ML Problem set 1

Krishna N. Dindial

February 25, 2026

1 Github

`https://github.com/kdindial/ML_Spring26.git`

2 Introduction

For this problem set I used linear regression, k nearest neighbors and multi-layer-perceptron to try to predict a star's gravity using the spectra of the star. I normalized features with the training data so that each feature in the training set had a standard deviation of 1 and a mean of 0. In other words, I shifted each parameter by subtracting the mean training data, and I scaled each parameter by the 1/standard deviation of the training data. I applied this transformation to the the validation set and the test set. I also normalized the training, test and validation labels this way. Looking back it was probably not necessary to normalize the labels. I "scored" the model by calculating the difference for each star in the model and the validation data and squared that difference. I then summed up the errors squared for each point in the validation set and used that sum. My code, and this writeup are in my github repository.

3 Linnear Regression

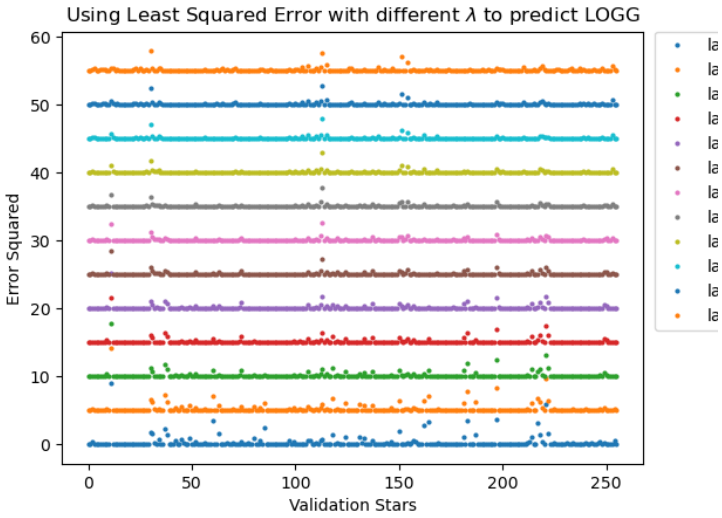
For the linnear regression I used np.linalg.solve() to solve the equation

$$Y^* = X^* X^T (X X^T)^{-1} Y \tag{1}$$

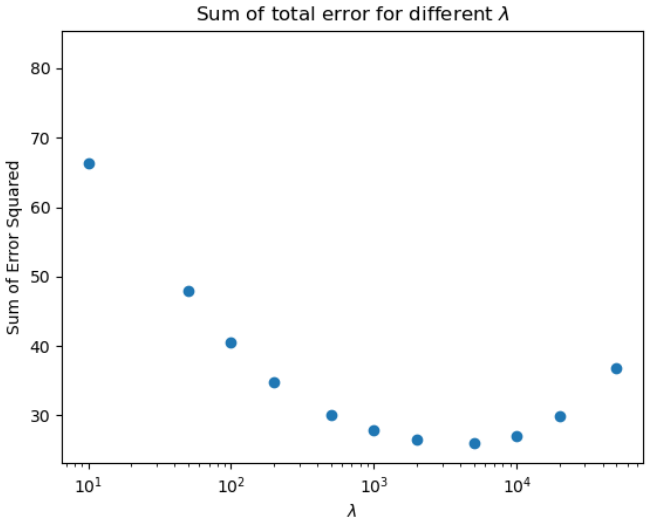
Where the design matrix made from X is the spectra of the training data, y is the surface of the of the training data, X* is the validation spectra design matrix, and Y* is the validation gravity. This equation works when the number of parameters is larger than the size of the training data. I wanted to play around with different parameters, so I added in a regularization factor λ, which changes the equation to

$$Y^* = X^* X^T (X X^T + \lambda I)^{-1} Y \tag{2}$$

To see how this affected my error, I plotted the error squared for each star as a function of different lambdas.



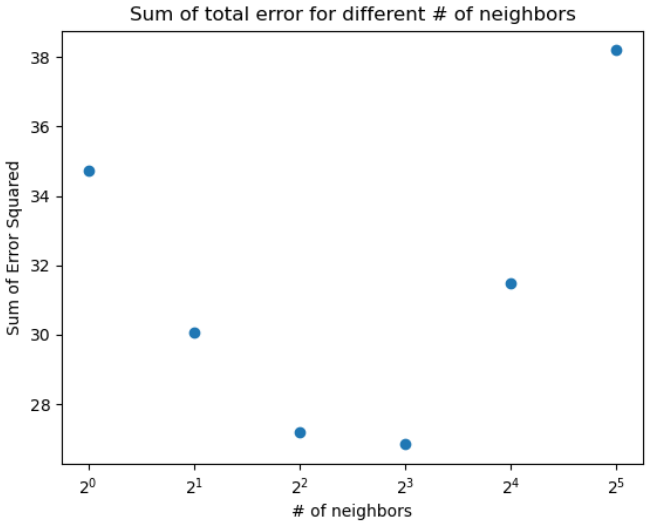
Its a bit of a strange plot, (I offset each lambda by 5 so that they would not all be stacked on top of eachother). But its interesting to see how for some stars with large error, the error gets smaller as lambda increases and for others the error gets larger as lambda increases.



To pick the best lambda, I plotted the total error squared as a function of lambda. The optimal lambda seems to be around 5,000, with a total error of around 25. I will use this value of lambda on the test set in a later section where I compare the models to the test data

4 K Nearest Neighbors

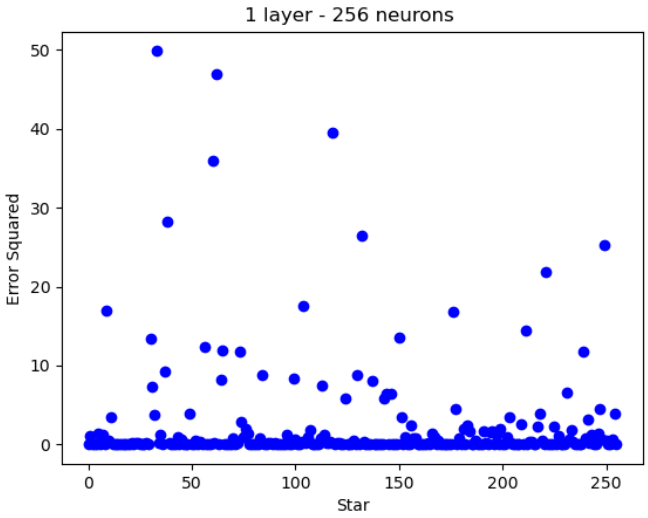
We have around 8k parameters, so I was a bit intimidated to calculate an 8k dimensional distance, for each star in the training data and each star in the validation data. I asked chatgpt how I should go about this, and it told me there is a sci-kit learn method that does this for you. I just imported the function and tried different fits for different numbers of neighbors.



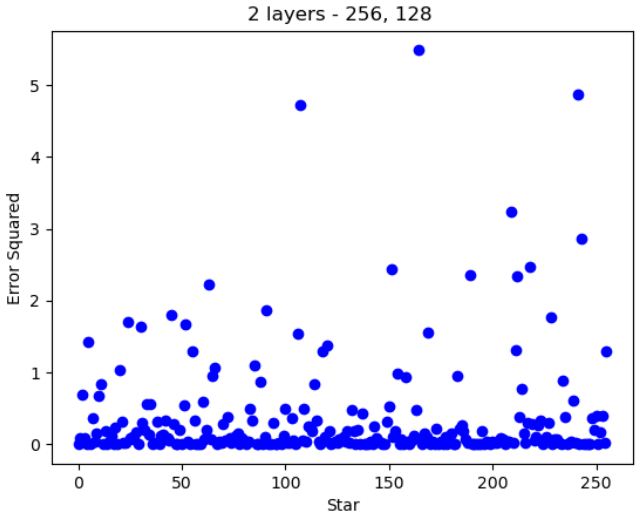
KNN with 10 neighbors performed the best with a total squared error of around 27.

5 Multi-Layer Perceptron

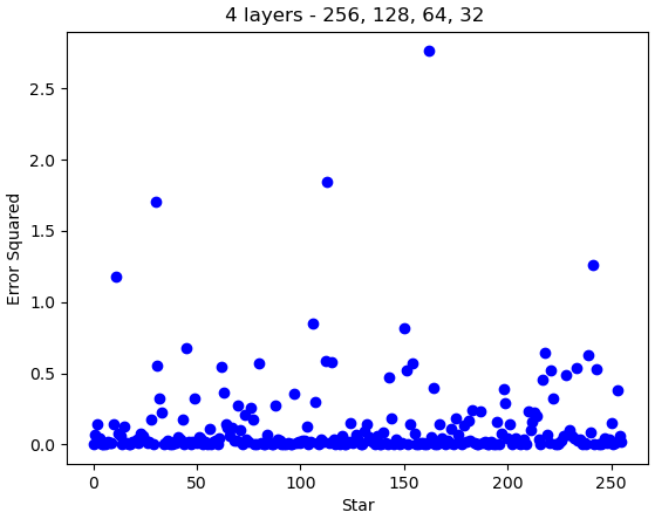
For the multi-layer perceptron model I used the MLPRegression function from sci-kit-learn. I used RELU as nonlinearity function. I used ADAM as my optimizer with an alpha parameter of 3e-4 per the suggestion of https://karpathy.github.io/2019/04/25/recipe/. For my first attempt I started with 256 neurons.



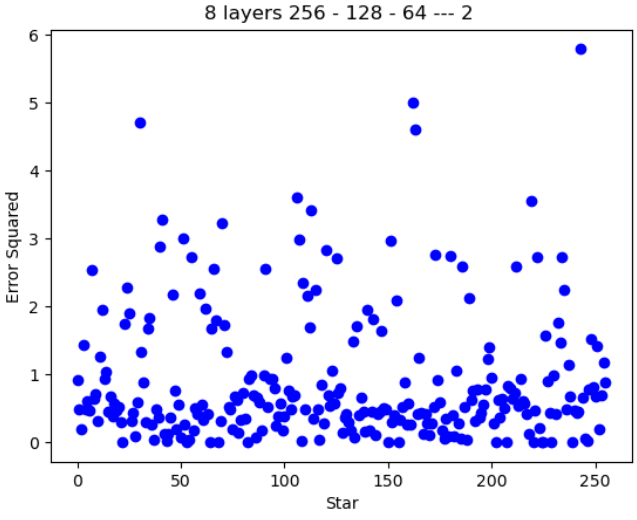
The sum of the squared error was: 466. This is much worse than linnear regression and KNN. I thought it might get better if I passed it through another layer with less neurons. For the second attempt, I passed the same parameters, except now now with 2 layers. The first had 256 neurons and the second of had 128.



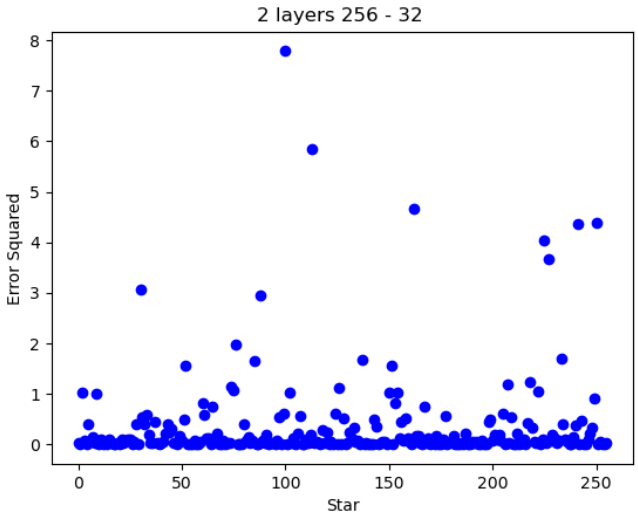
The sum of the errors was around 102. I then tried 4 layers, of size 256,128, 64 and 32. My motivation for this strategy is that I visualized the diagram Hogg often draws of converging from a large number of parameters, down to a smaller number of parameters.



The sum of the error was around 32. This is closer to KNN and linear regression. I then decided, why not go all the way with this pattern and I did 8 layers: 256, 128, 64, 32, 16, 8, 4, and 2.



The total loss squared was around 233, so it did not out perform the 4 layer version of this pattern. The next thing I wanted to try was skipping the intermediate layers. I tried it with just two layers, going from 256 to 32.



The sum of the errors was 93. I tried even more combinations of layers, but I am not including them all here, but they are in the notebook 'problem1.ipynb' and there are more plots in the figures section of the repository. The best scoring MLP I tried had 4 layers, going 256,128,64,32.

5.1 Randomization

We are asked to run the same regression with different random seeds.Since the model with 4 layers decreasing in size from 256 to 32 had the best performance, I used the same set up with 5 different random seeds. The sum of squared error for the different random seeds were: 32.187217805638014, 31.046279633403266, 41.990782296617915, 34.29658620479617 and 30.051910919472213. This seems pretty stable to me.

6 Performance on Test Data

I measured the performance by calculating the difference between the prediction of the LOGG of each planet in the test set and the actual LOGG of the test set, then squared this difference and then calculated the sum. I also realized its probably more physically meaningful to un-do the normalization transformation and then take the average absolute error. That way I get a value for the error that is in the units of LOGG.

Using linear regression with a regularization parameter of 5,000, the sum of the errors was 44.621932733752374. The average absolute error was 0.1465448620344

Using KNN with 10 neighbors, the sum of the errors was 54.015724. The average absolute error was 0.15366906.

Using MLP with layer sizes of 256, 128, 64 and 32, the sum of the errors was 59.974057873347576. The average absolute error was 0.15838947465216.

I am not an astrophysicist so I have no idea if an average LOGG error of 0.15 is good or bad.

7 Final Project

A former post doc in my lab wrote this paper where they used machine learning to predict the mean free path of an InAs quantum well from the crosshatch pattern measured from the AFM scan. https://arxiv.org/pdf/2409.17321 I think it would be interesting to see if I could improve the model. Before sending the afm data into the ML model, my colleagues, manually extracted the crosshatch pattern of each scan. This was a slow, manual process that required rotating each afm image and as a result, they really did not get enough data. I think I could make a model that skips this manual step, and also improve the model. One big problem I could see is that we might not have enough afm data. Each afm scan takes around 30 minutes and the data can be noisy, but if I can at least skip this manual de-noising process, I should have access to around over 500 afm scans.

If I do not end up doing this for my final project, I would like to do something with sports gambling.