NeurIPS - Ariel Data Challenge 2024
kaggle.com

1. Overview of the solution

I relayed at large on polynomial fitting, similar to what was done by @sergeifronov here with some modifications: I estimated the start/end of the ingress by fitting two 2nd degree polynomials that are connected with a line on the first half of the signal and the same on the second half (explained more in-depth later). Also, instead of using scipi.optimize.minimize, I used binary search, and instead of fitting a polynomial on all the signals, I fitted on the start+middle and middle+end separately, i.e., I got a separate prediction for the ingress and the egress parts.

I found that the ingress calculations were more accurate (in particular when fitting 2nd-degree polynomials- when fitting 3rd-degree, they are more similar in terms of accuracy but less accurate overall than the 2nd option). So, I gave them a larger weight. For sigma estimation, I fitted a 2D polynomial, utilizing a strong correlation between mean(sigma|pred) and std(pred) and a weak correlation between mean(sigma|pred) and the difference between the predictions on the ingress part and the egress part (i.e., if they are more similar, then sigma is smaller). I also utilized several postprocessing methods, including averaging on the wavelength in a wider window for lower-std preds, using mean(pred+pred*small_factor (i.e., adding fluctuation) for low-std predictions, and replacing preds with a mean(pred) for high wavelengths (the noisy ones). With all of this, I got 0.688/0.692.

Then came the jump to 0.703/0.715: I used TensorFlow to perform 2-dimensional polynomial regression, think Sergei's method but with polynomials also in the wavelength axis. This model was 0.691/0.703, and I ensembled two variations of it together with the first model (688/0.692) for my final score.

1.1 Crude baseline

1.1.1 Finding the ingress/egress start/end

I estimated the middle of the ingress/egress by finding the max/min of the first derivative of the signal, averaging the signal on all the wavelengths, and averaging in time with a rolling window the size of 20. Then, I estimated the start/end of the ingress/egress by the first point before/after the middle of each, where the first derivative becomes 0. It's very crude, but it was enough for a baseline. Then I took the mean in a window the size of 50 before the start of the ingress/egress and after their end and used the difference to calculate the reduction in the signal, i.e., the predictions.

1.1.2 Mean sigma per star

For stars 1/2, I used the RMSE between the predictions and the targets; for stars 3/4, I used 0.00016 (I don't remember if I took it from a public notebook or probed it myself). This resulted in a public/private 0.524/0.561

1.1.3 Fixing the predictions

I noticed that the mean of the predictions was lower than the mean targets, so I started to probe the best factors to fix it. This was the result:

pred = np.where((np.expand_dims(star, axis = -1)==star*0) == 0, pred+0.000083053, pred)

pred = np.where((np.expand_dims(star, axis = -1)==star*0) == 2, pred+0.00013, pred)

pred = np.where((np.expand_dims(star, axis = -1)==star*0) == 3, pred+0.00011, pred)

This correction improved my score to 0.543/0.578. This is already bronze medal range.

Later, I found a linear correlation between mean(preds) and mean(preds-targets), i.e., that I need to fix per plane with a multiplicative factor. After the competition ended, it turned out that this was due to the foreground; see here.

1.2 From public 0.543 to public 0.638

1.2.1 Better estimation of ingress/egress

I was not satisfied with the derivative method, fearing that it would miss noisy signals, especially in the private test that might be much more noisier. Also, it necessitated averaging the signal in time, leading to a loss of precision. Finally, I developed the following method:

I first divided the signal in the middle. Then, for the first half of the signal, I defined two points, p1 and p2, and fitted the following polynomials:

1. A 2nd-degree polynomial from the start of the signal to p1.
2. A 2nd-degree polynom from p2 to the end of the first half of the signal (under the assumption that ingress would always be in the first half and egress would always be in the second half)

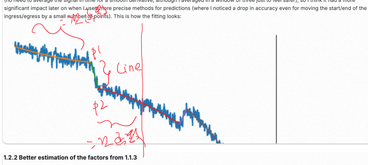
Then, I connected the functions 1 and 2 with:

.3. A line that connects the first polynomial's end with the second one's start.

Then, I varied p1 and p2 on the signal indices and searched for the combination that would give the lowest RMSE between functions 1+2+3 and the signal. First, I varied in strides of 100, then I varied in strides of 20 on the best section from the first iteration, and finally, I varied in strides of 1 inside the best section from the second iteration, enabling me to focus on the best start/end (p1/p2) of the ingress in about half an hour or so for all the planets by utilizing efficient polynomial fitting (in Numba) and multiprocessing. Then, I did the same for the second half of the signal with p3/p4 for the start/end of the egress.

I performed this fitting on the mean signal under the assumption that the ingress/egress are the same for different wavelengths. This assumption was supported by looking at the mean signal for wavelength from the lower end and from the higher one.

This improved my score to 0.545/0.583. It was a small improvement, but I was very satisfied since I deemed it more robust. Also, it is precise in time (no need to average the signal in time for a smooth derivative, although I averaged in a window of three just to feel safer), so I think it had a more significant impact later on when I used more precise methods for predictions (where I noticed a drop in accuracy even for moving the start/end of the ingress/egress by a small number of points). This is how the fitting looks:



1.2.2 Better estimation of the factors from 1.1.3

A series of probing led to better factors stars 3/4, resulting in 0.55/0.588 public/private.

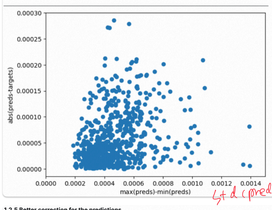
1.2.3 Improving the predictions

First, instead of estimating the signal before/after the ingress/egress by taking the mean signal before/after in a window of 50, I fitted a 2nd-degree polynomial up to p1, between p2 to p3 and from p4 to the end of the signal, and used their values in the middle of the ingress/egress to estimate the value of the signal for predicting the targets. I did it per wavelength for an averaged signal on the wavelength axis with a rolling window of 30 wavelengths. Then, I estimated the target by the mean of the predictions for wavelengths 0-220, excluding the predictions for the longer wavelengths from the mean since they were too noisy and only hurt the prediction.

1.2.4 Estimating sigma per planet

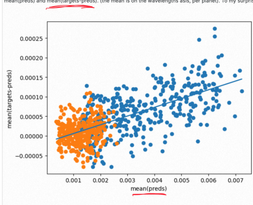
At this point, I still predict the mean target per planet but the mean sigma per star. I looked for a correlation between the mean sigma per planet and ADAMETHOD and finally found some correlation with the mean(pred) minimized up to a wavelength of 220, henceforth dimmer as "dif". Remember from 1.2.3 that I have, at this point, predictions per wavelength (averaged in wavelength window the size of 30) even though my final prediction is the total mean. So I can calculate this 'diff.' I predicted sigma as the mean sigma on the train set between different thresholds of diff (thresholds = [0, 0.0003, 0.0004, 0.0005, 0.0006, 0.0007, 0.00085, 1000]).

1.2.3+1.2.4 improved my score to 0.602/0.614. Here is how the correlation looks:



1.2.5 Better correction for the predictions

Let's go back to 1.1.3. Remember that I added a constant per star to the prediction? So, at some point, I tried to check the correlation between mean(preds) and mean(targets-preds). (the mean is on the wavelengths axis, per planet). To my surprise, this is what I found:



As I said in 1.1.3, it turns out it's due to the foreground; see here.

Replacing the correction terms from 1.1.3 with a linear correction found from the above regression resulted in a 0.619/0.627 score. This is already in the silver range.

1.2.6 Predicting per wavelength

Remember the 'diff' from 1.2.4? So, I found that for small diff, I can't predict per wavelength, but for large, I can (or rather, for large diff, the errors in predicting per wavelength are smaller than the errors from predicting the mean- think about a signal that varies between extreme values). In any case:

pred = np.where(np.expand_dims(diff, axis = 1)>features*0+0.00085, features, pred_0)

Here, 'features' are the 'raw' predictions per wavelength, and 'pred_0' is the mean prediction per planet. This is 0.626/0.632 public/private.

1.2.7 Longer wavelengths are too noisy

I added:

pred[:, 230:] = pred_0[:, 230:]

i.e., for longer wavelengths, I predict the mean even for large diff. This was a small improvement, 0.628/0.632.

1.2.8 Adding fluctuation to the mean signal

pred_0 = pred_0+(features-np.mean(features, axis = 1, keepdims = True))*0.08

This is a trick I saw also at some other solution, maybe 3rd place? I don't remember exactly. This was a small improvement, 0.630/0.635.

1.2.9 Improving diff

Remember 'diff'=max(pred)-min(pred) from 1.2.4? So, I replaced it with diff=std(pred) for all purposes. This improved my score to 0.638/0.637.

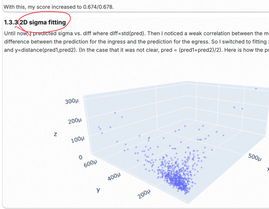
1.3 From 0.638 to gold range

1.3.1 Better prediction- Sergei's method

I read Sergei's notebook and got inspired, so I utilized the same method of searching for the prediction value that will give the lowest RMSE for a polynomial fitted on both the middle (shifted buy the prediction) and the sides. I used a custom binary search and calculated the ingress part (start+middle) and egress part (middle+end) separately. Later, I discovered that fitting a 2nd-degree polynomial in my method is equivalent to a 3rd-degree on the entire signal (start+mid+end), and a 3rd-degree in my method is equivalent to fitting a 4th-degree on the entire signal. In any case, I fitted a 2nd-degree polynomial- I knew they do not fit for some signals that need at least a 3rd-degree. Still, they consistently gave me better results than 3rd-degree polynomials despite my efforts to find ways to utilize higher-degree polynomials without hurting my score. With this, I jumped to 0.673/0.658 public/private.

1.3.2 Better estimation of sigma

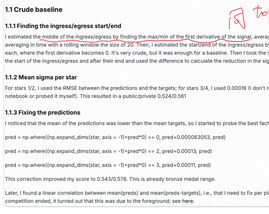
In 1.2.4, I predicted different mean sigmas for different thresholds of diff. I replaced it with a linear regression between every two thresholds instead of a constant. It looks like this:



With this, my score increased to 0.674/0.678.

1.3.3 2D sigma fitting

Until now, I predicted sigma vs. diff where diff=std(pred). Then I noticed a weak correlation between the mean sigma per planet and the absolute difference between the prediction for the ingress and the prediction for the egress. So I switched to fitting 2D polinomial with z=sigma, x=std(pred) and y=distance(pred1,pred2). (In the case that it was not clear, pred = (pred1+pred2)/2). Here is how the points look in the 3D space of XYZ:



1.3.4 Larger weight to pred1

I discovered that giving more weight to pred1 (the one on the ingress) results in a significant improvement. I weighted it 1.9:1, and my score improved to 0.681/0.695. This is gold range both in public and private.

1.3.5 Curvature/slope weighting

I gave more weight to pred1|pred2 relative to the other for more similar curvature/slope on both sides of the ingress/egress. This gave me 0.682/0.686.

1.3.6 Smart smoothing+fluctuation

I averaged the signal on a larger wavelength window for smaller std(pred). In addition, for lower std, I added stronger fluctuations for shorter wavelengths to the mean (probably would be clearer in my code). Anyway, with this, I reached 0.688/0.692.

1.4 Breaking over public 0.7

When I looked at the signal, I saw that the drift in time is similar for all the wavelengths but with the occasional gradual drifting in slope/curvature. Since they seem correlated, I wanted to fit a 2D polynomial in time and wavelength so that my fitting would rely on more data and be more accurate. In 1st place solution, they noted that the drift was modelled as f(time)*g(wavelength), so it would be better to fit two multiplied 1d polynomials instead of a 2d one).

What I did is to turn the problem into a regression problem on all the parameters together: I let the reduction in the signal for each wavelength be a parameter, and I also let the coefficient of a 2D polynomial be parameters; I defined the loss as the RMSE between the fitted polynomial and the signal with the middle shifted by the corresponding parameter for each wavelength, put everything inside a TensorFlow model with the loss I described, attached an Adam16 optimizer and half-cosine LR decay as usual and let it 'train.' Then, all I needed to do was retrieve the parameters that define the reduction of the signal per wavelength and calculate the value of the fitted polynomial at the ingress/egress to predict the targets.

I also used a normalization to bring all the wavelengths to the same scale: I subtracted from each wavelength its mean before/after the ingress/egress (excluding the middle), then divided each subtracted wavelength by said mean divided by the largest mean (so the wavelength with the largest mean was divided by 1, and wavelengths with lower means were divided by a factor smaller than one). I also normalized the loss by the std(signal) after the subtraction and divide. I carried all of it on a signal that I binned in the wavelength axis with a binning of 30 and, in time, with a binning of 3 to speed up computations.

In addition, to get all of this to run at an acceptable time (at this point, I basically 'train' a tensorflow model for each planet in the test set so it's at least several hours), I constructed my model so that it performs regression on several planets concurrently (basically the planets are aligned on the 'batch' dimension and the loss and parameters for each planet are separate, yet the parameters defined in x vectorized way so that all the calculation are performed together).

Well, to get to the conclusion- I 'trained' concurrently for 256 planets, it was blazing fast on GPU, and it worked great. The first model was a 2nd-degree polynomial in time, with the coefficient of the zero term in time a 2n-degree polynomial in wavelength and the coefficients of t, t^2 (t for time) a 1st-degree polynomial in wavelength. As with the older methods, I performed the regression separately for the ingress and egress parts, gave a larger weight to pred1 (the ingress prediction), and performed the same postprocessing techniques described above. This gave me 0.691/0.703.

Then I 'trained' a second model, this time with a 3rd-degree polynomial in time for the egress part (but still a 2nd-degree polynomial for the ingress as for the first model). I let the coefficient of all the terms of the egress part be a 2nd-degree polynomial in wavelength except for the coefficient of t^3, which I kept as a 1st-degree polynomial in wavelength. For the ingress it was the same as the first model. At this point, I had three models: the old good model from 1.3.6, the first 2D polynomial model, and the second 2D polynomial model. I weighted them equally for the spectrum predictions and 0.2:1:0 for the sigma predictions (I chose the weights based on the training set), and this was my final solution.