ECE133A Winter 2023 Final Project

Turner Kaminski 605574293 Justin Hu 205514102

March 25, 2023

Link to GitHub repository: Github
URL to GitHub repository `https://github.com/TurnerKaminski/ECE133A_Final_Project`

1 Introduction

1.1 Dataset Description

The dataset we will be working with is made up of two files. The first file is an 21263x88 matrix. Each row each belong to one superconductor, and the first 86 columns correspond an individual element commonly found in superconductor materials, such as oxygen, the 87th column is the critical temperature, and the 88th column is the written chemical formula. Overall, this file allows for storing the chemical breakdown and formula for each superconductor, as well as the corresponding critical temperature.

The second file is a 21263x82 matrix. The rows correspond to the same superconductors as the first file, so it is important that the ordering stays the same, and the first 81 columns correspond to features of each superconductor, and the 82nd is the critical temperature of each. The 81 features correspond to characteristics of 8 main categories, which include: Thermal Conductivity, Atomic Radius, Valence, Atomic Mass, First Ionization Energy(fie), Fusion Heat, Density, and Electron Affinity. Some examples of the characteristics include mean and standard deviation.

When reading the background on the data-set, it was made clear that outliers and incorrect data had already been removed, so it was unnecessary to clean the data-set in that way. As there are 81 features, it would be a safe guess that many are redundant/unnecessary, however without a strong background on superconductors, it is in our best interest to keep all features for the moment and use our data analysis to determine which are the best, rather than theoretical knowledge.

1.2 Problem Description

Using the features included in our data set, we plan to predict the critical temperature of a given superconductor. This means we will be doing a regression model, rather than classification. Due to a lack of formal knowledge on superconductors, will we be using purely data analysis to determine which features work best.

1.3 Project Plan

The first step is to load in our data and to understand it as well. Our GitHub repository shows the initial loading of our data, and some simple plotting and visualizing the data to help us understand it. Following this, it will be helpful to perform some processing on our data, such as different types of grouping, this will allow us to analyze each feature of the superconductors based on different grouping, for example, grouping by particular elements being present in the superconductor. Then we must find which features will best allow us to predict critical temperature, aka we must find which features have the highest correlation. Following these steps should allow us to make a preliminary model for predicting the critical temperature of a given superconductor.

Of course, throughout this process it will be expected to clean our code and optimize it as were dealing with a large data-set, and we will be performing lots of different analysis on it.

Rather than assigning roles to each student, we plan on working together throughout the project, so that we both get experience of performing the entire project.

2 Data Structure

2.1 Standardization

This section is pretty simple, Matlab allows us to just use normalize() to standardize each feature of our data set. Below is the mean and standard deviation for every feature.

	Feat1	Feat2	Feat3	Feat4	Feat5	Feat6	Feat7	Feat8	Feat9	Feat10	Feat11	Feat12
Mean	4.11	87.56	72.99	71.29	58.54	1.17	1.06	115.60	33.23	44.39	41.45	769.61
STD	1.44	29.68	33.49	31.03	36.65	0.36	0.40	54.63	26.97	20.04	19.98	87.49

	Feat13	Feat14	Feat15	Feat16	Feat17	Feat18	Feat19	Feat20	Feat21	Feat22	Feat23
Mean	870.44	737.47	832.77	1.29	0.93	572.22	483.52	215.63	224.05	157.98	134.72
STD	143.28	78.37	119.77	0.38	0.33	309.61	224.04	109.97	127.93	20.15	28.80

	Feat24	Feat25	Feat26	Feat27	Feat28	Feat29	Feat30	Feat31	Feat32	Feat33	Feat34
Mean	143.45	120.99	1.27	1.13	139.33	51.37	51.60	52.34	6.11e3	5.27e3	3.46e3
STD	22.09	35.84	0.38	0.41	67.27	35.02	22.90	25.29	2.85e3	3.22e3	3.70e3

	Feat35	Feat36	Feat37	Feat38	Feat39	Feat40	Feat41	Feat42	Feat43	Feat44	Feat45
Mean	3.11e3	1.07	0.86	8.66e3	2.90e3	3.42e3	3.32e3	76.88	92.72	54.36	72.42
STD	2.98e3	0.34	0.32	4.10e3	2.40e3	1.67e3	1.61e3	27.70	32.28	29.01	31.65

	Feat46	Feat47	Feat48	Feat49	Feat50	Feat51	Feat52	Feat53	Feat54	Feat55	Feat56
Mean	1.07	0.77	120.73	59.33	48.91	44.41	14.29	13.85	10.14	10.14	1.09
STD	0.34	0.29	58.70	28.62	21.74	20.43	11.30	14.28	10.07	13.13	0.38

	Feat57	Feat58	Feat59	Feat60	Feat61	Feat62	Feat63	Feat64	Feat65	Feat66	Feat67
Mean	0.91	21.14	8.22	8.33	7.72	89.71	81.55	29.84	27.31	0.73	0.54
STD	0.37	20.37	11.41	8.67	7.29	38.52	45.52	34.06	40.19	0.33	0.32

	Feat68	Feat69	Feat70	Feat71	Feat72	Feat73	Feat74	Feat75	Feat76	Feat77	Feat78
Mean	250.89	62.03	98.94	96.23	3.20	3.15	3.06	3.06	1.30	1.05	2.04
STD	158.70	43.12	60.14	63.71	1.04	1.19	1.05	1.17	0.39	0.38	1.24

	Feat79	Feat80	Feat81	Feat82
Mean	1.48	0.84	0.67	34.42
STD	0.98	0.48	0.46	34.25

Table 1: These are the mean and standard deviations of every feature in our data set.

2.2 *k*-means Clustering

For performing k-means, we made use of the kmeans() Matlab function. Before performing kmeans, we remove the actual critical temperature feature to remove any chance of the clustering biasing. We also are using standardized data found in the previous section to prevent any biasing due to scale of the particular feature's data. Since we are also using a clean data set, we don't need to worry about any egregious outliers.

Now, when performing kmeans we tested values of k from 1 to 10. This assumes that the optimum value of k is within these values, however any value of k above of 10 would most likely be suggesting over-fitting our data, so it is not worth looking at.

To analyze the effectiveness of the algorithm, and help us determine the best value of k, we find the root mean squared error between the predicted critical temps and the actual critical temps. The only flaw with this method of looking at values of k is that we can not determine whether we have over-fitting from just the RMSE, but since we are only testing values of k from 1 to 10, this should not impact our results.

From this method, the smallest RMSE value was most consistently at a k value of 6. This number of clusters is a reasonable assumption based on our data set, and makes intuitive sense. So based on this testing, k = 6 is the optimum number of clusters for our data set.

However, the RMSE values were not at a desirable value for use in further sections of our prediction. The most likely cause for this, purely looking at results is that our features have non-linear relationships. K-means clustering assumes that the clusters are spherical and well separated, which can prevent it from being a useful algorithm for particular data-sets. This also makes sense from a logical standpoint, as we know that our features are not simply a linear relationship to the critical temperature, from a background level of knowledge. So we can safely say that k-means clustering is not an effective algorithm for our data set.

2.3 SVD

Performing the SVD consisted of just using the svd() Matlab function on the standardized matrix of our data set. The highest entry has a value of around 825, while the smallest value was about 1.02. Since higher value entries are more significant when it comes to singular value decomposition. From our set of SVD values, 28 out of 82 have a value higher than 50, 18 out of 82 have a value higher than 100, and 9 out of 82 are higher than 200.

Because SVD values are an exponential decay, we can see that out of our 82 SVD values, SVD 65 and below are less than 1 percent of our highest SVD value. At SVD 42 and below are less than 3 percent of our highest SVD value. We can comfortably say that below 1 percent of our initial value is going to be redundant and no longer provide significant value, and one can argue that below 3 percent won't provide significant value either.

2.4 Correlation Matrix

Finding the correlation matrix consisted of using the corr() function on the standardized matrix. From the correlation matrix, we also extracted a single column that is the correlation of the 81 features against the 82nd feature (the critical temperature). From this, we were able to sort them in descending order to find which features from the original data set correlated the most with the critical temperature, aka what we want to predict with our model.

Many features are associated with the same type of data, for example having the range, mean, and standard deviation of thermal conductivity, all be separate features. Thanks to our correlation matrix we are able to see which characteristics from each main category is most correlated. The 8 main categories include: Thermal Conductivity, Atomic Radius, Valence, Atomic Mass, First Ionization Energy(fie), Fusion Heat, Density, and Electron Affinity. The order of the main categories listed above is also the rough order of the correlation between them and critical temperature. For example, our most correlated characteristic is the standard deviation of thermal conductivity, and 3 of the top 5 correlated features are a characteristic of the thermal conductivity coefficient. Following thermal conductivity, the next most two common main categories with multiple characteristics appearing with high correlations are atomic mass, and valence (number of chemical bonds) of the superconductors.

Of our 81 features, 25 of them have a correlation above 0.5 against the critical temperature. Figure 2 is the plot of the heat map coinciding with the correlation matrix.

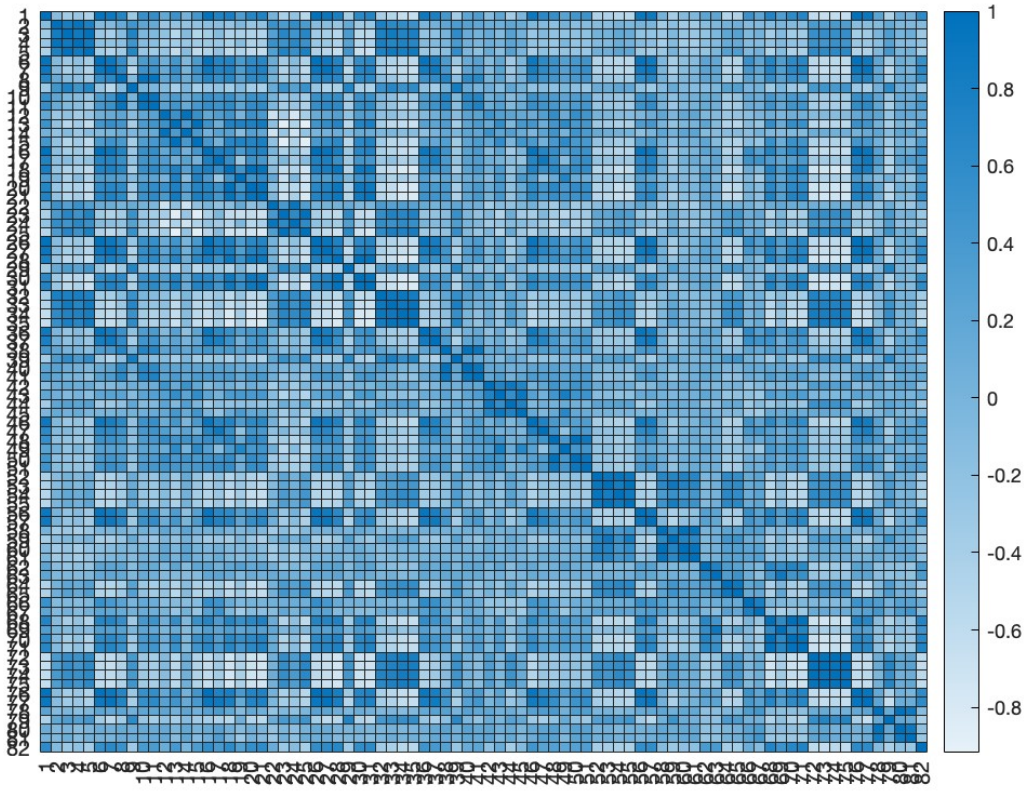

Figure 1: The heatmap from the correlation matrix

3 Model Evaluation and Prediction

3.1 Linear Model

For an initial linear model, we are simply using our original data set with the features standardized, and the target feature removed. When creating this model, we partitioned the data into 10 folds taking advantage of the cvpartition() function in Matlab. We then loop through the folds, and for each fold take our partitioned data and separate it into training and testing sets. Then we define our linear model, using the fitlm() Matlab function, and then use it to predict our target values. Once we've done this, we find our RMS error and the model parameters, storing these for each fold individually. The table below displays the RMS errors for each of the 10 folds.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
RMSE	0.513	0.518	0.495	0.496	0.508	0.525	0.511	0.538	0.526	0.513

Table 2: These are the RMSE values for the 10 folds on the inital linear model

Because our target feature was normalized to standard deviation = 1, our average RMSE value being .514 tells us that there is still much room for improvement as that isn't a particularly powerful predicting model.

3.2 Feature Engineering

The first idea we had for feature engineering was to reduce the dimensions of our features, as we have 81 features and from our previous analysis and many had low correlation with our target output. However, when removing the 10 lowest correlated features, the average RMSE increased from .514 of our previous model, to about .520. While this is not a super significant change in performance, we do note that our feature matrix is standardized, meaning that this change is more significant than it initially appears. Since it increased rather than decreased the RMSE we no longer pursued dimensional reduction. Looking back, we hypothesize that the reduction of features, even one with low correlation, may lower the predicting power of our model since the low correlation features do not actually interfere with the higher correlated ones.

Following this, we decided to try some logarithmic and polynomial feature transformations to add more features. Overfitting was not a worry because considering we have over 21000 samples, around 100-120 features should not create an issue. Due to a lack of background knowledge on superconductors and which features would be optimal for these transformations, we simply compared the correlation of the features to the target feature before and after transformation, and if the correlation improved, add that newly transformed data as a new feature to our data set. Based on some scatter plots of specific features vs target we had looked at, logarithm seemed like the most obvious option so we chose to do that first. We also performed a squaring transformation. These two transformations ending up adding 45 new features combined. Simply performing these two transformations and then running our linear model procedure from part 3.1 to this new data set lowered our model's average RMSE from .514 to .495. Once again, due to our RMSE being standardized, this is a fairly decent improvement to our model from a relatively small number of additional features added. Below is a table of the RMSE for each respective fold of the data set with both loged and squared features added.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
RMSE	0.505	0.492	0.491	0.502	0.502	0.483	0.496	0.474	0.490	0.520

Table 3: These are the RMSE values for the 10 folds on the engineered linear model

With our original features as well as the squared and logged features, we went from 81 features up to 126 features.

3.3 Regularization

For the first part of this section, we are performing regularization on our linear model from the previous section. This involved splitting our data set into test and training sets then testing the RMSE for both the training and testing sets across a range of lambda values. This just meant looping over a range of lambda values and using the fitrlinear() Matlab function which works the same as the fitlm() function from the previous section but allows for regularization to be added. In figure 2, you can see the plot of the lambda values vs the RMSE for both the training and test sets.

When choosing the best lambda value, our RMSE actual increased for the regularized model from our normal linear model, from about .49 up to .56. This suggested that our model from part 3a was underfitting the data so that regularization provides no benefit.

Following this, we revised our feature engineering from the previous step by adding random features to the regularization model to help combat underfitting. This involved using the equation

$$f_i(x) = max(0, r_i^T x + s_i)$$

where r and s are randomly generated and x is one of the original features. Adding these random features increases the linear regression power since you have much more features to work with. We have about 21000 samples in our dataset, and overfitting generally occurs when you have over 20 percent the number of features to samples. Since we have 126 features before adding random variables and 20 percent of our samples would be about 4200, we decided to try adding 4000 randomly generated features so the regularization parameter had something to work with.

After adding these random features, it was able to decrease the RMSE of the regularized model from around .56 down to .43. This means that the regularized model with random features was able

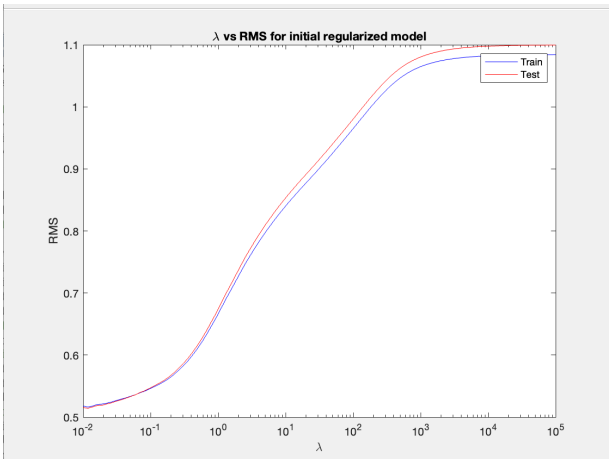

Figure 2: Lambda vs RMS for the first regularized linear model

to improve upon the original linear model with no regularization added. However, to us this is still a rather disappointing RMSE given that we now have over 4000 features.

Figure 3 shows the same plot as figure 2, but for the regularized model with random features.

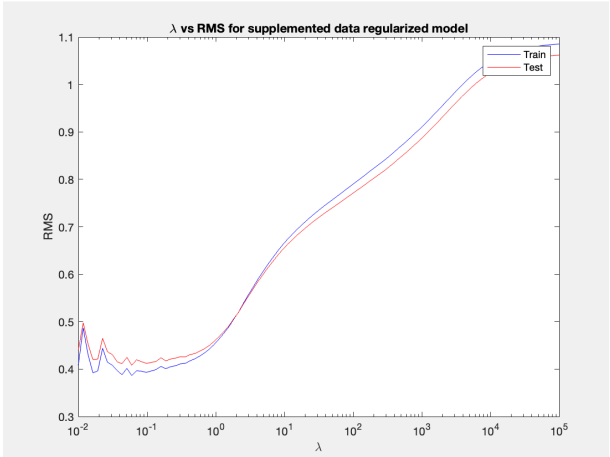

Figure 3: Lambda vs RMS for the regularized linear model with random features

Other features could have also been created using the interaction features our original features, but due to a lack of background knowledge on the topic of superconductors, the addition of random features was the most suitable option for increasing the number of features our model had to work with. The addition of random features should theoretically also increase the generality of our model, since the model isn't just perfectly fitting itself to known values and superconductors.

3.4 Non-linear Models

When performing least squares model fitting, we are minimizing a cost function defined by

$$\sum_{i=1}^{n}(\hat{f}(x^{(i)};\theta) - y^{(i)})^2$$

In this section, we are essentially doing the same thing as we did in part 3a (with no feature engineering or regularization) except that we will be trying non-linear functions for $\hat{f}(x^{(i)};\theta)$ instead. Using the generalized least squares curve fitting function lsqcurvefit() in Matlab, we are able to specify a custom prediction function instead of the default linear form as the first argument to the function call. Then, we run through the same procedure as we did in part 3a, dividing our data set into 10 folds and

comparing the RMSE between different non-linear functions as well as storing the parameters we obtained for future reference.

To generate a non-linear prediction function we used two general approaches, modifying the standard linear model by directly applying a non-linear transformation and coming up with an original non-linear prediction function from scratch. The non-linear modifications we tried were squaring, cubing, logarithmic, Q function (unit normal tail area), and inverse tangent. For illustration, the least squares objective for our logarithmic model would be

$$\sum_{i=1}^{n}(log(X\theta) - y)^2$$

where $X\theta$ is the standard linear least squares function. Below are tables representing the RMSE for each fold of data for the 5 non-linear models obtained through modification of standard linear model.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10	Average
RMSE	0.858	0.859	0.869	0.859	0.839	0.861	0.843	0.862	0.852	0.865	0.86

Table 4: Square model. Relatively long run-time

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10	Average
RMSE	0.0.559	0.569	0.550	0.562	0.561	0.568	0.566	0.550	0.561	0.549	0.56

Table 5: Cube model. Surprisingly short run-time

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10	Average
RMSE	1.492	1.445	1.544	1.494	1.393	1.427	1.463	1.494	1.425	1.395	1.46

Table 6: Logarithmic model. Very poor performance

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10	Average
RMSE	0.764	0.773	0.772	0.778	0.774	0.763	0.762	0.757	0.774	0.768	0.77

Table 7: Q function model. Actually takes forever

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10	Average
RMSE	0.538	0.529	0.537	0.514	0.521	0.508	0.520	0.515	0.527	0.534	0.52

Table 8: Inverse tangent model. Surprisingly good?

Just as experimentation we also came up with two more completely original crazy looking non-linear models, since the only requirement for the prediction function is that it takes the input matrix and outputs a prediction vector.

Original Model 1:

$$\hat{f}(x^{(i)};\theta) = \|\theta_1 * e^{(sin(\theta_2 * x^{(i)}))} * (x^{(i)})^{\theta_3}/\theta_4\|/\theta_5$$

Original Model 2:

$$\hat{f}(x^{(i)};\theta) = \sum_j [|log(\theta_1 * x^{(i,j)})| * \theta_2^{(x^{(i,j)})}] + \theta_3$$

Below are tables representing the RMSE for each fold of data for the 2 original non-linear models.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10	Average
RMSE	1.000	0.989	0.980	0.986	1.003	1.022	1.006	0.994	1.002	1.012	1.00

Table 9: Original Model 1

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10	Average
RMSE	1.215	1.252	1.205	1.201	1.227	1.232	1.216	1.210	1.241	1.228	1.22

Table 10: Original Model 2

Coming up with a crazy looking non-linear function isn't difficult, but coming up with one that the computer could solve in a reasonable amount of time was, which is why we couldn't produce results for more of these functions. Most of the original non-linear functions we tried could not complete a single fold in 6 minutes, meaning that the total run time would almost certainly exceed an hour.

Looking at these results from the non-linear models we tried, none of them could surpass the simple linear model from part 3a. The best performance came from applying an inverse tangent on the standard linear function, with an average RMSE difference of only 0.01. We hypothesize that this is because of arctan() basically being a linear function that is damped for larger magnitudes of input, which the parameters could adjust for. The second best performance came from the cube model with an average RMSE difference of 0.05 from the linear model, which we hypothesis is because cubing allows for negative values, giving the model more flexibility compared to the square model, for example. While the Q function model did not perform as badly as we anticipated, it didn't do well either, and the log model along with our original non-linear models performed extremely poorly. It seems that given our limited knowledge about the scientific principles behind our data set and a shortage of features with sufficient predicting power, these simpler non-linear models are not able to provide us with better performance compared to the simple linear model. Outside of using a machine learning algorithm as the ultimate general purpose non-linear model, regularization with additional random features seems to be our best bet at further improving the performance of our models.

4 Complexity Analysis

4.1 Number of Flops

Flop computation for 3a model:

From Lecture 8 Page 16, using QR factorization to solve a least squares problem takes $2mn^2$ flops, where m is rows and n is columns. In our model, for each of the 10 folds, m = 19137 and n = 81. Thus, total flops to obtain parameters for 3a would be 2511157140 (2.51 billion) flops. Now that we have the 81 parameters, we will need to perform another matrix-vector multiplication in order to get the predicted values of y. Matrix-vector multiplication is 2mn flops, m = 2126, n = 81, flops = 344412. Calculating RMSE requires 3 flops per element so 2126*3 = 6378 in total. Repeat this for 10 folds, so 3507900 flops (3.51 million).

Flop computation for 3b model:

3b model is really just the same thing as 3a but with a different number of features, aka different number of n. We had 126 features in the improved model so n = 126, total flops to obtain parameters for 3b is 6076380240 flops (6.08 billion). Calculating RMSE requires 10*(2mn + 3n) flops = 5421300 flops (5.42 million)

Flop computation for 3c model:

In part 3c we are doing tikhonov regularization with a total of 4126 features. The stacked matrix will therefore have dimensions m = 21263 + 4126 and n = 4126. Note that we are only doing this for 1 fold instead of 10. In total we will get 723957350776 flops (724 billion).

With 4126 parameters, test & RMSE will take 2*21263*4126 + 3*21263 = 175526065 flops (176 million).

Flop computation for 3d model:

Flop computation for part 3d becomes a lot more involved. In part d we are using the levenberg-marquadt method to solve nonlinear least squares. Each update involves computing f(x) and Df(x) as well as solving a regularized least squares equation.

Levenberg-marquadt step is computed as

$$\Delta x^{(k)} = -(Df(x^{(k)})^T Df(x^{(k)}) + \lambda^{(k)} I)^{-1} Df(x^{(k)})^T f(x^{(k)})$$

we can write the linear equation as

$$-(Df(x^{(k)})^T Df(x^{(k)}) + \lambda^{(k)} I) * \Delta x^{(k)} = Df(x^{(k)})^T f(x^{(k)})$$

Here, $f(x^{(k)})$ has dimensions 1x19137 and $Df(x^{(k)})$ has dimensions 19137x81

For our best model aka tan^{-1} modification of linear model, computing $f(x^{(k)})$ requires 19137 vector multiplications of 81 dimensional vectors, plus 19137 tan^{-1} computations of the 19137 results. Vector inner product is known to be 2n flops. We will estimate tan^{-1} using taylor series expansion with 3 terms (up to x^5) which will be 10 flops (a gross underestimate, but we can't really do much better). Thus, total flops computing $f(x^{(k)})$ once is 19137*2*81 + 10*19137 = 3291564 flops.

Computing $Df(x^{(k)})$ requires computing the gradient, defined by a $\mathbb{R}^{81} \to \mathbb{R}^1$ function for all 19137 outputs, each of which contains 81 derivatives. Using the simple calculus approximation $\frac{f(x+h)-f(x)}{h}$, we will need to compute $f(x^{(k)})$ twice and perform subtraction and division for each set of 81 derivatives. We already know flops of computing $f(x^{(k)})$ from above. Thus, flops for computing $Df(x^{(k)})$ once is 81*[3291564*2 + 2*19137] = 536333562 flops.

Now, we will compute the flops required to setup the linear equation above. Matrix multiplication takes 2mnp flops, so flops to compute the first matrix is 2*81*19137*81 + 81^2 = 251122275 flops. Flops to compute the right side vector is 2*81*19137 = 3100194 flops. Flops to solve linear equation is $2 * 81^3$ = 1062882 flops (the cheapest step so far). Finally, there is another 81 flops required to actually apply the levenberg-marquadt step to our parameter vector.

Adding up all of the flops required for a single iteration, we get 3291564 + 536333562 + 251122275 + 3100194 + 1062882 + 81 = 794910558 flops (795 million).

Using the output option for lsqcurvefit, we find that the total number of iterations to compute the best non-linear fit model for all 10 folds is 87. Thus, total number of flops required to obtain non-linear model is 87*794910558 = 69157218546 flops (69.2 billion).

Computing $f(x^{(k)})$ for 2126 test data points takes 2126*2*81 + 10*2126 = 365672, plus 2126*3 for RMSE x10 folds for a total of 3720500 flops (3.72 million).

Here is a table summarizing all of the computations performed above:

	Obtaining Model	Test & RMSE	Total Flops
3a model	2.51 billion	3.51 million	2.51 billion
3b model	6.08 billion	5.42 million	6.08 billion
3c model	724 billion	176 million	724 billion
3d model	69.2 billion	3.72 million	69.2 billion

Table 11: Flops Summary

Since we normalized our data set to a standard deviation of 1, we will be comparing the complexity and performance of our model by calculating flops per 0.01 RMSE reduction from 1 as a sort of performance efficiency measure. Here is a table comparing the complexity of our models to their RMSE:

	Average RMSE	Complexity (flops)	Efficiency
3a model	0.514	2.51 billion	51.6 million flops/reduction
3b model	0.495	6.08 billion	120 million flops/reduction
3c model	0.43	724 billion	12.7 billion flops/reduction
3d model	0.524	69.2 billion	1.45 billion flops/reduction

Table 12: Complexity v.s. RMSE

Even though the 3c model has the best RMSE, it is incredibly complex, requiring almost x300 the flops required for the 3a model, making it the least efficient model if we care a lot about performance. Perhaps unsurprisingly, the simple 3a linear model turns out to be the most efficient model. Realistically, further RMSE reductions become more and more valuable whereas this is a simple linear analysis, and if the model only needs to be computed one time then the cost can be more easily amortized with its usage.

4.2 Stability

Because our models have anywhere from 81 up to 4126 parameters, it is impractical to report the mean and standard deviation for every parameter of every model in a table for the sake of the report formatting. Instead, we analyzed the numerical stability of each model by using the mean and standard deviation of the parameters as a percent with respect to the mean. We have included histograms of the parameter standard deviations with respect to their mean for parts 3a,3b, and 3d.

For the standard linear model from part 3a, of the 81 parameters, 59 of them have a standard deviation less than 10 percent with respect to the mean. A majority of the parameters having such consistency across 10 folds of cross validation definitely suggests the linear model being numerically stable.

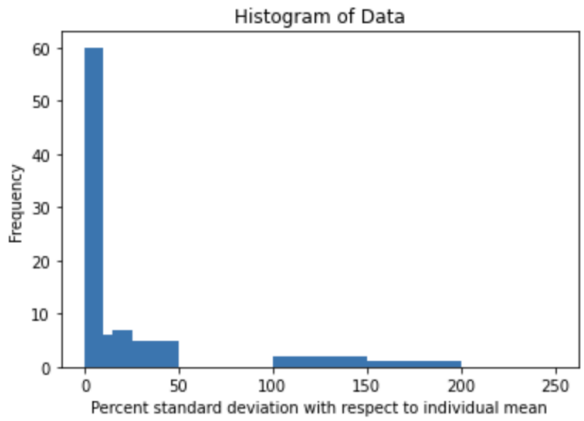

Figure 4: Standard deviations as a percent with respect to each individual mean for 3a parameters

For the 3b model, where we added 45 extra features computed from mathematical transformations of previous features, we don't see the same level of numerical stability. This time out of the 126 features, only 60 had standard deviations of less than 10 percent with respect to their means. This still indicates at least some numerical stability, it is not as strong of an argument as we had for the model of 3a. However, it is expected that with an increase in features, there will be more difficulty keeping the algorithm stable due to overfitting or ill conditioning. Since 126 features is nowhere in the range of overfitting our data, this suggests an increasingly ill conditioned design matrix. An ill conditioned matrix can be often due to redundant features, highly correlated features, and plenty more, but in our case these two seem the most likely. One way to address an ill conditioned matrix is to add regularization, as we did in part 3c.

For the best model of part 3c, we only have one fold due to the expensive computation so there is no histogram. When we revised the regularized model by adding random features to increase the fitting power, the norm of the parameters comes out to about 0.51. It is very useful to compare this to the norm of our regularized model without random features, because even though we have significantly increased the number of features, from 126 to about 4100, we have lowered the norm of the parameters. This suggests that the extra parameters don't add significant complexity to the model and also indicates that the regularization parameter is effective in constraining the model. In the context of this model, given it's relatively strong predicting performance and comparison to the other regularized model, there is a good argument that this model is numerically stable.

Our best model from 3d (inverse tangent) has good indication of being numerically stable, with 49 of the 80 parameters with standard deviations less than 10 percent of their respective means. A

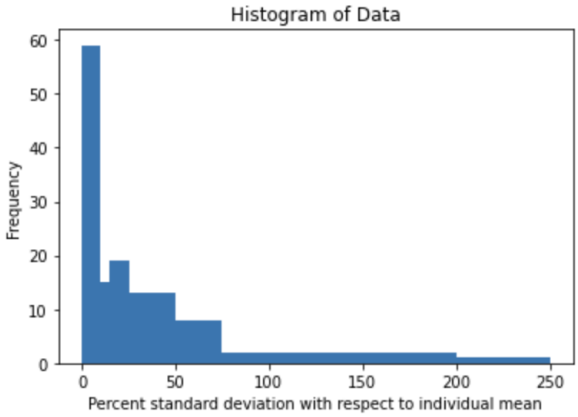

Figure 5: Standard deviations as a percent with respect to each individual mean for 3b parameters

few parameters were also just outside of this, with 55 of the 80 under 15 percent. This follows closely behind the linear model from part 3a, which is expected since the nonlinear model is a more complex representation of the data. They also performed roughly the same when it came to predicting power, the nonlinear being slightly worse, part of which could possibly due to slightly worse stability. Although we are just reporting on the best model as per the report instructions, we also looked at a few other functions that would be prone to either cancellation or rounding and indeed saw poor levels of stability, giving us a nice sanity check.

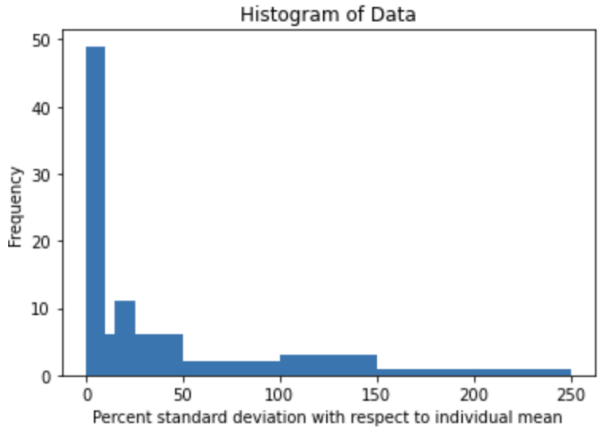

Figure 6: Standard deviations as a percent with respect to each individual mean for 3d parameters

Although it is difficult to make concrete conclusions about numerical stability just by looking at the norm and standard deviation of parameters across folds, we can draw educated conclusions given the context of our models. As one might have expected, the most stable model was the original and simplest linear model from part 3a, although the inverse tangent non-linear modification also had impressive stability. The addition of random features in part 3b reduces the numerical stability of the model, but the regularization performed in the best model from 3c seems like a good way to keep this in check.

5 Conclusion

Throughout this project we walked through some of the most common and important steps of creating a prediction model for a given data set. We found data in which we would like to fit a predicting model, analyzed the features of our data, decided on the type of prediction that would be performed, and created various prediction models and analyzed their performance as well as analyzing the efficiency

and stability of these models.

Our goal was to predict the critical temperature of superconductors given a variety of chemical and molecular properties such as electron affinity and molecular mass. The lowest RMSE we were able to achieve with our models was 0.43, coming from our regularized linear model with added random features. Originally we had too few features in comparison with the amount of samples, only 81 features vs 21263 data points, and the regularization did not have much to work with. To overcome this issue, we added random features by using random data generated based on existing features in order to increase the fitting power of the model. Having a significantly increased number of features also allows the regularization to have a much higher impact, so the improvement we saw when using regularization before and after adding random features made sense. However, .43 is still a non-desirable RMSE, meaning our best model still wouldn't be very useful in predicting the critical temperature of a newly discovered or theoretically modeled superconductor material. All of our best models from each step were decently stable algorithms, although we did have a few non-linear models that performed poorly and were not numerically stable. Overall, we were able to create a variety of different models that were usable but lacked a desirable level of precision.

One of the biggest issues we encountered was a lack of domain knowledge. The relationship between various molecular properties and the critical temperature of a superconductor is a complicated issue, and without very strong background knowledge in physics and chemistry it was difficult to perform many of the most common model improvement methods related to feature analysis/engineering. One such method is creating interaction features to give your model more data and features to work with, especially if you're able to create interactions that produce linear relationships for the linear model to use. Another such method is knowing a good function to use for the non-linear model. Without domain knowledge on your data, it becomes more of a guess and check situation when creating functions to base non-linear models on. With proper domain knowledge, both of these methods, and many more, could've been utilized to increase the accuracy of our models.

Overall, this project provided us with a solid foundational experience and familiarity with the process of generating a variety of prediction models, and many of our largest issues could be solved with the selection of a different data set. However, it was also a good learning experience dealing with a difficult and unfamiliar data set, educating us on many of the difficulties encountered in data analysis such as feature engineering and coming up with a good non-linear model. We were also forced to become more creative, utilizing a wider variety of problem solving methods such as the addition of random features to improve performance on an unfamiliar data set.

References

[1] Hamidieh, Kam, A data-driven statistical model for predicting the critical temperature of a superconductor, Computational Materials Science, Volume 154, November 2018, Pages 346-354

Data collected from UCI Machine Learning Repository:
https://archive.ics.uci.edu/ml/datasets/superconductivty+data