. We want to strike a good balance when picking a gamma γ, as we want to be able to capture relationships within the data while not having a too simple or complex model. For the SVR model using the Gaussian Radial Basis Function kernel, we will vary the regularization λ and gamma γ to find the optimal combination that works best on our validation data. We will then run this model on our testing data.

4.3 Multi-Layered Perceptron

We selected MLP as the final algorithm to test because we were interested in how more complex models would perform on the preprocessed dataset. More specifically, we were aware of potential nonlinear relationships between features such as distance and total_elevation_gain which an MLP model could capture on and perform well to a certain degree relative to purely linear models like Linear Regression. I specifically chose PyTorch as our MLP library of choice instead of scikit-learn because it allows for a more fine-grained control over the model's architecture through its extensive customizable parameters.

The first step before training the MLP model was to perform hyperparameter tuning, in which we'd need to select the most important subset of parameters to vary and test on. The first parameter I selected was the learning rate because it controls the step size the optimizer takes in the weight space to minimize the error. Learning rate plays a vital role in controlling how well the model adapts to the error feedback, where large values can cause overshoot to a suboptimal solution and small values result in a slow learning model. I tested the learning rate using a wide range of values separated by powers of 10 to determine which value would result in a model best optimized for learning.

The second parameter I selected was the number of neurons in the hidden layer. The neuron count can improve the model's performance significantly by introducing a new set of features that the model can learn from. The more neurons are introduced, the more accurate the model becomes, but after a certain threshold, the improvements become negligible [8]. Since learning behavior varies with different configured models, we can identify the learning threshold for our MLP by incrementing the neuron count by factors of 2.

The third parameter was the number of hidden layers. As demonstrated in class, increasing the hidden layer count allows the model to capture more complex, non-linear patterns from the training dataset. To isolate the effects of increasing the number of layers, we kept the number of neurons in each layer consistent and incremented the layer count until the performance improvements became negligible, similar to the neuron count parameter.

The fourth parameter we tested was experimenting with different neuron count and layer patterns. We were particularly interested in how the model would perform if the number of neurons increased, decreased, or reached a peak midway as the data moved through each layer. We extended this idea by also varying the number of layers within these patterns. By balancing the complexity across layers, we can observe the model's preference in terms of when to introduce more features (i.e. early, in the middle, or later on in the layers).

The fifth parameter we varied was the activation function. The activation function plays a pivotal role in how models converge to an error minima through the process of gradient descent. We specifically selected the three most effective activations, by general consensus, which were the Rectified Linear Unit, Tanh, and the Leaky Rectified Linear Unit. We chose Tanh as the most basic, non-linear activation function which suffers from the vanishing gradient problem in the flat levels of the curve. We chose the Rectified Linear Unit as a mode comparison to observe how the absence of the vanishing gradient in the positive sum region will improve the model's learning capabilities. The last function we decided to test with was the Leaky Rectified Linear Unit, which completely resolves the zero gradient problem presented with the regular Rectified Linear Unit. Overall, we wanted to select a variety of functions each with unique properties and observe their performance effects.

The last parameter we selected was the model optimizer which dictates how gradients derived from the activation function are used during the training process. PyTorch provides an array of optimizer

functions, and similar to how we selected our activation functions, we wanted a variety of algorithmic implementations. We decided to test SGD, Adam, and RMSProp. SGD provides a cheap (due to batch property), baseline algorithm whereas Adam integrates second moments and dynamically adaptive learning rates for each network weight for efficient convergence. We introduced RMSProp for diversity purposes as it leverages recent gradient magnitude to adjust the learning rates independently and ensures balanced learning rates.

After observing the effects of each parameter on the model performance, we can derive an optimal range of values for each. Using the range, we decided to leverage a 3rd-party library called Optuna to perform hyperparameter tuning to vary each parameter within the range bounds we've set. Since it is computationally expensive to derive the relationships and interplay between unique parameters when their values change, the library can perform such an experiment for us through multiple trials of different model configurations. We set the loss function in the Optuna experiment to be MSE, where the library will then output the set of parameter values that resulted in the lowest MSE. Since the dataset is small, we decided to enforce a shuffle split during each experimental trial to ensure each subset is trained and tested on.

5 Preprocessing Results

5.1 Data Cleaning

After observing that the distribution of null-fields is relatively small and insignificant, we dropped all instances with any null fields. By doing this, we removed 33 instances in our original dataset of 421 instances. We subsequently removed outliers in the new dataset, which led to 16 instances being removed because they had relatively extreme data that was outside of our z-score standard of 3.

5.2 Feature Scaling

Utilizing StandardScaler().fit_transform(), we successfully standardized our data so that each column has a mean of 0 and a standard deviation of 1. Now we can be confident that if one feature dominates another it is not because of a difference in their scales.

5.3 Feature Selection

After testing each base model, the following results display the RMSE after evaluation on the condensed feature set.

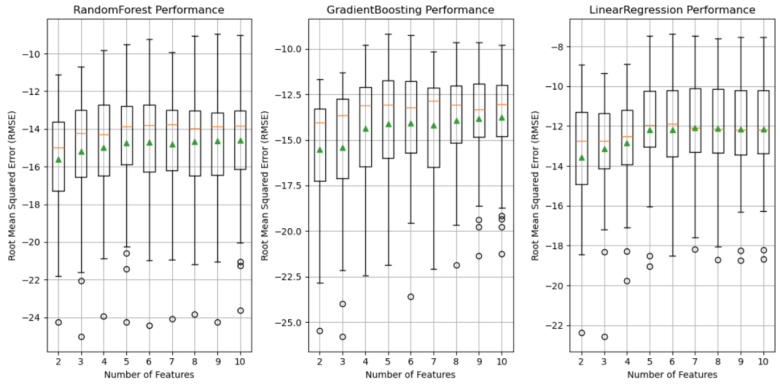

Figure 1: Condensed Feature Set Performance

The lowest RMSE was achieved through a Linear Regression model as the estimator and filtering for 7 features in the final feature set. The features selected and their respective rank was shown below.

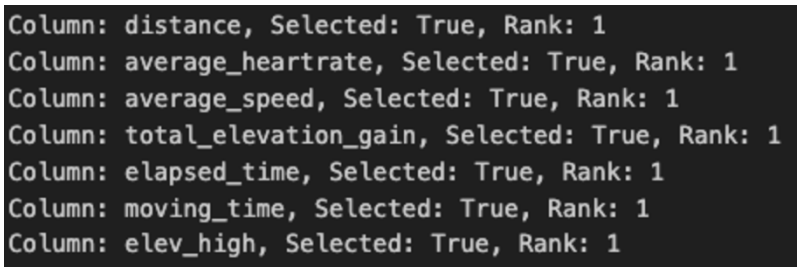

Figure 2: Features Selected

After performing RFE, we filtered out the unselected features and proceeded to train the models with the fully preprocessed dataset.

6 Model Selection Results

6.1 Regression

When iterating through our wide range of degrees [3, 5, 8, 10, 12], we found that the lowest testing RMSE for each degree's best model was generally very high across the board. The lowest testing RMSE was 63.4 with a degree of 3 and a regularization λ of 19.3. Our observation on this first set of degrees is that the testing RMSE increased as the degree was increased, which hints to overfitting with higher degrees. Since our current lowest testing RMSE is still relatively high at 63.4, we decided to try a new set of degrees that are smaller to prevent the overfitting we saw before. We conducted the same procedure but now on a new set of degrees [1,2,3,4]. We observed a significantly smaller testing RMSE in the best models with a degree of 1 and 2. The model with a degree of 2 and regularization λ of 1.5 had a slightly smaller RMSE of 13.5 as shown in Figure 4, so we decided to use this combination as our final model for regression. Since this model with a degree of 2 gives good results, we are able to conclude that our dataset is relatively simple but still contains some non-linear relationships which is why the degree of 2 has better results than a degree of 1. Our dataset is not extremely complex and non-linear though, which is why the larger degrees tended to give worse results and overfit.

6.2 Support Vector Regression

After adjusting the regularization C and epsilon ϵ for our SVR model with a linear kernel, we created a plot showing RMSEs for various parameter combinations as shown in Figure 5. We found that the best combination of regularization C and epsilon ϵ is with C being 215.4, epsilon being 0.02, resulting in a testing RMSE of 10.2. Figure 5 shows that epsilon ϵ has almost no impact on the RMSE performance of our SVR model. Instead, RMSE changes based on what regularization C we choose. As C decreases, so does our RMSE. Thus, our model was able to naturally stay within the margin and did not have to be heavily penalized to do so. This implies that our dataset is relatively simple, as our SVR model was able to decipher a linear relationship with our data and do so while staying within our margin as defined by epsilon ϵ. The fact that epsilon ϵ has an insignificant impact on RMSE could be because our data is so inherently simple and linear that the linear SVR is able to fit the data and thus no margin of allowed error is needed.

After varying the regularization C and gamma γ for our SVR model with a Gaussian Radial Basis function kernel, we created a plot showing RMSEs for various parameter combinations as shown in Figure 6. We found that the best combination of regularization λ and gamma γ is with λ being 1000 and γ being 0.02, resulting in a testing RMSE of 14.9. These metrics all point to the model tending to overfit. Regularization λ was high to ensure the model stays within the margin, and gamma ϵ was low, meaning that the resulting model was very narrow and prone to overfitting. Combining all of this with a higher testing RMSE compared to the SVR model using a linear kernel, it is evident that the linear function was better able to fit the data better than the Gaussian Radial Basis function that required higher dimensions. This was possible as our data is relatively linear and the SVR model with a linear kernel is sufficient.

It must be stated why our SVR model with a linear kernel and linear regression model differ in their testing RMSEs. This can be due to a variation in the parameters used, but it is largely attributed to the difference between their loss functions. Linear regression utilizes mean squared error, which depends on the difference between the predicted and targeted values, as its loss function. On the other hand, the SVR model with a linear kernel uses a loss function that depends on penalizing predicted values that lay outside of the margin specified by epsilon ϵ. The difference in their testing RMSEs can mainly be attributed to the difference in their loss functions, which impacts how they treat errors and learn.

6.3 Multi-Layered Perceptron

After varying the learning rate (lr = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]) and plotting the training and validation RMSEs, we noticed a convex function. The minimal error resided between the learning rate range of 0.001 to 0.1, where the validation RMSE was consistently less than the training RMSE and therefore indicates a lack of overfitting. The plot can be found in Figure 7.

Varying the number of neurons in the hidden layer (neuron_counts = [8, 64, 256, 1024, 4096, 16,384, 65,536, 131,072]), the resulting RMSE plot revealed a plateau in performance as the neuron count increased which aligned with our predictions. The validation RMSE was smaller than the training up until the plot began to plateau at approximately 65,536 neurons, in which after the RMSE discrepancy began widening. We decided to take the neuron count range smaller than 65,536 as the range of interest. The plot can be found in Figure 8.

Using a base neuron count of 32, we varied the number of hidden layers between 0 to 7 layers at which after that point, the change in RMSE became flat. We observed a significant drop in the validation RMSE with 6 hidden layers, as shown in Figure 9. For the other layer counts, aside from no hidden layer, the validation RMSE was either close or below the training RMSE, justifying the entire testing range to be used during hyperparameter tuning.

To test the effect of different neuron-layer patterns, we used the list of patterns found in Figure 10. The resulting plot in Figure 11 reveals a consistently larger training RMSE. We observed significant performance improvements when the number of neurons decreased across a deep network of layers, as shown by the two drastic minimas in the plot, specifically when the early layers had a larger number of neurons (64 to be exact).

After varying the activation functions, we noticed very minimal difference between the performance of each, as shown in Figure 12. The validation RMSE was also consistently larger than the training RMSE. This is likely due to the fact that the activation function is working with a shallow network (the base model which has one hidden layer), which restricts the impact the function has on the model's performance. The same could be said about the optimizer, as shown in Figure 13. Functions like RMSProp and Adam are far more potent in deeper architectures due to their inherent ability to adapt learning rates and manage gradient dynamics effectively across multiple layers. Adversely, these functions deliver suboptimal results when training an untuned network.

Using the observations made on the individual parameter contribution above, I configured Optuna's hyperparameter experiment. The following shows the hyperparameter space I experimented in.

```
# Hyperparameter space for Optuna.
learning_rate = trial.suggest_loguniform('learning_rate', 1e-2, 10)
num_layers = trial.suggest_int('num_layers', 1, 4)
hidden_neurons = [trial.suggest_int(f'neurons_l{i}', 8, 16384) for i in range(num_layers)]
activation_fct = trial.suggest_categorical('activation_function', [nn.ReLU, nn.Tanh, nn.LeakyReLU])
optimizer_type = trial.suggest_categorical('optimizer', ['SGD', 'Adam', 'RMSProp'])
```

Figure 3: Hyperparameter Space The Experiment Was Run In

The experiment generated the following optimal parameter values as shown in Table 2.

Parameter	Value
Learning Rate	0.021595541091483213
Hidden Layer Configuration	[9323, 14582]
Activation Function	Rectified Linear Unit
Optimizer	Adam

Table 2: Initial Experiment Optimal Value

After training and evaluating the model over multiple dataset folds through shuffle splitting, the model achieved an average test RMSE of 102.065. Unsatisfied with the result, we suspected the poor performance to be due to overfitting with the excessively large number of neurons per layer. I decided to limit the neuron count range to 256 instead of 16384 and the following parameter values were generated by Optuna as shown in Table 3.

Parameter	Value
Learning Rate	0.033287585471540114
Hidden Layer Configuration	[63, 51, 37, 57]
Activation Function	Rectified Linear Unit
Optimizer	Adam

Table 3: Experiment Optimal Value After Limiting The Neuron Count Range to 256

The model had a semi-decreasing neuron pattern which we observed to be the optimal configuration when testing the individual parameter effects on learning performance. Unfortunately, the model achieved an abysmal average test RMSE of 111.23, even worse than previously.

7 Conclusion

Throughout this report we attempted to create optimal machine learning models to predict burned calories while running by learning on our dataset. Since we developed our own dataset, a lot of preprocessing was required before testing our models. Preprocessing included cleaning the data for null fields and outliers, feature scaling to ensure standardization in scales, and feature selection to choose the most important features. We first utilized regression, varying the degree and regularization used. We found that the optimal regression model used a degree of 2, generating a testing RMSE of 13.5. We also worked with the Support Vector Regression (SVR) model, varying the kernel, regularization, epsilon, and gamma values. We determined that the best SVR model utilized a linear kernel and resulted in a testing RMSE of 10.2, while the complex Gaussian Radial Basis function was overfitting on our simple dataset. Overfitting was also a common trend with our MLP model, which we optimized by hypertuning each parameter such as the learning rate, hidden layer configuration, activation function, and optimizer. Our best MLP model led to a testing RMSE of 102.065, which again ended up being a too complex model that tried to find underlying non-linear relationships in our relatively simple dataset. Additionally, besides its large RMSE, the MLP model required much higher computational power than our linear models. We can conclude that for our data, the best option remains to be simple models such as the SVR model with a linear kernel since our data is mostly linear and does not require complex and excessively optimized modern networks like the MLP model. This report shows how critical it is to test various models on your data, as different datasets can have different underlying relationships which will affect each model's performance.

8 Citations

[1] C. Luff, "How many calories does running burn?," Verywell Fit, https://www.verywellfit.com/how-many-calories-does-running-burn-2911108 (accessed Dec. 4, 2024).

[2] R. A. Robergs and L. Kravitz, "Making Sense of Calorie-burning Claims.," University of New Mexico, https://www.unm.edu/~lkravitz/Article%20folder/caloricexp.html (accessed Dec. 4, 2024).

[3] "Scientist busts myths about how humans burn calories–and why | the nation ahead," Scientist Busts Myths About How Humans Burn Calories and Why, https://nationahead.com/2022/02/18/scientist-busts-myths-about-how-humans-burn-calories-and-why/ (accessed Dec. 4, 2024).

[4] P. H. Charlton et al., "The 2023 wearable photoplethysmography roadmap," Physiological measurement, https://pmc.ncbi.nlm.nih.gov/articles/PMC10686289/#:~:text=Photoplethysmography%20is%20a%20key%20sensing,activities%20like%20sleep%20and%20exercise. (accessed Dec. 4, 2024).

[5] "How can a Garmin watch calculate calories burned?," FitStrapsUK, https://fitstraps.co.uk/blogs/news/how-can-a-garmin-watch-calculate-calories-burned (accessed Dec. 4, 2024).

[6] "Recursive feature elimination (RFE): Working, Advantages Examples," Analytics Vidhya, https://www.analyticsvidhya.com/blog/2023/05/recursive-feature-elimination/ (accessed Dec. 4, 2024).

[7] J. Brownlee, "Recursive feature elimination (RFE) for feature selection in Python," MachineLearningMastery.com, https://machinelearningmastery.com/rfe-feature-selection-in-python/ (accessed Dec. 4, 2024).

[8] T. Deng, "Effect of the Number of Hidden Layer Neurons on the Accuracy of the Back Propagation Neural Network," ResearchGate, https://www.researchgate.net/publication/377755645_Effect_of_the_Number_of_Hidden_Layer_Neurons_on_the_Accuracy_of_the_Back_Propagation_Neural_Network (accessed Dec. 4, 2024).

9 Appendices

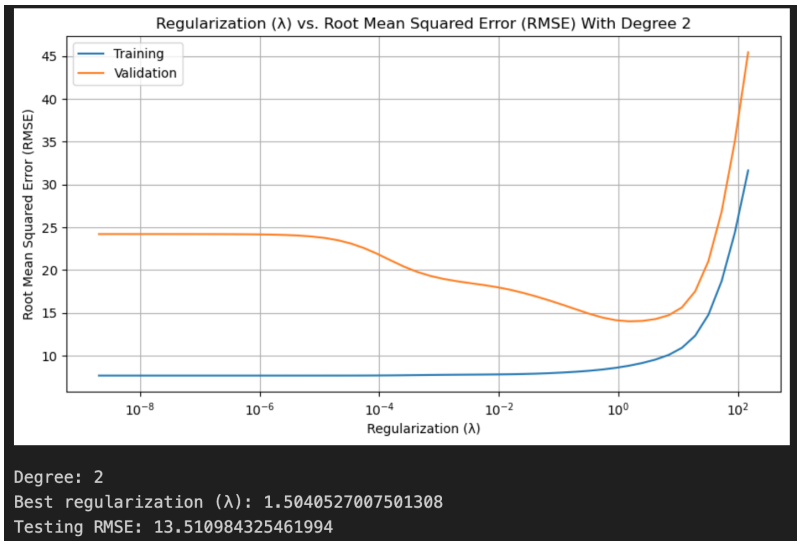

Figure 4: Optimal Regression Performance With a Degree of 2

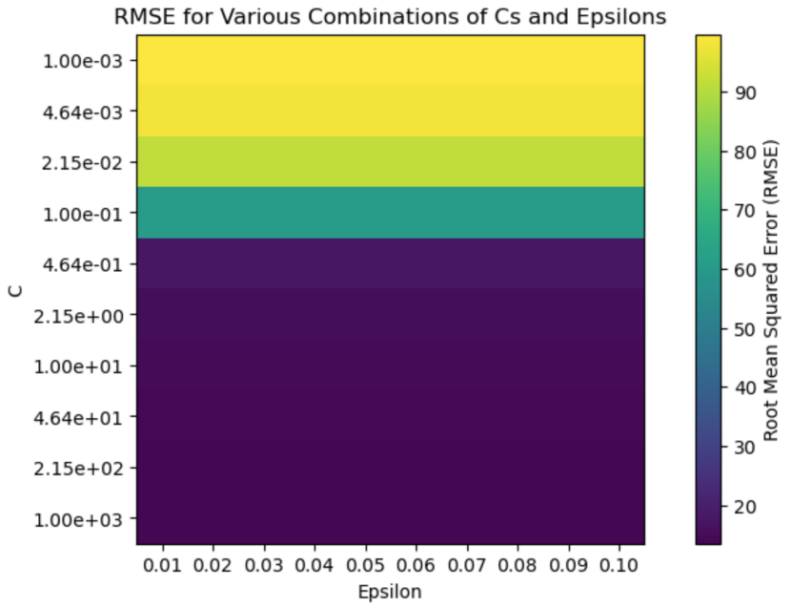

Figure 5: SVR Performance With a Linear Kernel While Varying Epsilon and C

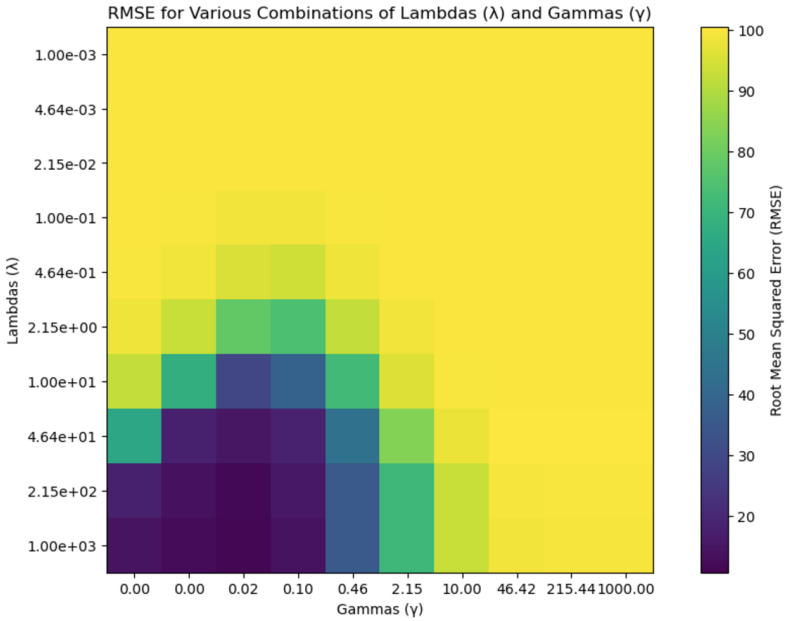

Figure 6: SVR Performance With a Gaussian Radial Basis Function Kernel While Varying Gamma and Lambda

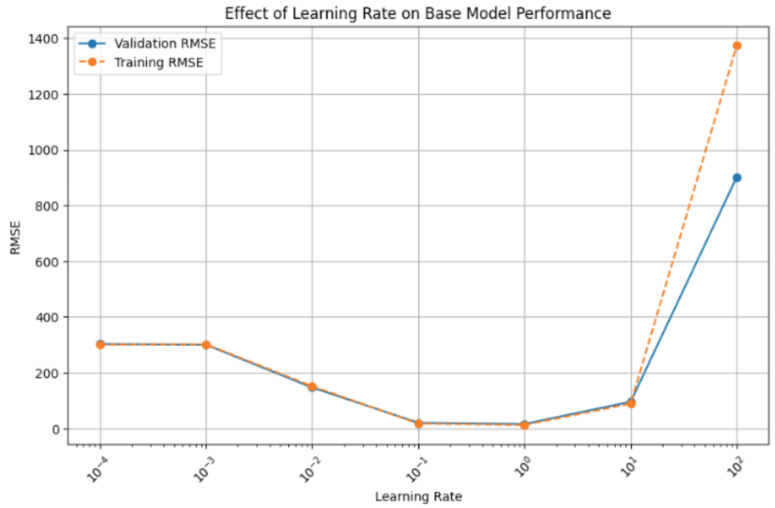

Figure 7: Effect of Learning Rate on Base Model Performance

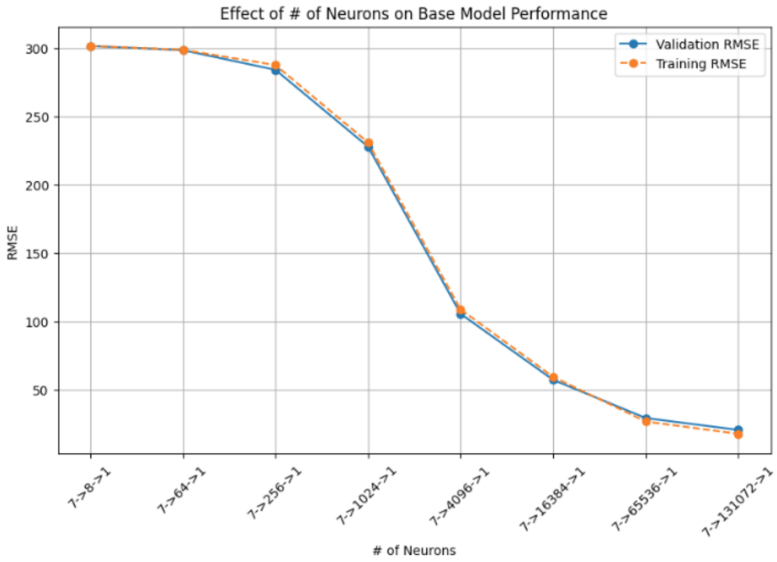

Figure 8: Effect of # of Neurons on Base Model Performance

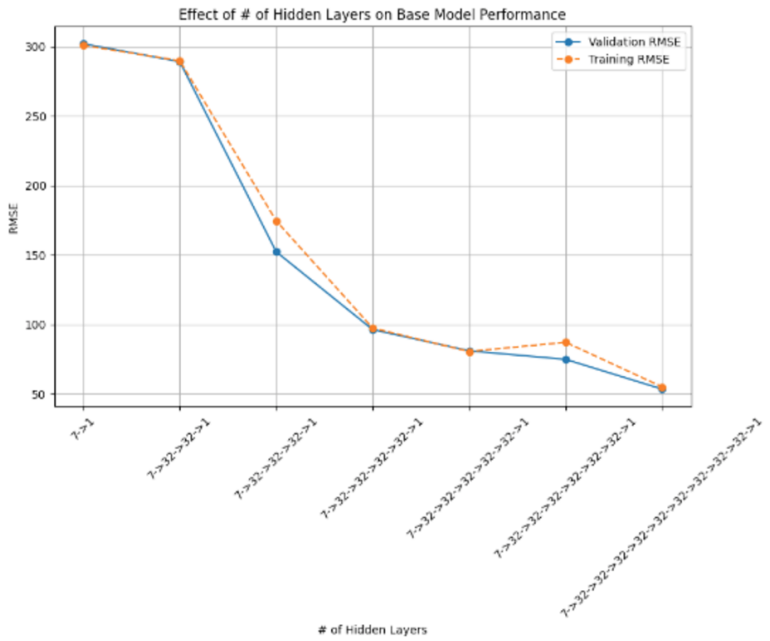

Figure 9: Effect of # of Hidden Layers on Base Model Performance

```
layer_configs = [
    [X_train.shape[1], 16, 32, 64, 1],
    [X_train.shape[1], 16, 32, 16, 1],
    [X_train.shape[1], 32, 16, 8, 1],
    [X_train.shape[1], 32, 16, 32, 1],
    [X_train.shape[1], 8, 64, 32, 16, 8, 1],
    [X_train.shape[1], 8, 24, 12, 6, 1],
    [X_train.shape[1], 8, 8, 1],
    [X_train.shape[1], 16, 16, 16, 1],
    [X_train.shape[1], 64, 32, 16, 8, 1],
    [X_train.shape[1], 8, 1]
]
```

Figure 10: Various Layer Configurations Used

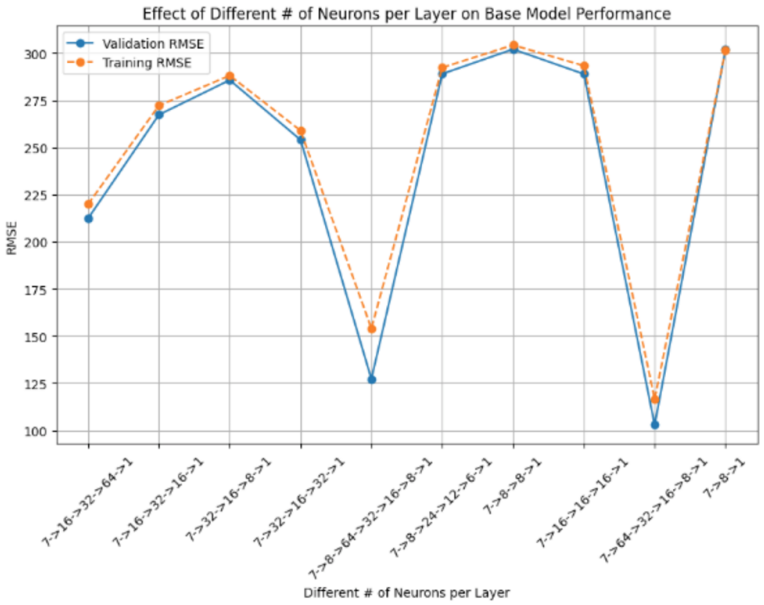

Figure 11: Effect of Different # of Neurons per Layer on Base Model Performance

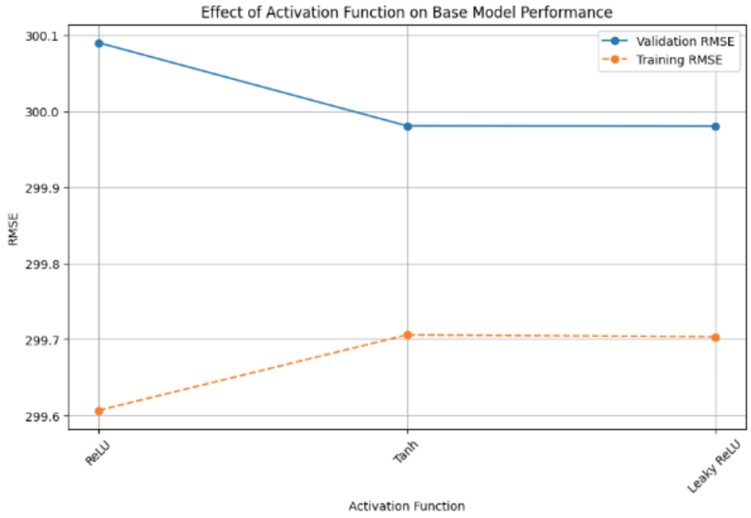

Figure 12: Effect of Activation Function on Base Model Performance

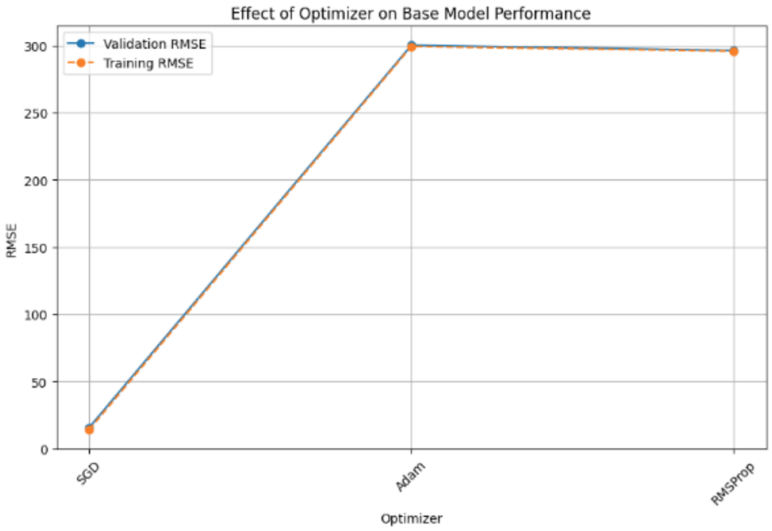

Figure 13: Effect of Optimizer on Base Model Performance