Probabilistic Programming P3

Eduard-Vlad Manea 407

Glicko-2 Enemy Rating Deviation prediction

Abstract

Glicko-2 is an algorithm used for creating a rating-based system flexible enough to be able to range from video games up until real sports.

HEMA, Historical European Martial Arts, is a martial arts full contact sport like Olympic fencing, with differences in protection gear, weapons used and rules.

Glicko-2 in HEMA

In HEMA, events between clubs occur, to be able to compare the fencers between each other, a rating system is needed. The most used site for seeing how good a fencer is or to see the history of matches for each competition, https://hemaratings.com/ is used.

How Glicko-2 works

For Glicko-2 algorithm to work, it needs a couple of variables as input:

- The current rating for the fencer we want to calculate the new rating
- The current rating deviation for the fencer we want to calculate the new rating deviation

- An array of ratings of the opponents the fencer has fought in the competition
- An array of rating deviations of the opponents the fencer has fought in the competition
- The result of the matches
- Volatility, a variable that changes the rating deviation of a fighter over a period of time, very useful if a fencer has not fought in many years

Before we continue, let us define what each input means:

- **Rating**: A value that describes how good a fencer is (e.g., 1365)
- **Rating Deviation (RD)**: This value tells us an approximation of how much the rating of a fencer will change based on the outcome of a match (higher RD means that the rating will change much more than a lower RD). This variable also is a way for the algorithm to interpret how sure is it of the rating it gives a fencer. A higher RD means the algorithm is not too sure of the fencer's current rating. A lower RD means the algorithm is very sure that the fencer's current rating is the right one.
- **Score**: The algorithm takes one or multiple matches and outputs the new rating of a fencer. An important parameter is the score of each fight the fencer was in. The values can be **0/0.5/1**, each representing **lose/draw/win**. If the fencer wins a fight, we expect the rating to grow, if the fencer loses the fight, we expect the rating to fall and if it is a draw, depending how good the opponent was, the fencer can get or lose points.
- **Volatility**: as explained above, it changes the RD of a fencer over a period of time. This is good for fencers that do not participate to competitions for a long while. As time goes by, the volatility rises the RD of a fencer, telling the program that the current rating

becomes increasingly unreliable in defining the fencer's true rating.

Implementation of the Glicko-2 algorithm

Following the steps to recreate the algorithm, http://www.glicko.net/glicko/glicko2.pdf, I have made a small modification in hoping of being able to create the inverse function of said algorithm. The modification consists in assuming that the volatility is a constant, rather than a variable that changes at step 5 in the document. When running the algorithm, the new volatility was almost identical to the current volatility (0.05999 vs 0.06). After furthermore investigation, the difference of 0.000001 between each other impacted rating and RD starting at the 2nd decimal of each value. Because of this, in order to avoid a heavy computational process that would not have any possibility to create an inverse of it, I have removed it completely, assuming the volatility is a constant of 0.06.

Simulating matches for verification

To verify that what was implemented was right, I simulated several matches ten thousand times trying to see if the final values converge towards the same value or interval of values.

In one simulation, I have created 20 consecutive matches of one fencer against 20 different fencers.

To verify that the final values converge to an interval of close values, for the enemy ratings and enemy RDs I have used normal distribution and uniform distribution, we will compare below the differences.

Normal distribution:

- enemy **rating**: normal(1300, 200)

- enemy **RD**: normal(150, 50)

Uniform distribution:

- enemy **rating**: uniform(700, 2000)
- enemy **RD**: uniform(50, 250)

NOTE: Score has 80% to be 1 and 20% to be 0, it means that most of the time, the fencer will win no matter the enemy's rating. This was done to show that even if there are unpredictable matches, the algorithm still converges somewhere. When I created the train data, the chances of winning are impacted by both players ratings.

One step of the algorithm is computing v, following the formula:

$$\mathrm{V} = \left[\sum_{j=1}^{m} g(\phi_j)^2\, E(\mu, \mu_j, \phi_j)\left(1 - E(\mu, \mu_j, \phi_j)\right)\right]^{-1}$$

Where

$$g(\phi) = \frac{1}{\sqrt{1 + \frac{3\phi^2}{\pi^2}}},$$

$$E(\mu, \mu_j, \phi_j) = \frac{1}{1 + e^{\left(-g(\phi_j)(\mu - \mu_j)\right)}}$$

From this, we can observe that $E(\mu, \mu_j, \phi_j)$ is actually sigmoid(x), where x = $g(\phi_j)(\mu - \mu_j)$

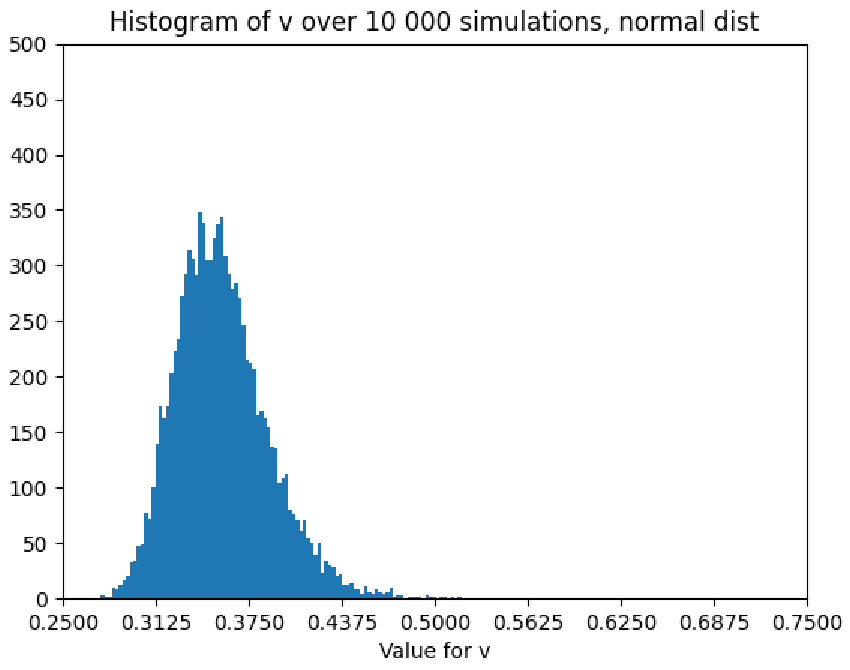
ϕ - is **Rating Deviation**

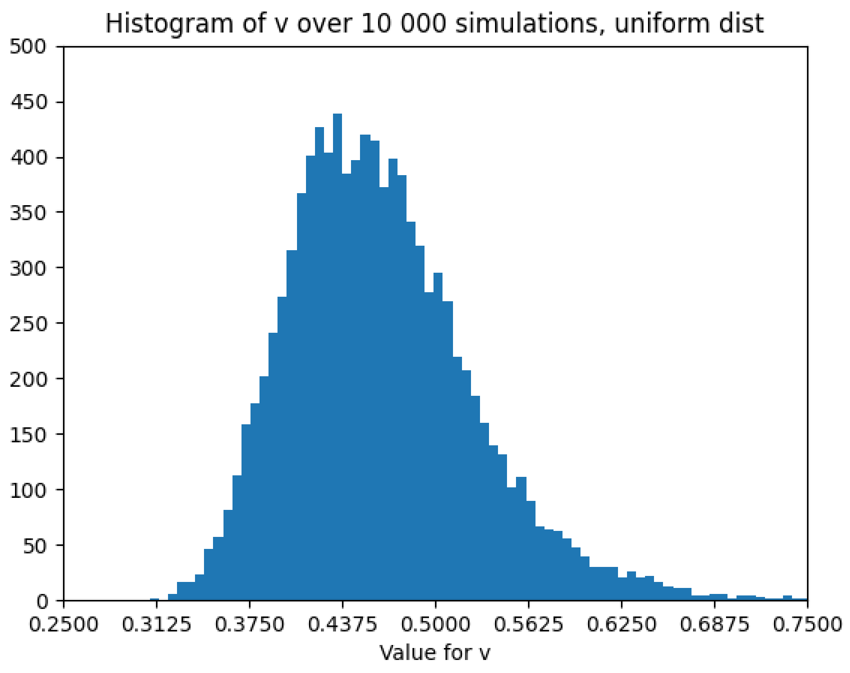
ϕ_j - is **Rating Deviation** of the j enemy in the array

μ - is current **Rating**

μ_j - is current **Rating** of the j enemy in the array

V is the estimated variance of the fencer based only on the games outcome.





From these 2 figures, we can observe that the variance in the uniform distribution case is higher. This is because there are more enemies with high ratings than in the normal distribution case, enemies that have only 20% to win against the fencer.

Another key step worth mentioning is updating the current **rating** and **RD**

$$\phi' = \frac{1}{\sqrt{\frac{1}{\phi^{\#2}} + \frac{1}{v}}}$$

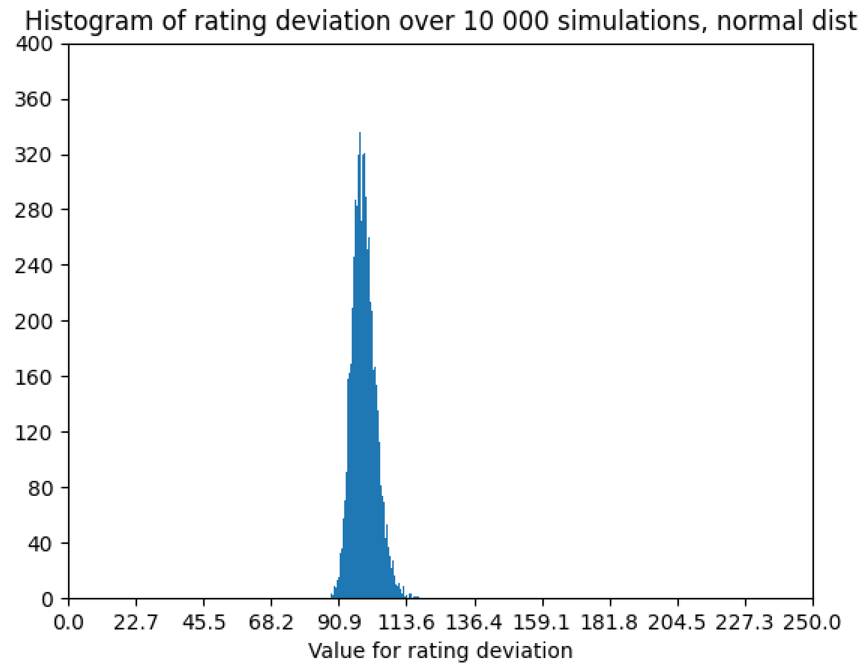
$$\mu' = \mu + \phi'^2 \sum_{j=1}^{m} g(\phi_j)\left(s_j - E(\mu, \mu_j, \phi_j)\right)$$

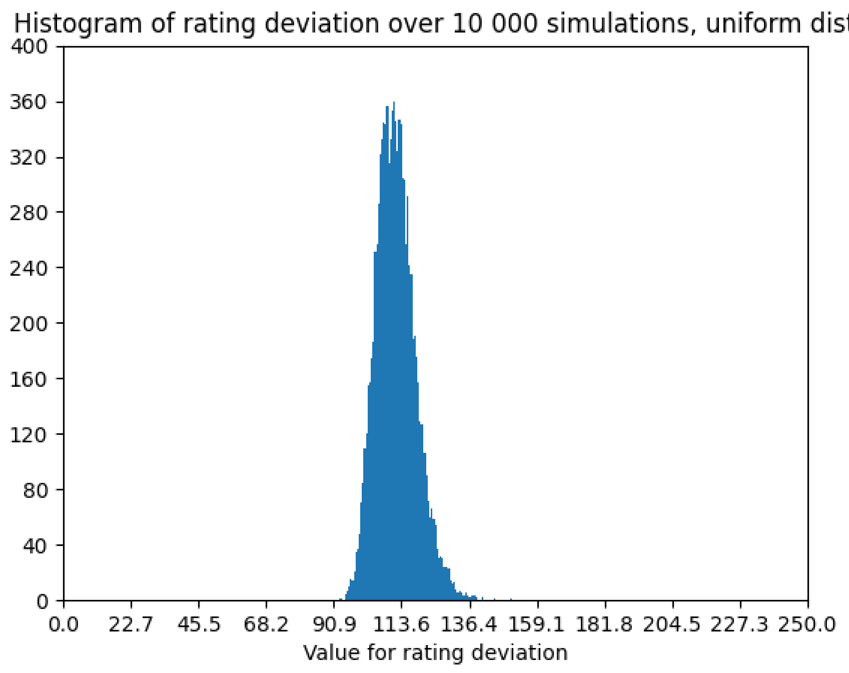
Where

$$\phi^{\#} = \sqrt{\phi^2 + \sigma^2},$$

$\sigma =$ Volatility

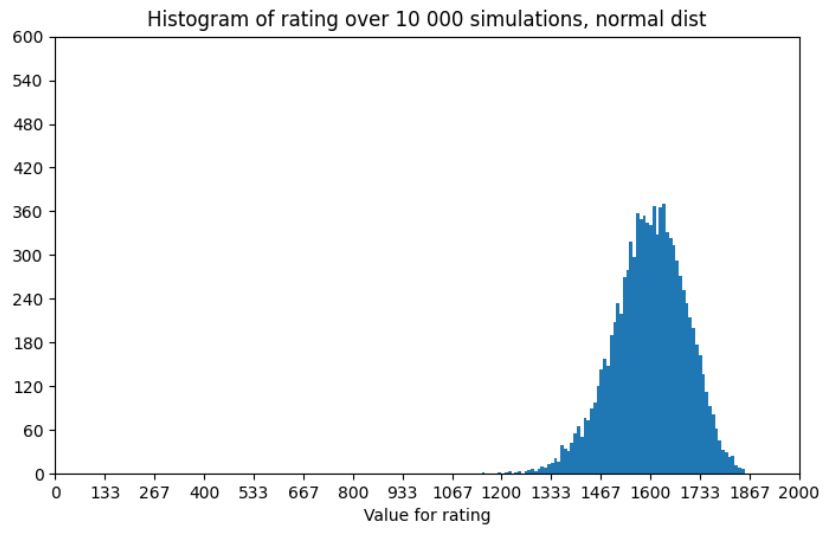
The figures for the rating deviation after 10 000 are:

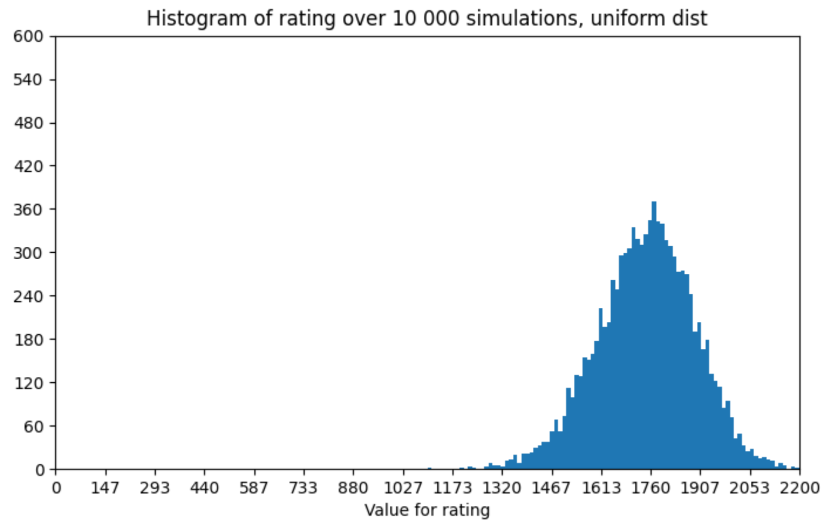




From this we can conclude that the algorithm does not care if you win or lose against better or weaker opponents, the **RD** changes nearly the same in both cases because there were 20 matches played each time. The **RD** is modified based on the number of games.

The rating however, should change based on who the fencer plays against.





And it does, because on the uniform case, there are more high rated players the fencer fights and wins, the rating gets modified accordingly.

Failed attempt at finding RDs

The first attempt was to try and find the inverse of the glicko-2 algorithm but unfortunately, I have encountered multiple problems.

The first problem was the usage of the “Illinois algorithm”, a variant of the regula falsi procedure. This was used to obtain the new volatility. The solution was, after close observations, to assume the volatility as a constant that does not need to be changed in order to skip this step entirely, both forwards and backwards.

The second big problem which I have encountered to try finding enemy RDs was the fact that the algorithm allowed the usage of an array of enemies when calculating one's rating and RD. This impose the idea that, when going backwards, I would have had on equation with multiple unknown variables which I needed to find. A solution for this problem was, instead of treating the fights as they were being done in parallel, we would see them sequentially, thus, instead of accepting an array of enemies, we would accept one enemy, calculate the new rating and RD, and use them as the new input for the next fight.

Old way:

(Current_rating, Current_RD) X[fight_1, fight_2, …, fight_n] -> (final_rating, final_RD)

New way:

(Current_rating, current_RD) X fight_1 -> (new_current_rating_1, new current_RD_1) X fight_2 -> …. X fight_n -> (final_rating, final_RD)

The differences were not big, even after 50 matches:

Old way:

- final_rating was 1545.1714218203456
- final_RD was 62.60599871323218

New way:

- final_rating was 1550.8148500944383
- final_RD was 73.83821580627654

So, the solution was a success, instead of using multiple enemies at once, we would take them one by one, using a fight's output as another fight's input both ways.

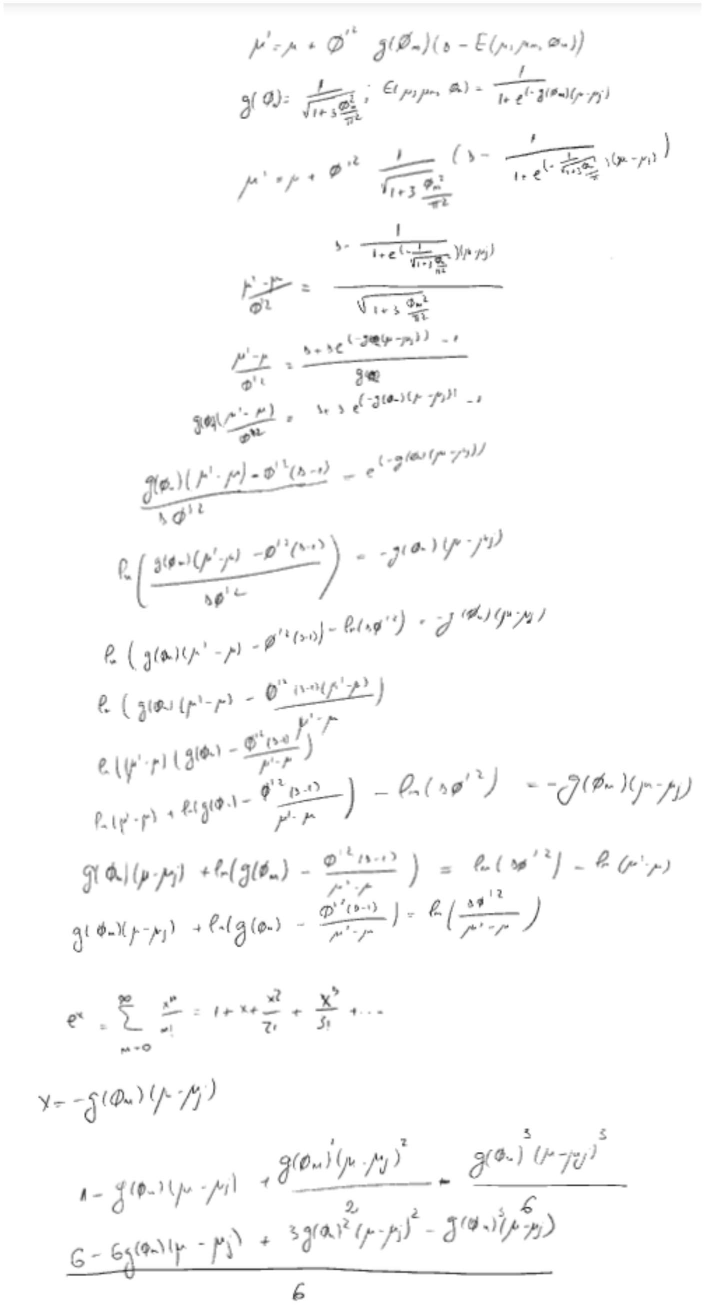
The third and final problem, which I could not solve, was a mathematical one. The RD is used both in a polynomial and an exponential, especially in a multiplication that uses both at the same time. This would mean that I would need to find x, where y = P(x)*E(x), where P is a polynomial and E is an exponential.

After trying different methods, the "best" one was to transform the exponential in a Maclaurin series that looks like this:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Not knowing exactly what is the best approximation for the required interval, a small polynomial approximation of the equation would not have sufficed, thus we could have reached a polynomial of 14th degree (or more) without a problem, having thus maximum of 14th possible (or more) answers.

Alas, this way of solving the problem had reached its end.



$$\mu' = \mu + \phi'^2 \, g(\phi_m)(s - E(\mu, \mu_m, \phi_m))$$

$$g(\phi) = \frac{1}{\sqrt{1 + 3\frac{\phi_m^2}{\pi^2}}}; \quad E(\mu, \mu_m, \phi) = \frac{1}{1 + e^{(-g(\phi_m)(\mu - \mu_j))}}$$

$$\mu' = \mu + \phi'^2 \frac{1}{\sqrt{1 + 3\frac{\phi_m^2}{\pi^2}}} \left(s - \frac{1}{1 + e^{(-\frac{1}{\sqrt{1+3\frac{\phi_m^2}{\pi^2}}})(\mu - \mu_j)}}\right)$$

$$\frac{\mu' - \mu}{\phi'^2} = \frac{s - \frac{1}{1 + e^{(-\frac{1}{\sqrt{1+3\frac{\phi_m^2}{\pi^2}}})(\mu - \mu_j)}}}{\sqrt{1 + 3\frac{\phi_m^2}{\pi^2}}}$$

$$\frac{\mu' - \mu}{\phi'^2} = \frac{s + s e^{(-g(\phi)(\mu - \mu_j))} - 1}{g(\phi)}$$

$$g(\phi)\frac{(\mu' - \mu)}{\phi'^2} = s + s e^{(-g(\phi_m)(\mu - \mu_j))} - 1$$

$$\frac{g(\phi_m)(\mu' - \mu) - \phi'^2(s-1)}{s\phi'^2} = e^{(-g(\phi_m)(\mu - \mu_j))}$$

$$\ln\left(\frac{g(\phi_m)(\mu' - \mu) - \phi'^2(s-1)}{s\phi'^2}\right) = -g(\phi_m)(\mu - \mu_j)$$

$$\ln(g(\phi_m)(\mu' - \mu) - \phi'^2(s-1)) - \ln(s\phi'^2) = -g(\phi_m)(\mu - \mu_j)$$

$$\ln\left(g(\phi_m)(\mu' - \mu) - \frac{\phi'^2(s-1)(\mu' - \mu)}{\mu' - \mu}\right)$$

$$\ln(\mu' - \mu)\left(g(\phi_m) - \frac{\phi'^2(s-1)}{\mu' - \mu}\right)$$

$$\ln(\mu' - \mu) + \ln\left(g(\phi_m) - \frac{\phi'^2(s-1)}{\mu' - \mu}\right) - \ln(s\phi'^2) = -g(\phi_m)(\mu - \mu_j)$$

$$g(\phi_m)(\mu - \mu_j) + \ln\left(g(\phi_m) - \frac{\phi'^2(s-1)}{\mu' - \mu}\right) = \ln(s\phi'^2) - \ln(\mu' - \mu)$$

$$g(\phi_m)(\mu - \mu_j) + \ln\left(g(\phi_m) - \frac{\phi'^2(s-1)}{\mu' - \mu}\right) = \ln\left(\frac{s\phi'^2}{\mu' - \mu}\right)$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

$$x = -g(\phi_m)(\mu - \mu_j)$$

$$1 - g(\phi_m)(\mu - \mu_j) + \frac{g(\phi_m)^2(\mu - \mu_j)^2}{2} - \frac{g(\phi_m)^3(\mu - \mu_j)^3}{6}$$

$$\frac{6 - 6g(\phi_m)(\mu - \mu_j) + 3g(\phi_m)^2(\mu - \mu_j)^2 - g(\phi_m)^3(\mu - \mu_j)^3}{6}$$

$$\underline{\mu' = \mu + \phi'^2 \; g(\phi_n)(s - E(\mu, \mu_n, \phi_n))}$$

$$g(\phi_n)(s - E(\mu, \mu_n, \phi_n)) = \frac{\mu' - \mu}{\phi'^2}$$

$$s\,g(\phi^n) - \frac{g(\phi_n)}{1 + e^{(-g(\phi_n)(\mu - \mu_j))}} = \frac{\mu' - \mu}{\phi'^2}$$

$$s\,g(\phi^n) - \frac{6 \cdot g(\phi_n)}{12 - 6g(\phi_n)(\mu - \mu_j) + 3g(\phi_n)^2(\mu - \mu_j)^2 - g(\phi_n)^3(\mu - \mu_j)} = \frac{\mu' - \mu}{\phi'^2}$$

$$v = \frac{1}{g(\phi_n)^2 E(\mu, \mu_j, \phi_n)(1 - E(\mu, \mu_j, \phi_n))}$$

$$\phi^* = \sqrt{\phi^2 + \sigma^2}$$

$$\phi' = \frac{1}{\sqrt{\frac{1}{\phi^*} + \frac{1}{v}}}$$

$$\mu' = \mu + \phi'^2 \; g(\phi_n)(s - E(\mu, \mu_n, \phi_n))$$

$$= \mu + \frac{g(\phi_n)(s - E(\mu, \mu_n, \phi_n))}{\frac{1}{\sqrt{\phi^2 + \sigma^2}} + g(\phi_n)^2 E(\mu, \mu_j, \phi_n)(1 - E(\mu, \mu_j, \phi_n))}$$

$$\mu' = \mu + \frac{g(\phi_n)(s - E(\mu, \mu_n, \phi_n))}{1 + \sqrt{\phi^2 + \sigma^2} \cdot g(\phi_n)^2 E(\mu, \mu_j, \phi_n)(1 - E(\mu, \mu_j, \phi_n))}$$

$$\mu' = \mu + \frac{g(\phi_n)(s - E(\mu, \mu_n, \phi_n))}{1 + \sqrt{\phi^2+\sigma^2} \cdot g(\phi_n)^2 E(\mu, \mu_j, \phi_n)(1 - E(\mu, \mu_j, \phi_n))}$$

$$\mu' - \mu = \frac{g(\phi_n)(s - E(\mu, \mu_n, \phi_n))}{1 + \sqrt{\phi^2+\sigma^2} \cdot g(\phi_n)^2 E(\mu, \mu_j, \phi_n)(1 - E(\mu, \mu_j, \phi_n))}$$

$3g(\Phi_m)$

$$g(\phi_n)(s - E(\mu, \mu_n, \phi_n)) = (\mu' - \mu)(1 + \sqrt{\phi^2+\sigma^2} \cdot g(\phi_n)^2 E(\mu, \mu_j, \phi_n)(1 - E(\mu, \mu_j, \phi_n))$$

$$g(\phi_n)(s - \frac{6 - 6g(\phi_n)(\mu - \mu_j) + 3g(\phi_n)^2(\mu-\mu_j)^2 - g(\phi_n)^3(\mu-\mu_j)^3}{6})$$

$$\frac{6g(\phi_n)s - 6g(\phi_n)(\mu-\mu_j) + 3g(\phi_n)^3(\mu-\mu_j)^2 \cdot g(\phi_n)^4(\mu-\mu_j)}{6}$$

$$(\mu' - \mu)(1 + \sqrt{\phi^2+\sigma^2} \cdot g(\phi_n)^2 \cdot (\frac{6 - 6g(\phi_n)(\mu-\mu_j) + 3g(\phi_n)^2(\mu-\mu_j)^2 \cdot g(\phi_n)^3(\mu-\mu_j)^3}{6})(\frac{6g(\phi_n)(\mu-\mu_j) - 3g(\phi_n)^2(\mu-\mu_j)^2 + g(\phi_n)^3(\mu-\mu_j)^3}{6}))$$

Data

The data that would be used for training and testing consists of 1000 matches created through the simulation part, each fight being against a different fencer.

The score now can be a win/draw/lose, not like before when it could have been just a win/lose.

For each fight, I calculate sigmoid((rating – enemy_rating) / 300) which are the chances of the fencer winning against the current enemy.

If the absolute difference between rating and enemy_rating is near 0, I give it and 80% chance to end in a draw, otherwise, I take the chances from above and use a Bernoulli distribution to find if the player one or lost. This way of finding the score affects if there is a significant difference between the fencers. If the enemy has a higher rating than the fencer, our past experiences tell us that the chances to win are lower for the fencer.

```
score_chances = sigmoid((real_rating - real_hema_rating_enemies) / 
300)

scores = np.array(np.random.binomial(1, score_chances, matches), 
dtype="float")

for i in range(len(scores)):

   sig = sigmoid((np.abs(real_rating - real_hema_rating_enemies[i])) / 
300)

   if  sig <= 0.55 and sig >= 0.45:

       draw = np.random.binomial(1, 0.8, 1)

       if draw == 1:

           scores[i] = 0.5
```

The first 800 fights are training data

The last 200 fights are test data

Bayesian linear regression

I created a simple linear regression, where there are 6 inputs:

- Current rating
- Current RD
- Enemy rating

- Score of the match
- New rating
- New RD

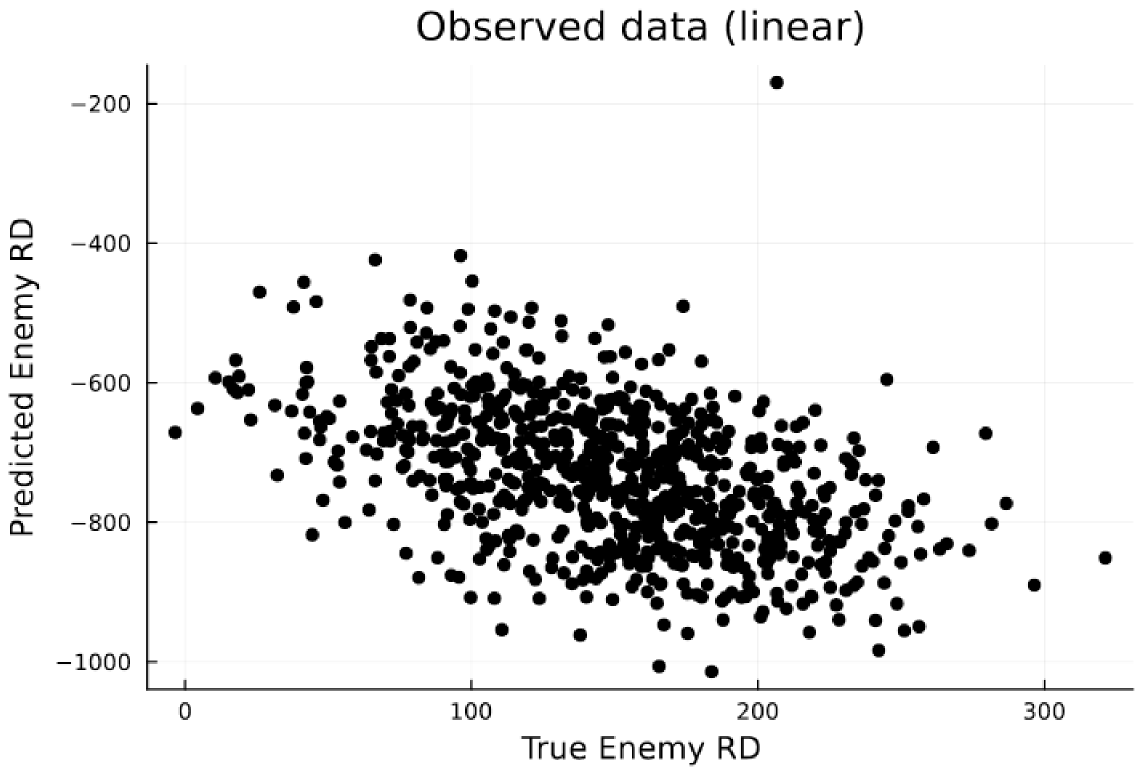
And the output was the approximate value of an enemy RD.

For each input, I created a weight for each input that connects the output, with different prior distributions:

1. Normal(0, 1)
2. Normal(0, 2)
3. Normal(0, 3)
4. Uniform(-2, 2)

I also have added a bias with a prior distribution of Normal(0, 2).

Just by calling the model without having it anything learned, we observe that it has just random values.



```
function
do_inference(model, xs, ys, amount_of_computation)

observations = Gen.choicemap()

for (i, y) in enumerate(ys)

    observations[(:y, i)] = y

end

(trace, _) = Gen.importance_resampling(model, (xs,), observations,
amount_of_computation);

return trace

end;
```

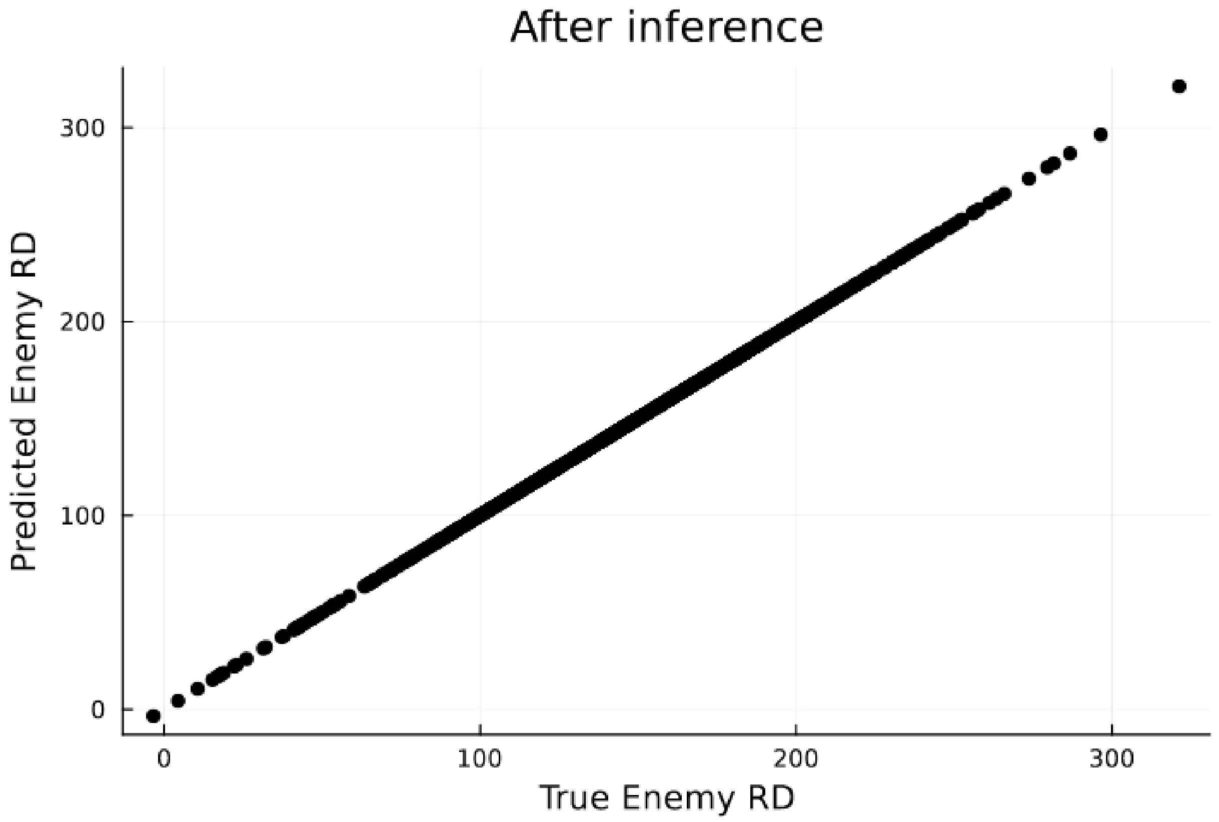
After this, in order do to inference only on the weights and bias, I've added constraints for each predicted value from the train data.

Then called Gen.importance_resampling() which takes in the model, the data and return the trace that fitted the best with the observations.

After calling the inference for about 100 times,

```
y_gen = []
choices = get_choices(trace)
for i in 1:800
    push!(y_gen, choices[(:y, i)])
end
print(y_gen)
```

We get all the y's from the trace.

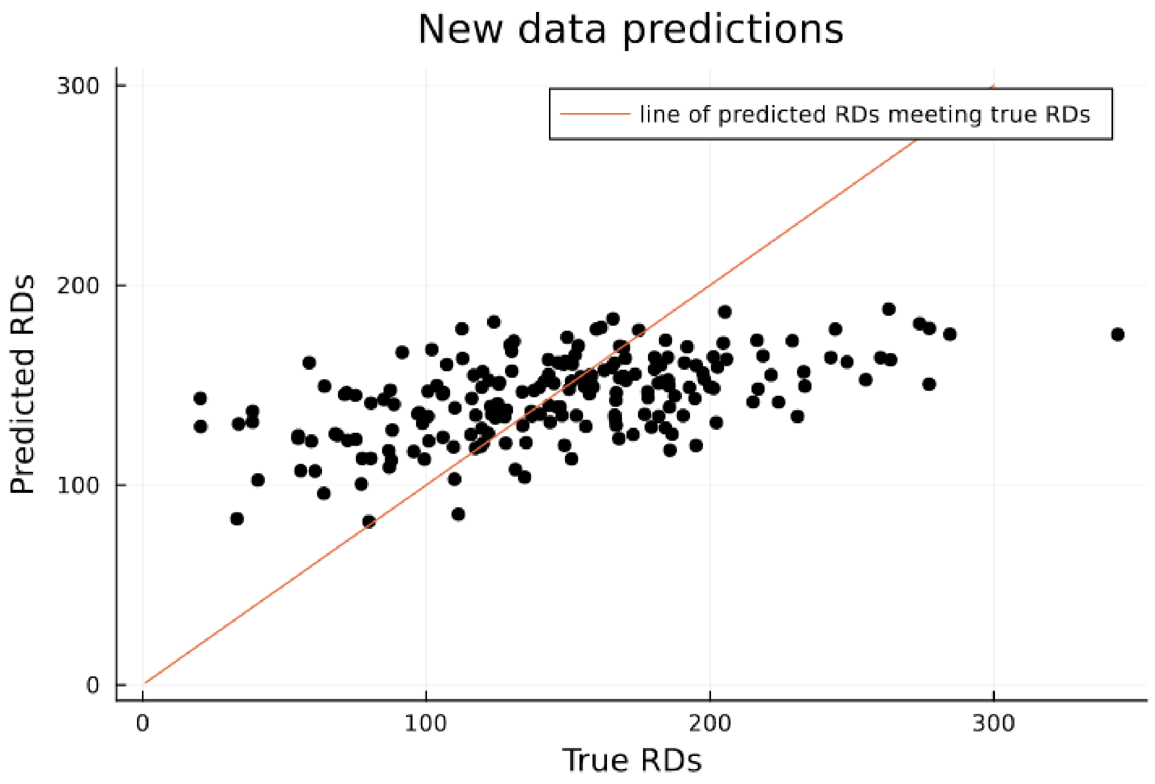


As we can see, it learned pretty well from the data we gave, now it is time to test it with new data.

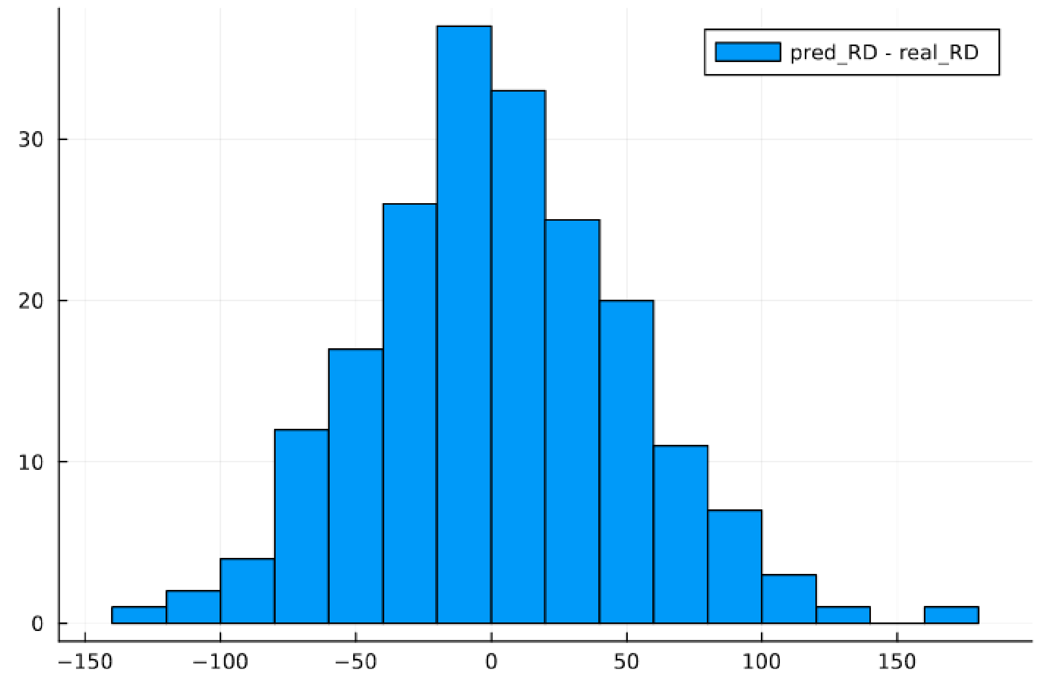
We can create a trace of a generative function using the API method Gen.generate() in which the values of specific random selections are limited to predetermined values. The restrictions are a choice map that links the addresses of the restricted random selections to the values they should have.

By executing fresh iterations of the model's generating function in which the random choices corresponding to the parameters have been confined to their inferred values, we will be able to predict the value of the y-coordinates at new x-coordinates.

We then do inference and predict new data.



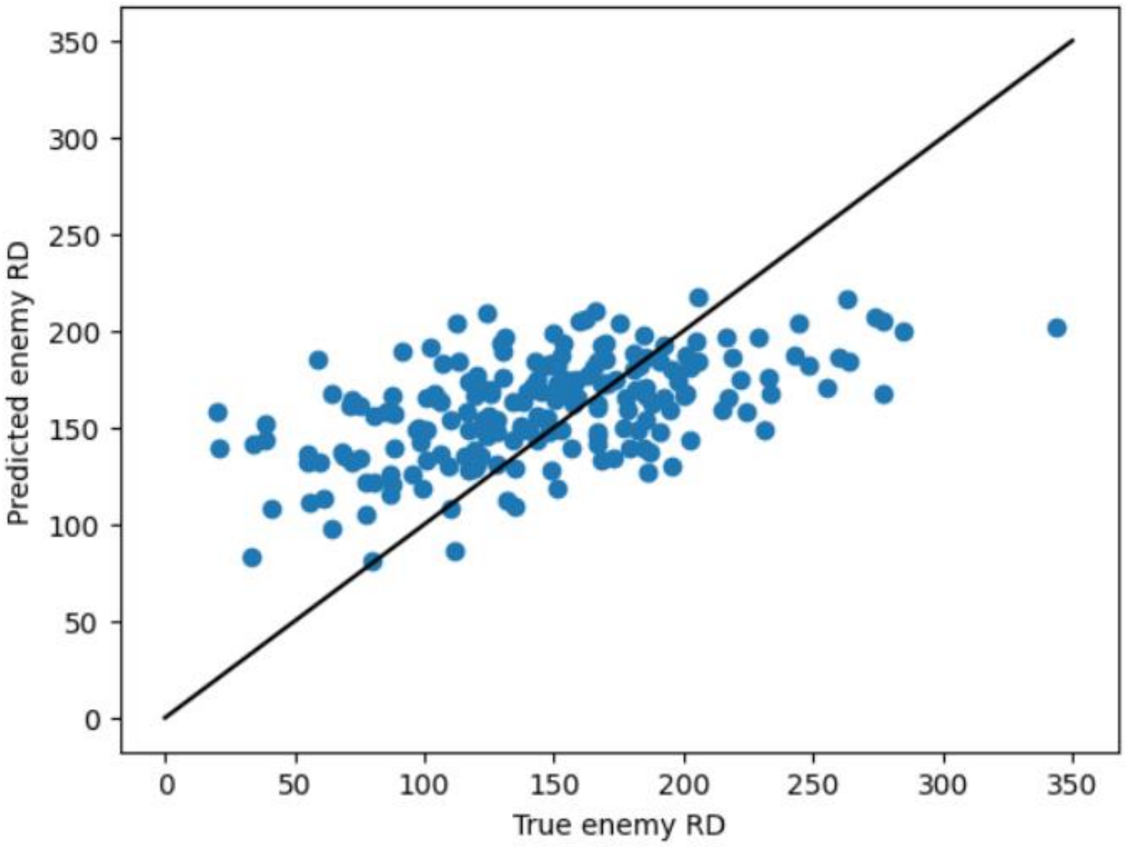
As we can observe, it has learned pretty well when comparing with the random choices. Most of the predicted RDs are very close as values to the real ones.



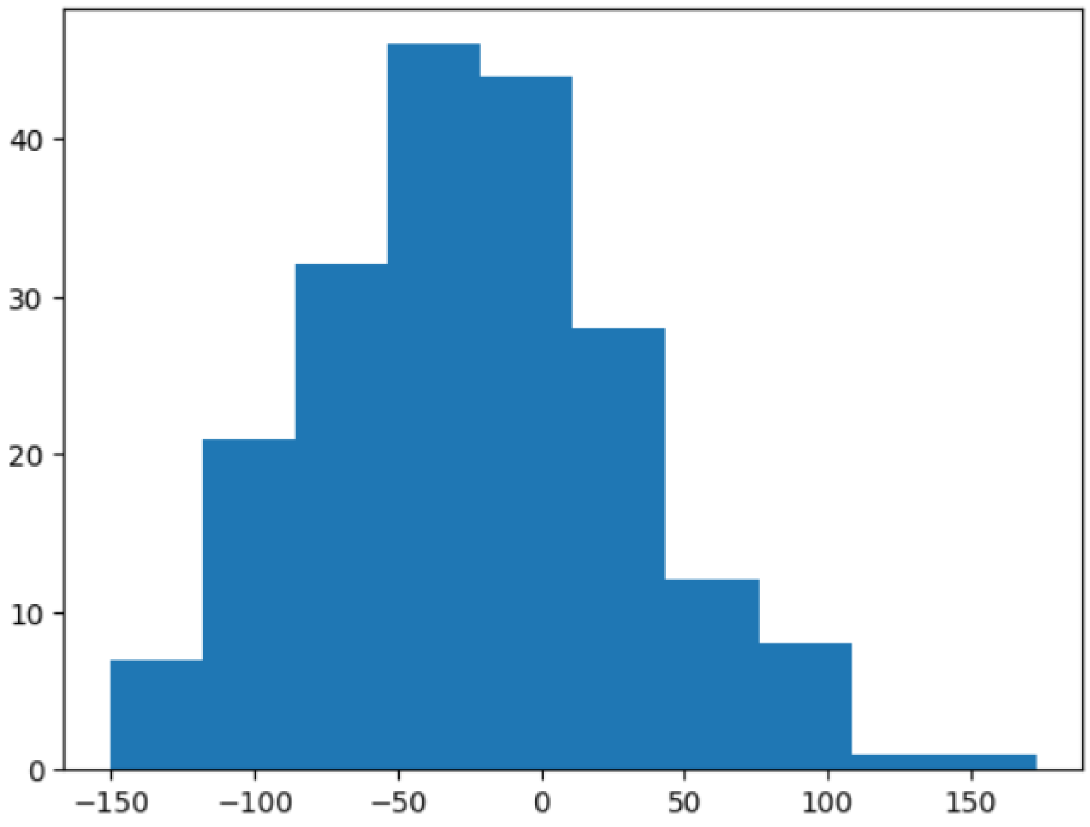
Only a few were very distant (100 or more), quite a few were distant (between 50 and 100) and most of them were quite close (50 or less).

Calculating the average difference between predicted RDs and true RDs we get the value: 37.79. But this value does not help us understand if the model learned good or not, we need to compare it with a typical, simple neural network that has the same architecture as this one.

The predictions from a simple regression with NN:



Predicted enemy RD
True enemy RD
0
50
100
150
200
250
300
350

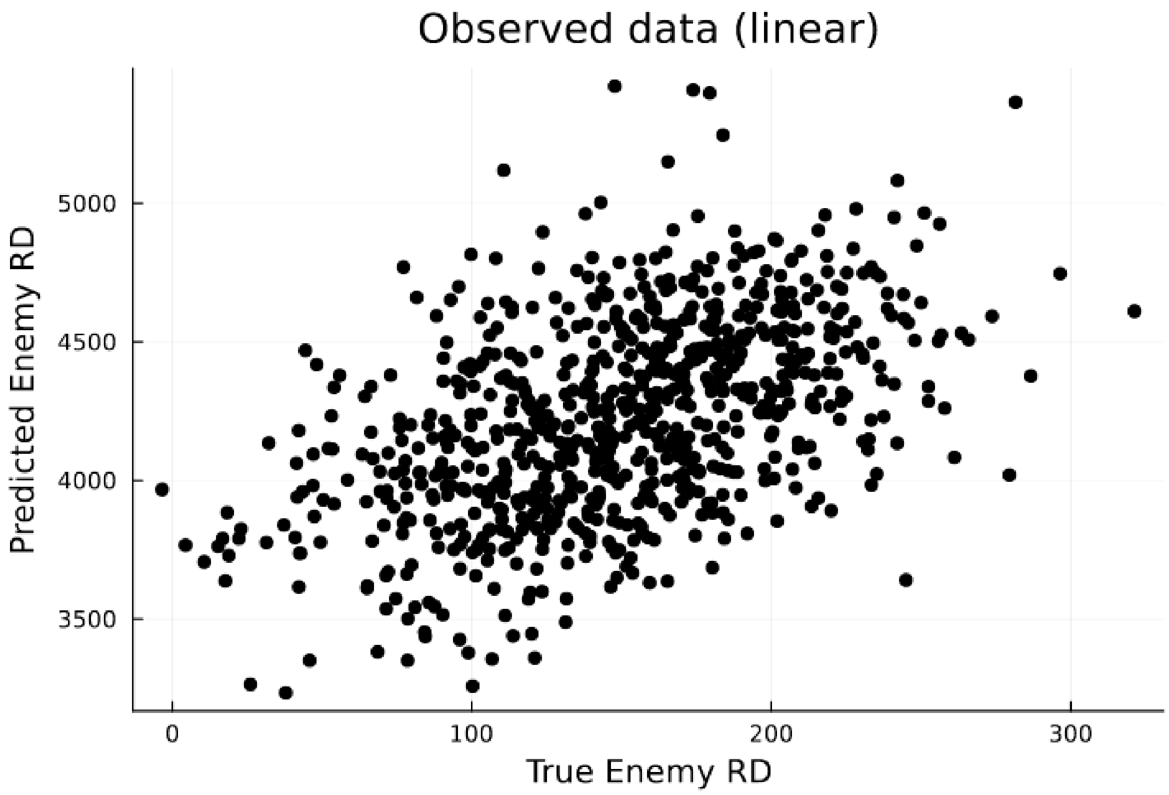
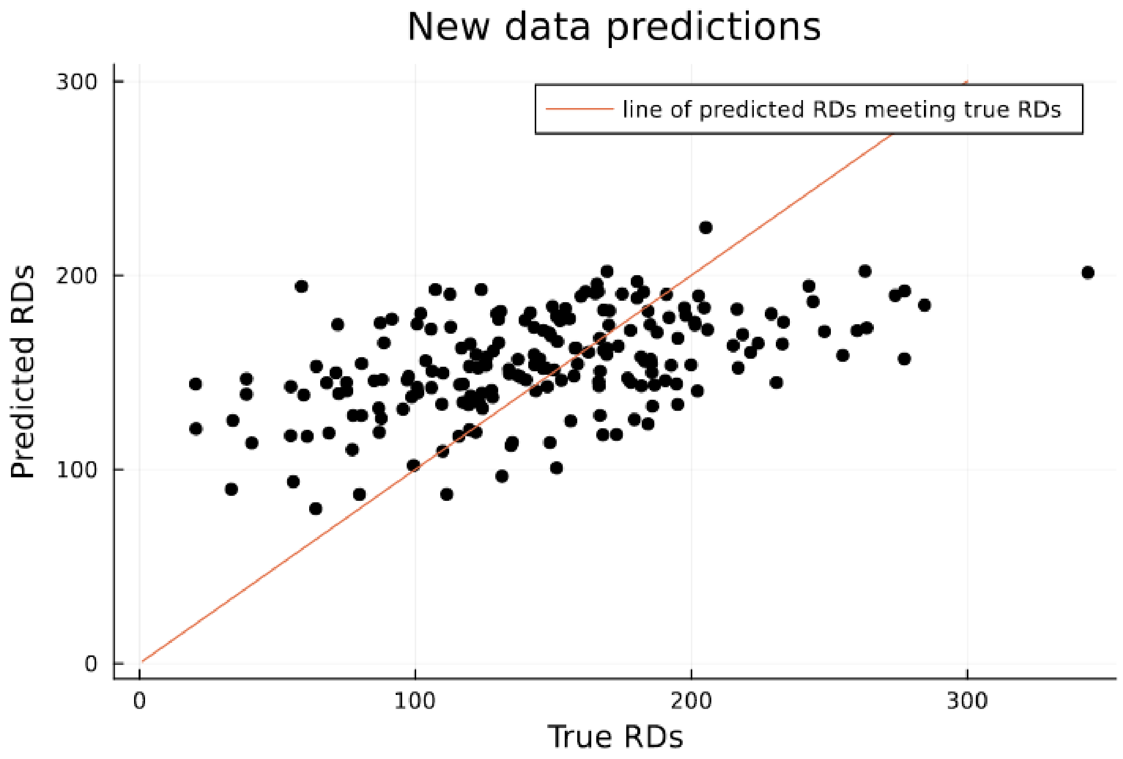


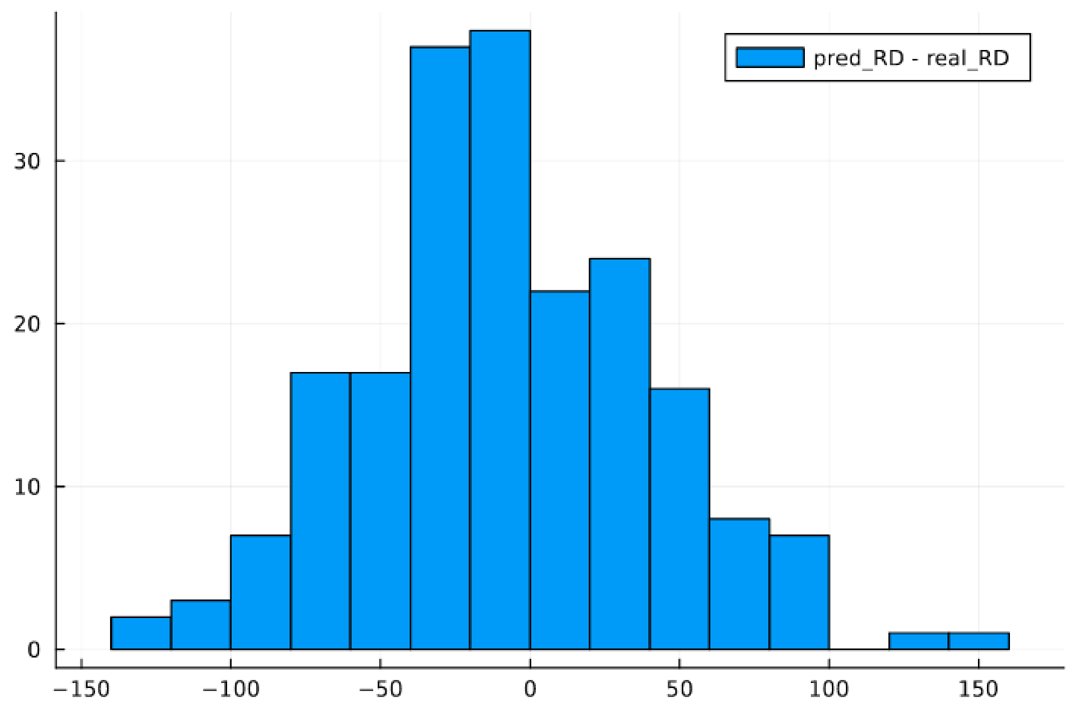
The average difference between true RD and predicted RD is 48.56.

So, the Bayesian model did better than simple NN when using the same architecture, 37.79 vs 48.56.

To further explore what we could do with this model, I have modified the prior distributions:

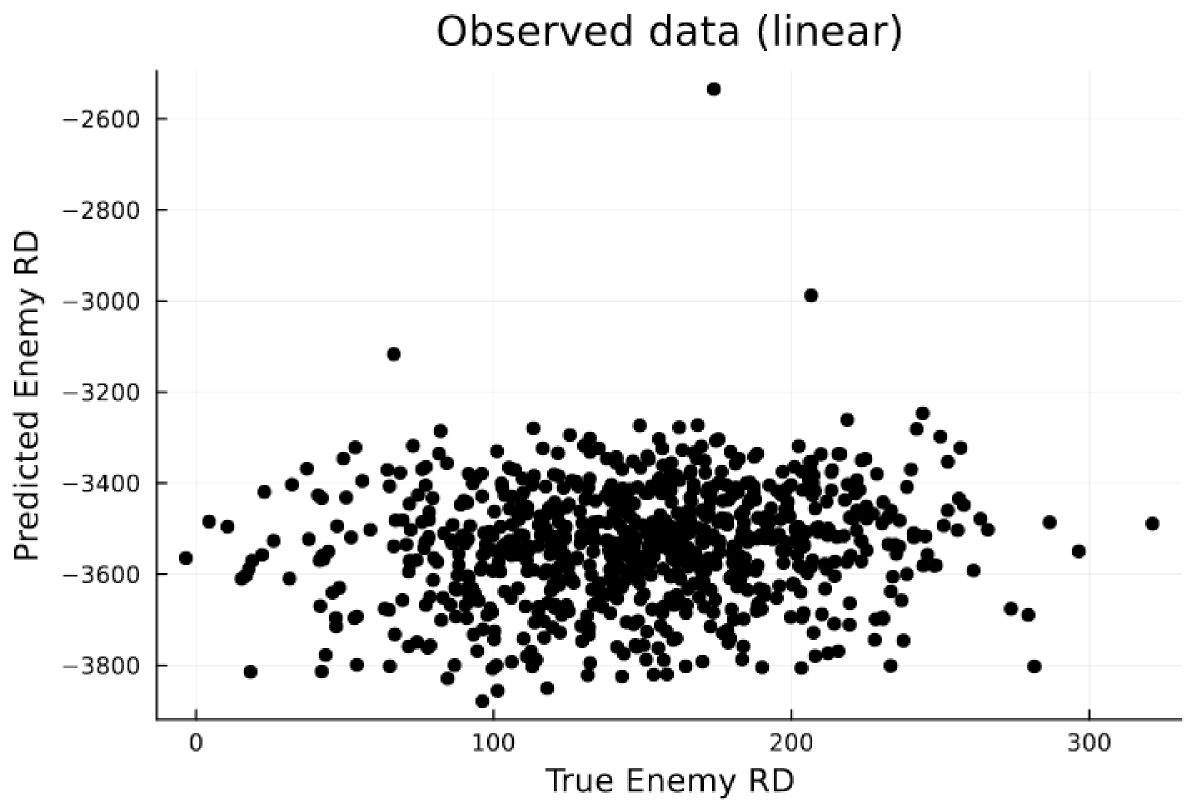
Normal(0, 2):

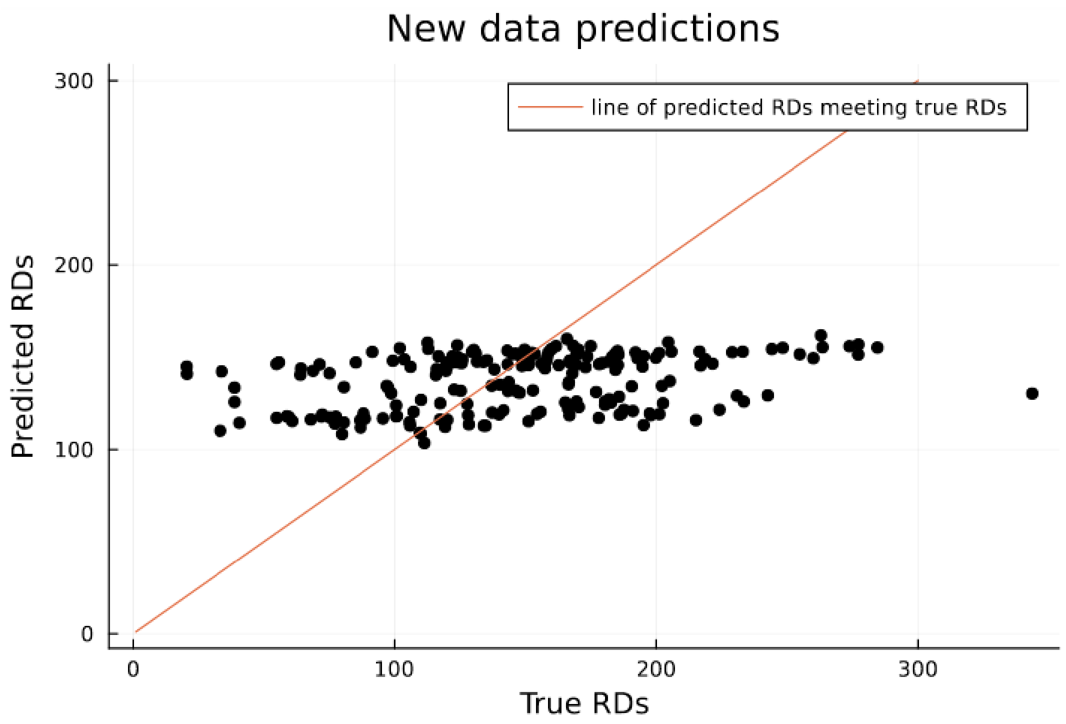

Observed data (linear)
Predicted Enemy RD
True Enemy RD
5000
4500
4000
3500
0
100
200
300

New data predictions
line of predicted RDs meeting true RDs
Predicted RDs
True RDs
300
200
100
0
0
100
200
300

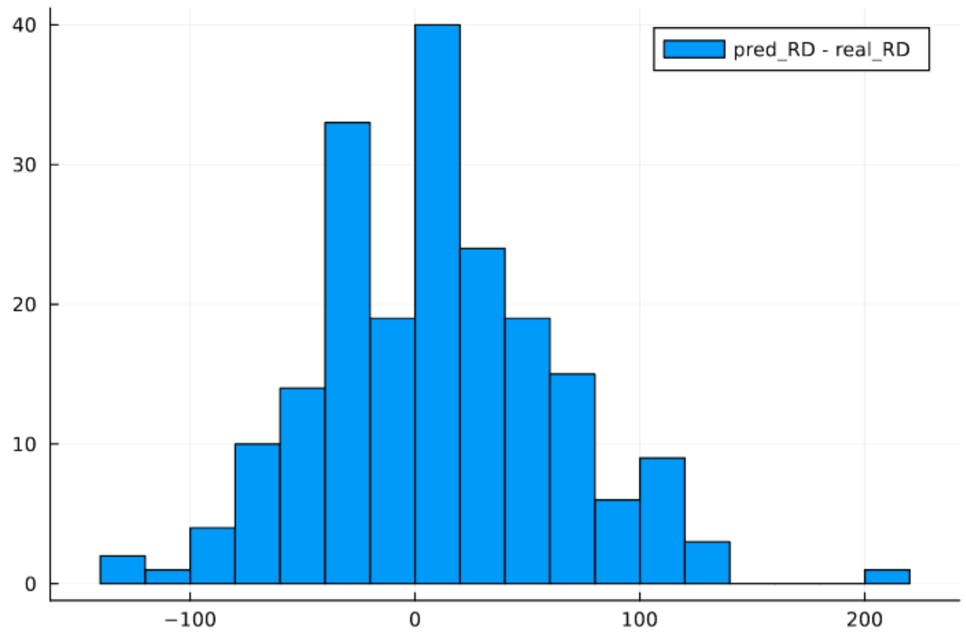


Average difference: 39.63

Normal(0, 3):

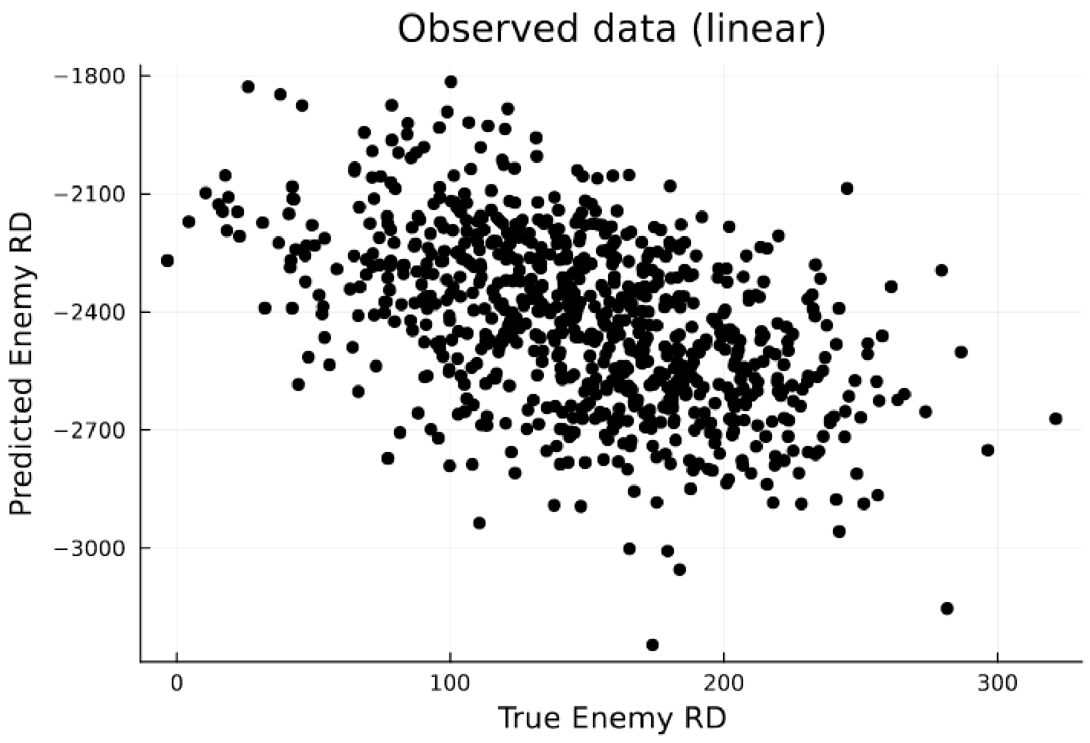

Observed data (linear)
Predicted Enemy RD
True Enemy RD
−2600
−2800
−3000
−3200
−3400
−3600
−3800
0
100
200
300

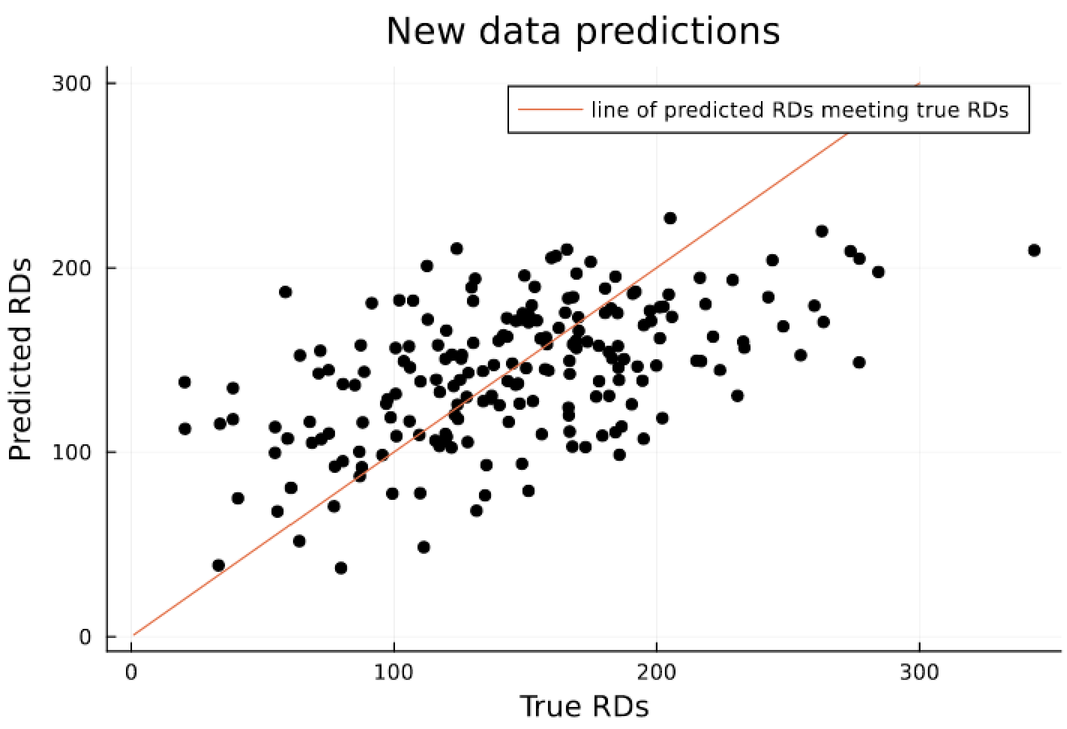

New data predictions
line of predicted RDs meeting true RDs
Predicted RDs
True RDs
300
200
100
0
0
100
200
300

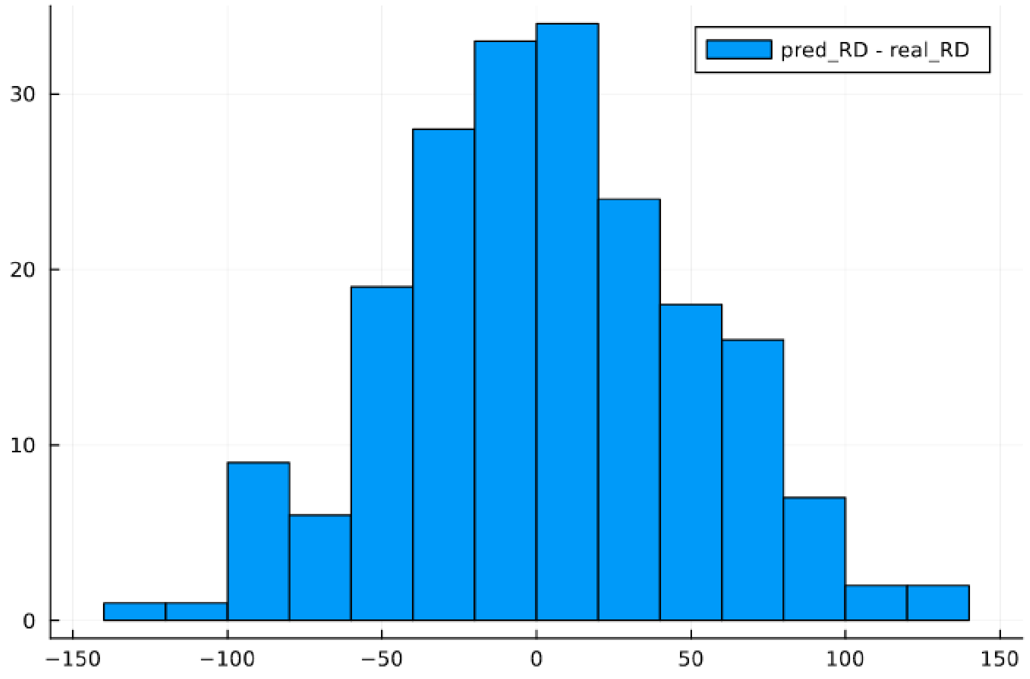


Average difference: 41.91

Uniform(-2, 2)





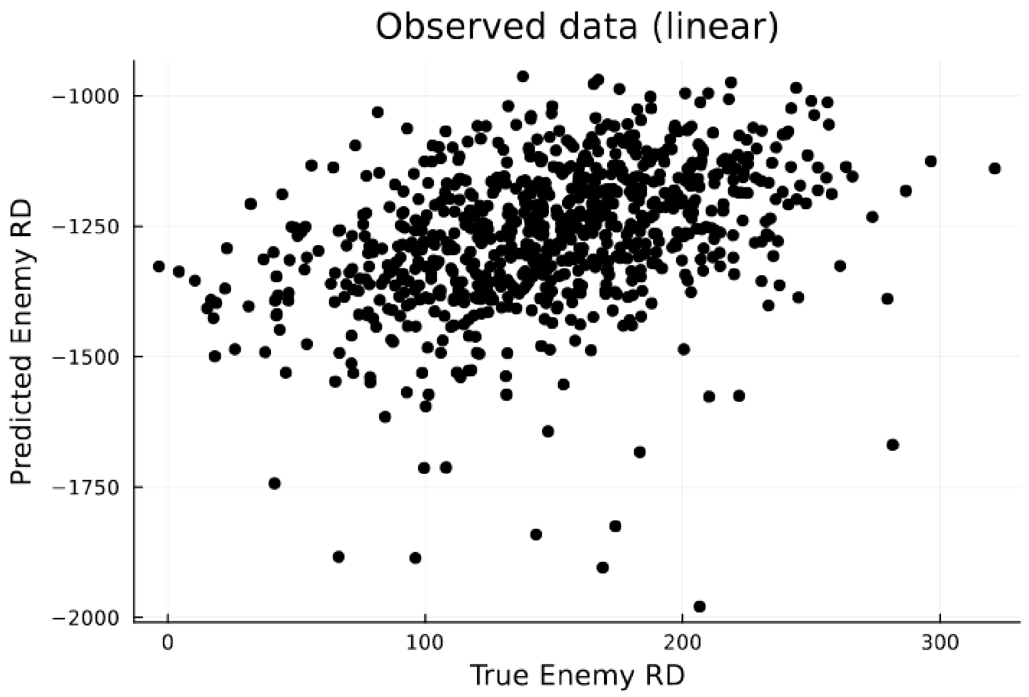


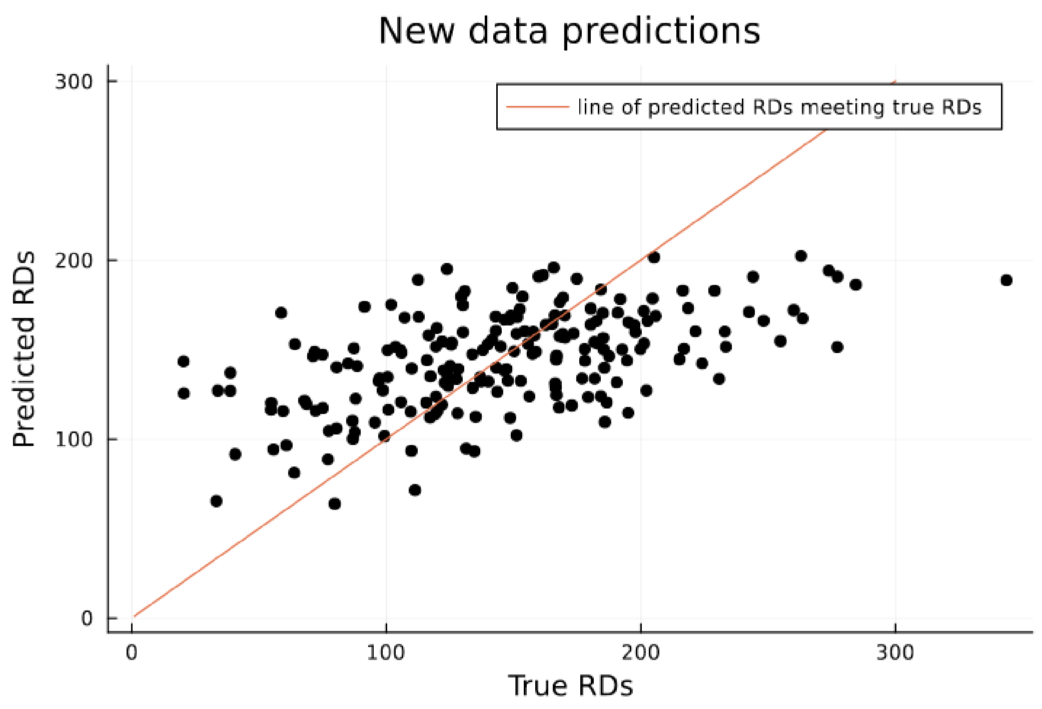
Average difference: 38.18

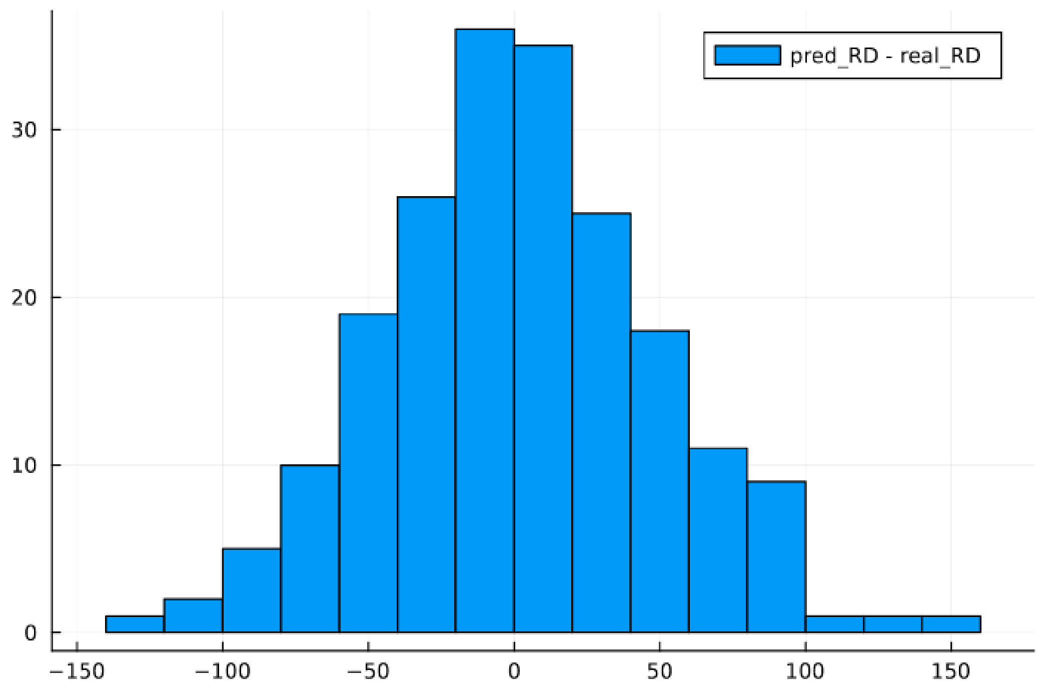
We can observe that the uniform is less compact in the middle and more spread throughout its predictions.

Complex Bayesian Linear Regression

Originally, I tried implementing in such way that you could choose the number of hidden layer to introduce, and the number of neurons on each layer. Unfortunately, it did not work the way I wanted to in regards with tracing and constraints, so a working, more complex linear regression contains a hidden layer of 3 neurons, each connected to the input layer and the output layer.







Average difference: 37.32

By a small difference, the more complex model is a bit more accurate than the simple one, 37.79.

Conclusion

Through multiple comparisons between each other and with a NN, the more “complex” Bayesian model has the best accuracy with just 1000 samples of self generated data. Because of this, ideas that could further the research would be managing a correct way of implementing even more complex models, use even less data and compare it with a simple NN (maybe 100 train and 50 test?) and writing polynomial solver in order to pursue the dropped idea from the beginning.

Bibliography:

- https://hemaratings.com/
- https://www.gen.dev/tutorials/intro-to-modeling/tutorial
- http://www.glicko.net/glicko/glicko2.pdf