Final Project

Matthew Roberts
CMSE 381
May 3rd, 2023

Introduction

Statistical learning algorithms are used all around us today in a wide variety of applications. They help select the sky in lightroom, they help classify images, they even drive helicopters and cars. In this seemingly endless list of uses, there is one particular use case that has had strong development in recent years, and that is cell biology.

Cell biology has many applications that affect the population without them even knowing it. Crops have their genes modified to better withstand pests and environmental conditions. New drugs are developed and tested utilizing research in this field. Learning how the body works and responds to food, environment, conditions and so much more relies heavily on this field. In order to perform many of the applications above, we need a toolkit to measure the number of proteins inside the cell. Proteins are responsible by and large for running the cell as they are what perform cell actions. Unfortunately, directly measuring the number of proteins is expensive, time consuming, and extremely complex. We need a better way to do this.

Cue statistical learning.

There are a number of factors that are responsible for the production of proteins and as such are correlated with them. None more so than RNA. RNA floats from the nucleus to the ribosomes scattered throughout the cell. RNA is directly responsible for the production of proteins within a cell. Perhaps by measuring the number of RNA copies, we can determine the number of proteins that are present within a cell. Counting the copies of RNA is cheaper, less challenging, and significantly less time consuming than counting proteins. That makes this method appealing.

The goal of this paper is to outline the process that I took to predict protein counts based on the number of RNA copies in each cell.

Methodology and Results

The only thing initially given to us was the data. It came in the form of .h5ad files which is something that I had never worked with before. It was necessary to understand how the data was structured within this file before building a model.

To read the data, I needed to import a library called anndata. I'm able to do so by working in the dance-env within the HPCC.

```
import anndata as ad
```

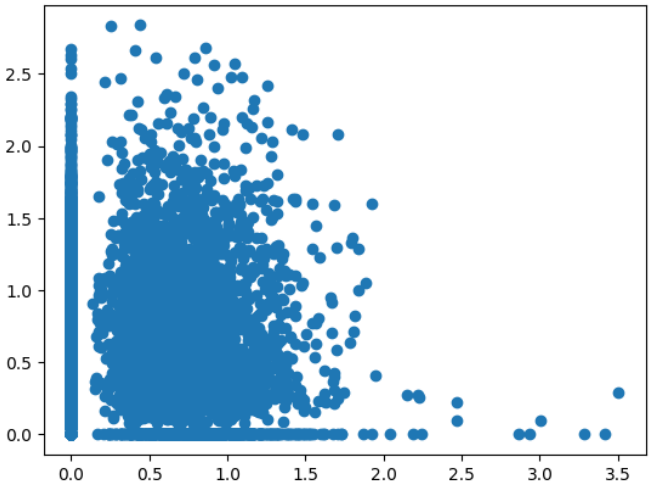
Data was then imported in. After modifying the data, I came to realize a couple of important realities. The first was that this data was set up with many different tables. I could call upon cells v cell attributes, I could call upon a table of all RNA values, or I could call upon a table of all the protein attributes and copies. What I really wanted to pull from the data was copies of RNA and number of proteins. The code below shows us pulling the number of RNA copies into an array.

```
# Not normalized
A = adata_gex.layers['counts']
A.toarray()
```

However, I could also pull normalized copies. This is really what I wanted to work with as normalized data reduces the model error.

```
# Normalized
A1 = adata_gex.X
A1.toarray()
```

To visualize the data, I plotted the number of a specific protein v the number of copies of the RNA sequence that is directly responsible for producing that protein.



But wait… why are there so many zeros along both axes? If the RNA sequence for producing the protein in the cell is present shouldn't that protein exist? And if that protein exists shouldn't there be RNA sequences present that code for it? This simple plot shows the difficulty in predicting the number of a specific protein based on the number of copies of the RNA sequence that code for it. Gene expression, cell activities, and other such unknown processes make this relationship anything but one to one. As a result, whatever model that I use will not only need to make use of traditional learning methods (linear

models, MLP's, etc) but also need to utilize some sort of either feature reduction or cleaning method to make this data more "approachable".

Baseline PCA Mean

The first model was handed to us, and that was a pca average. The code ran a PCA analysis on the data and then constructed an average value to use as the predicted value. The RMSE for this "model" was 0.58. It shouldn't be too difficult to get lower than this.

Baseline PCA Linear

The second model was also given to us to use as a starting point. This model again reduced the number of features on the data by utilizing Principal Component Analysis. However, instead of calculating a single prediction value for all points, it constructed a linear regression model that could do a significantly better job at predicting the values. After running with all of the data, an RMSE value was obtained of 0.3827.

Lasso with PCA

One of the issues that linear regression faces is that it tends to overfit the data. This means that it may perform exceptionally well on the training data, but fail to deliver even remotely good results on the testing data due to high variance in the model. To alleviate this issue, one of the functions we learned about in class was Lasso. This method takes the coefficients for the linear model and one by one reduces them to zero. This creates a far less complex model by adding a penalty term which helps to reduce overfitting. This seemed like an approach that would help with the data given. The model was changed from a linear model to a lasso model. After running on all of the data, a final RMSE was determined of 0.48. Not great and I'm not really sure why. I would think that with no penalty term, the original linear model would revert to overfitting, but that doesn't appear to be the case since Lasso did even worse. My guess is that the PCA reduction removed enough information to prevent overfitting. By applying lasso I "removed" even more information, leading to a worse outcome.

Ridge with PCA

Despite the issues with lasso, I decided to still apply a similar technique for redundancy. Similar to Lasso, ridge also helps to prevent overfitting. However, rather than reducing coefficients to be zero, ridge instead reduces coefficients *towards* zero. This doesn't necessarily make the model less complicated by eliminating coefficients, but it reduces the impact of the coefficients in the mode. This in turn reduces the variance of the model. The model was trained and then tested on all of the data and an RMSE was obtained of 0.3815. This was just barely better than the original model which in all likelihood had to do with the selection of data rather than an improvement in the model.

Basic MLP

Rather than focusing my efforts on a model that may not best represent the data, I instead turned my attention to the Multi-Layer Perceptron. Perhaps by utilizing a more complex model I could lower the

RMSE in a significant way. A 2 layer MLP was initialized, similar to an in class assignment that we did early this semester. This was run on all of the data, but unfortunately didn't deliver the results I was looking for, with an RMSE of over 0.4.

I then added batch normalization to the model in the hopes that it might reduce the testing error. It did a little bit, but it was not enough. The full model can be seen below.

```
# Hyperparameters for our network
input_size = 13953
hidden_sizes = [5000,500]
output_size = 134

# Build a feed-forward network
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.BatchNorm1d(hidden_sizes[0], affine=False),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.BatchNorm1d(hidden_sizes[1], affine=False),
                      nn.Sigmoid(),
                      nn.Linear(hidden_sizes[1], output_size))
print(model)
```

MLP with BN and PCA

Since the PCA above delivered strikingly good results for only a basic linear model, I decided to apply that to my MLP. I did so and saw a small reduction in RMSE. At this point it was time to start playing around with different parts of the model to tune it to get as low as possible. The first thing I did was modify the hidden layer sizes. It started out as 400 and 300, and was increased drastically to 5000 and 500 with an output of 134. This I found delivered the best results for us at the time of this writeup. I also experimented with the number of layers that the model utilized. The original model had just two layers. I started adding layers and found that past three layers I did not find much success unfortunately. However, three layers was the best RMSE I had found so far with just over 0.37. Another part of the code was modified, being the activation function for the model. I swapped out the final Sigmoid function with Tanh and found that this pushed the RMSE into the 0.36 range. I finally combined everything I had learned from tinkering around into the following model and ran it.

```
    # Train the model on the PCA reduced gex 1 and 2 data
    # Hyperparameters for our network
    input_size = n
    hidden_sizes = [5000,5000,5000]
    output_size = 134
```

```
    # Build a feed-forward network
    model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                          nn.BatchNorm1d(hidden_sizes[0]),
                          nn.ReLU(),
                          nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                          nn.BatchNorm1d(hidden_sizes[1]),
                          nn.ReLU(),
                          nn.Linear(hidden_sizes[1], hidden_sizes[2]),
                          nn.BatchNorm1d(hidden_sizes[2]),
                          nn.Tanh(),
                          nn.Linear(hidden_sizes[2], output_size))
```

This yielded the best yet RMSE of 0.3680

Magic + MLP + PCA

A final last ditch effort was made to see if I could push my model to be any better. I had heard success stories from classmates about the usage of a library called magic-impute. I looked into it and discovered that it would fill in holes where data should have been. My count of RNA copies was probably wrong in many scenarios, yielding 0 copies often when there probably should have been at least a few copies. Magic-input helped to remedy this reality by filling in some of these zeros with what the value should have been. To implement this in my code, I would need to do this before the PCA reduction, as the reduction depended on the proper counts of RNA copies. After adding to the code, I then ran it and got an RMSE of 0.4. This wasn't good. At the time of this writing I was unable to tweak it and/or determine what went wrong with my limited time. Implementation is below.

```
# Running Magic Operation
magic_operator = magic.MAGIC()
result = magic_operator.fit_transform(input_gex.X)
```

Conclusion

First and foremost, the goal of building a model better than the baseline mean and baseline linear model with pca was accomplished. Those two values were 0.58 and 0.385 respectively. The final model that I built returned an RMSE of 0.3680. Fortunately, this was not only below the baseline models, but also below 0.37 which was a personal goal with this project.

The second biggest takeaway for me was just how "mess around and find out" building a statistical model was. In all the classes I have taken, and all the assignments I have finished, there was always a "right way" to do things. Whether it was solving a math problem, or building a sort of searching algorithm, there were always generally accepted "right" ways to do things. In building models for this project, I came to

realize that there is no such thing. As was mentioned at some point in class, building a model cannot rely on memorizing a specific way to do things. It comes from a deep understanding of how each model behaves, and what modifying each part of the model does… in other words, your intuition on what should be done. This is something that I struggle with, and so made this project difficult. I had no idea what adding a layer to my MLP would do. Or removing neurons from a few of the layers. Or how I structured the MLP with ReLU or other activation functions. I had absolutely no idea what changing each one really truly meant for the model. As a result, my time was spent more in guessing and tweaking things, than actually making informed decisions.

There were a few other takeaways.

It was really quite impressive just how difficult it was to beat the basic PCA linear model. Perhaps this has more to do with my lack of understanding, but it was frustrating at times how difficult it was to beat the baseline models. I would build a multi layer perceptron with multiple layers, thousands of neurons, with magic imputation to go along with it, and it was still outmatched by the basic linear model. It really went to show that you can't just throw machine learning at every corner and expect good results right off the bat.

One of the big frustrations I had with this project (and perhaps building models in general) was my inability to submit jobs and get results immediately. The code took sometimes upwards of an hour to run, and that was after the hours that it took for the slurm queue system to actually start my program. That meant that I had to know damn well that my code was going to run correctly without any sort of bugs when I submitted it to the queue. If not, I wasted 5 hours. This also meant that I couldn't start the project three days before the due date and expect to get good results. I didn't realize this until a week before the project was due, so I got lucky, but I could have certainly made this model a lot better if I had started early.

This project did a good job of putting into perspective what skills are necessary to succeed in this arena. Creativity, time, certainly good coding skills, and a deep understanding of how these things work. While I may not have been able to reduce my RMSE to extremely low values that others in the class were capable of, I learned more about statistical learning in this project than I have in any other class and assignment that I have taken. And that really is the most important part of this.