Kaggle Plasticc Challenge

Belinda Trotta

December 22, 2018

This challenge requires us to classify objects in outer space into one of 15 categories based on the light they emit at various frequencies, and some basic metadata. Fourteen of the categories are present in the training data; the other category is for objects of types which have not yet been observed.

My solution is implemented in Python and uses LightGBM gradient boosted classification tree models. It scores 0.84070 on the private leaderboard, and runs in around 6 hours on a 24 Gb laptop (including calculating features, training, and prediction). It uses only elementary operations to calculate the features: there's no curve fitting or optimisation, which helps keep the runtime down. Apart from the hints revealed in the forum discussions, my original insights that gave the most improvement in score are: the Bayesian approach to removing noise from the flux measurements, adding features based on scaled flux values, adding features to capture the behaviour around the peak, and understanding how to optimise the metric (including for class 99). All these are described in more detail below.

1 Feature engineering

1.1 Removing noise from flux

Since some of the flux values have large errors, we use a Bayesian approach to estimate the most likely true value. We assume a prior distribution given by the flux observations for the same object and passband, and assume the observed value comes from a distribution whose mean is the true value of the flux, and which has standard deviation `flux_err`, since we are told in the Starter Kit kernel [1] that this is a 68% confidence interval.

We then calculate the mean of this posterior distribution from which we assume the observed value is drawn, using the formula in [2]. Note that we have only a single observed value from the posterior distribution, that is, $n = 1$ in the above calculation. This gives an estimate of

$$\frac{\mu_p/\sigma_p^2 + f/f_{err}^2}{1/\sigma_p^2 + 1/f_{err}^2}$$

where μ_p and σ_p are the assumed prior mean and standard deviation (i.e. the mean and standard deviation of the flux for the given object and passband), f is the observed flux,

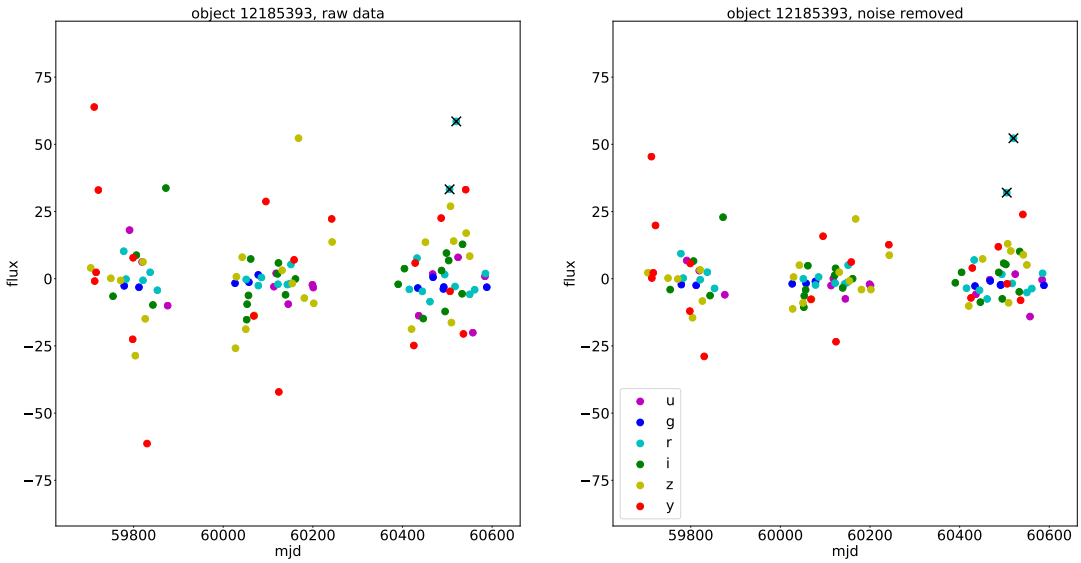

Figure 1: Example of an object's data before and after removing noise. Notice how the noisy y band values are closer to zero, and the detected values (marked with crosses) are more distinct from the other values.

and f_{err} is the flux error. Figure 1.1 shows an example. This normalisation is a similar idea (but not the same) as the normalisation done in the Starter Kit.

1.2 Estimating flux at the source

The closer an object, the brighter it appears (i.e. the higher the flux). But to classify objects into types, it's more useful to know its absolute brightness at the source. From [3], we see that luminosity is proportional to the square of the luminosity distance D, which is proportional to $10^{(distmod/5)}$.

However, for the provided data, a scatter plot shows that this calculated luminosity distance is proportional to redshift (which, according to the graph in the above reference, corresponds to a particular assumption about the parameters ω_M, ω_λ). Therefore, we can use the squared redshifted as the scaling factor, instead of the squared distmod. This is preferable becasue we sometimes have an exact measurement of redshift (in `hostgal_specz`), whereas distmod is calculated from the approximated `hostgal_photoz`.

We apply this correction before doing any further processing. There is a (vague) hint about this in [4].

I experimented with trying to calculate adjusted passbands and times to account for redshift, but this didn't produce any appreciable improvement. I think this is because of the incompleteness of the observation data. We don't know the simultaneous flux in

every observed passband, since only one is measured at a time, so we can't accurately compute the flux in the source passband (because light in a single passband at the source can spread via redshift over several passbands at the observation location). Trying to use the adjusted mjd is also difficult, because there are relatively few observations and they are not evenly sampled.

1.3 Basic features

I calculated some basic aggregate features by object and passband: maximum, minimum, mean, standard deviation, and lower and upper quartile. I calculated these both on the raw data, and on the data after it had been scaled by subtracting the minimum and dividing by the maximum absolute flux. This scaling reveals the shape of the curve, normalising for magnitude.

For each object I calculated the overall maximum detected flux, the passband with the highest flux, the passband with the most detections.

1.4 Extracting the peaks

However these basic aggregate features don't really capture the shape of the peaks that appear in the light curves. Since these peaks can occur at any time during the observation period, we need a way to extract the data from these peaks. I did this by finding the "most extreme" minimum and maximum times for each object, defined as the time with detection = 1 when one of the object's passbands differs the most from its median value. Then, for each passband, I retrieve the closest flux value before and after this peak time. I do this for both the raw (but de-noised) flux and the max-scaled flux. Also, I calculate the "duration" of the peak in each passband, defined as the period of continuous detections around the peak, and the time difference between the overall peak and the individual passband peaks.

From Section 1d of the Starter Kit kernel [1], we see that the way that the flux drops after the peak is an important feature for distinguishing various kinds of transient objects. To capture this, I added features measuring the mean flux in each passband for various intervals before and after the peak.

1.5 Periodicity and the "spread" of peaks

I added the feature recommended in [5], which measures the longest time between detections for each object.

This helps identify periodic objects: for objects with only a single peak, this value will be small, whereas for periodic objects having several peaks the value will be larger. This feature gives a large improvement in accuracy. As well as that, though, this thread was very useful to me more generally because it encouraged me to make use of the `detected` field, which previously I had been ignoring.

Inspired by this feature, I calculated some related ones. I calculated the same feature by passband. Also, I calculated the maximum distance between "high" detections,

defined as times when `detected` equals 1 and flux is at least 0.75 times its maximum value. I also calculated the standard deviation of the times of the detections in each passband. Finally, I calculated some aggregates (minimum, maximum, standard deviation, median and proportion of detections) for each passband during the interval between the minimum and maximum detection.

2 Model

The model is gradient-boosted classification trees, implemented in LightGBM. There is a separate model for each class. As observed in [6], each class in the training data consists either of only galactic objects or only extra-galactic objects. Moreover, the kernel notes that these can be distinguished by the fact that galactic objects have `hostgal_photoz` equal to 1 (as also mentioned in the data note). Therefore for the galactic classes, we train the model only on the galactic data, and similarly for the non-galactic models.

For each class, we define separate models for exact and approximate redshift, since in the case that exact redshfit data is available (i.e. `hostgal_specz` is not null), we should use this. However when exact data is not available, we need to use the estimated data in `hostgal_photoz`. In order for the model to use this data properly, it needs to be trained on data that excludes `hostgal_specz` (otherwise, it will never split on `hostgal_photoz`, since `hostgal_specz` has more predictive power).

The test data is quite different from the training data. To prevent overfitting to the training set, I used early stopping in LightGBM with a validation set sampled from the training data but weighted to resembed the test data. Specifically, the data note says "The training data are mostly composed of nearby, low-redshift, brighter objects while the test data contain more distant (higher redshift) and fainter objects." So I resampled to acheive a similar distribution of `hostgal_photoz`, and excluded objects with `ddf = 1` from the validation set (since these comprise only 1% of the test set).

3 Code structure

Because the test set is so large, we process it in chunks. First we run `split_test.py` to break the test data into chunks (in such a way that the objects are not split accross chunks) and save it as hdf5 files for faster reading. This only needs to be done once and takes around 15 minutes. The features are calculated in `calculate_features.py` (this takes around 4 hours to run). The code to train the model and predict is in `predict.py`. This produces raw predictions for each class and saves them in a csv file, taking around 1.5 hours to run. Finally, we run `scale.py` to apply scaling to each class, and regularisation to the class 99 prediction (described in more detail below). This scaling script is separated from the modelling code because the modelling code is slow to run, and this way we can quickly try out different regularisation parameters using the previously saved model output. The scaling script runs in around 1 minute.

4 Finding the weights

The evaluation metric is

$$-\left(\frac{\Sigma_{i=1}^{M} w_i \Sigma \frac{y_{i,j}}{N_i} \ln(p_{i,j})}{\Sigma_{i=1}^{M} w_i}\right),$$

where M is the number of classes (15), w_i are the class weights, which are not given, N_i is the number of positive samples in each class, $y_{i,j}$ is 1 if the j^{th} data point is in the i^{th} class, and $p_{i,j}$ is the predicted probability that the j^{th} data point is in the i^{th} class. If the prediction is equal to a constant value p_i for each class i, this reduces to:

$$-\left(\frac{\Sigma_{i=1}^{M} w_i \ln(p_{i,j})}{\Sigma_{i=1}^{M} w_i}\right).$$

This makes it possible to find the weights w_i by probing the leaderboard. The thread [7] contains the loss metrics obtained by submitting 14 submissions, each predicting all the data points to belong to a single class with probability 1.

In [8] it is claimed that the weights are approximately 1 for all classes except 15, 64, and 99, which have weight 2.

We show here how to derive these from the probed losses obtained by setting all predictions to a single class. Predictions of 0 and 1 values are changed by Kaggle's evaluation algorithm to 10^{-15} and $1 - 10^{-15}$ respectively. (See [9].)

Write $p_0 = \ln(10^{-15})$ and $p_1 = \ln(1 - 10^{-15})$, and $W_{sum} = \sum_{i=1}^{M} w_i$. Then predicting all data points to belong to class i gives a loss of

$$\begin{aligned} L_i &= -\left(\frac{\sum_{k=1,k\neq i}^{M} w_k p_0 + w_i p_1)}{w_{sum}}\right) \\ &= -\left(\frac{p_0(w_{sum} - w_i) + p_1 w_i}{w_{sum}}\right) \\ &= \frac{w_i(p_0 + p_1) - w_i p_0}{w_{sum}}. \end{aligned}$$

Thus

$$L_i w_{sum} = w_i(p_0 + p_1) - w_{sum} p_0$$

whence

$$w_i = \frac{w_{sum}(L_i + p_0)}{p_0 + p_1}.$$

Thus for any two classes i, j, we have $w_i/w_j = (L_i + p_0)/(L_j + p_0)$. Since the denominator of the metric is w_{sum}, the metric is not affected by multiplying all the weights by any non-zero factor. So without loss of generality we may assume that $w_0 = 1$, and calculate all the other weights from this.

5 Optimising the metric

Once we have calculated the weights (and made some assumption about the N_i), we can calculate the metric. One way to optimise our predictions for this metric is just to implement it as a custom metric and use this to train the model. This approach is implemented in [10]

As the kernel shows, optimising the metric makes a big difference to the leaderboard score. However, the approach above isn't ideal because it doesn't take account of class 99, which is not present in the training data. Below we show how to train a model using the standard log loss and scale the outputs to optimise the metric. Importantly, this approach also tells us what scaling to apply to class 99.

Our calculation closely follows that in Section 2 of "Notes on Backpropagation" by Peter Sadowski [11] (although we change the variable names to align with those used on Kaggle's description of the evaluation metric).

For any object x, let h_i be the predicted probability that x belongs to class i. Write $s_i = \ln(h_i)$, so we can write the loss as

$$L = \frac{-1}{w_{sum}} \sum_{i=1}^{M} \frac{w_i}{N_i} y_i \ln(p_i)$$

where $p_i = e^{s_i} / \sum_{k=1}^{M} e^{s_k}$. As in Sadowski, we have

$$\frac{\delta p_i}{\delta s_k} = \begin{cases} p_i(1-p_i) & \text{if } i = k \\ -p_i p_k & \text{if } i \neq k \end{cases}.$$

Modifying Sadowski's calculation to account for the weights, we get

$$\begin{aligned} \frac{\delta L}{\delta s_i} &= \sum_{k=1}^{M} \frac{\delta L}{\delta p_k} \frac{\delta p_k}{\delta s_i} \\ &= \frac{\delta L}{\delta p_i} \frac{\delta p_i}{\delta s_i} + \sum_{k=1, k\neq i}^{M} \frac{\delta L}{\delta p_k} \frac{\delta p_k}{\delta s_i} \\ &= \frac{-w_i y_i p_i (1-p_i)}{w_{sum} N_i p_i} + \sum_{k=1, k\neq i}^{M} \frac{-w_k y_k (-p_k p_i)}{w_{sum} N_k p_k} \\ &= \frac{-w_i y_i}{w_{sum} N_i} + \sum_{k=1}^{M} \frac{w_k y_k (p_i)}{w_{sum} N_k} \end{aligned}$$

Hence

$$\frac{\delta L}{\delta s_i} = 0 \iff \frac{w_i y_i}{w_{sum} N_i} = \sum_{k=1}^{M} \frac{w_k y_k (p_i)}{w_{sum} N_k} \iff p_i = \frac{\frac{w_i}{N_i} y_i}{\sum_{k=1}^{M} \frac{w_i}{N_k} y_k}.$$

Moreover,

$$\frac{\delta L}{\delta h_i} = \frac{\delta L}{\delta s_i} \frac{\delta s_i}{\delta h_i} = \frac{1}{h_i} \frac{\delta L}{\delta s_i},$$

so

$$\frac{\delta L}{\delta h_i} = 0 \iff \frac{\delta L}{\delta s_i} = 0.$$

This means that if our raw prediction of y_i is p_i^* the prediction that minimises the expected loss is

$$\frac{\frac{w_i}{N_i} p_i^*}{\sum_{k=1}^{M} \frac{w_i}{N_k} p_k^*}.$$

In particular, if we want to predict a constant value p_i for each class i, we can calculate the optimal p_i as follows. Write $N = \sum_{i=1}^{M} N_k$ (i.e. N is the total number of predicted values). Then the expected value of y_i in each class i is N_i/N. Hence the optimal prediction is

$$p_i = \frac{\frac{w_i}{N_i} \frac{N_i}{N}}{\sum_{k=1}^{M} \frac{w_i}{N_k} \frac{N_k}{N}} = \frac{w_i}{w_{sum}}.$$

This is the approach taken in [12].

6 Predicting class 99

I predicted class 99 using 1 minus the maximum prediction of the other classes. But this gives an overestimate, since the max is always strictly less than 1 (because the model doesn't have perfect certainty), even in the training set. So I adjusted for this by comparing to the max in the training set. Specifically, for each combination of galactic/non-galactic and exact/approximate redshift, I calculated the average of 1 minus the maximum of the known columns on the training data (call this *train_max_remainder*), and did the same for the test data (call this *test_max_remainder*). Then I scaled *test_max_remainder* by mutliplying by

$$\frac{(test\_max\_remainder - train\_max\_remainder)}{test\_max\_remainder},$$

so that its new average will be $(test\_max\_remainder - train\_max\_remainder)$. However, before calculating this average on the train set, I resampled the training predictions to have similar distribution to test, as described above.

The regularised prediction was calculated as $\mu + (p - \mu) * \alpha$, where p is the raw prediction, μ is the average prediction and α is a regularisation parameter (small α means more regularisation). The regularisation was done separately for galactic and non-galactic classes, since these have very different average raw predictions of class 99 (this is consistent with the results reported in [13] from leaderboard probing).

I used LightGBM's one-vs-all multiclass loss function when training the model, rather than the cross-entropy, because the standard cross-entropy forces the predictions to add to 1, which biases them upwards in this case because of the unknown class not present in the training data. The one-vs-all gives slightly worse accuracy on the training set compared to multiclass log loss, but since we have to rescale all the probabilities anyway to account for class 99, I don't think this matters much.

7 Things that didn't work

I tried out several approaches that didn't seem to add anything, as described below.

- Resampling. I tried adding some more training data, consisting of sythetic object data with some of the points deleted. (Either randomly of the part before the peak).
- Neural networks. Possibly the problem was that I couldn't find a good way to deal with the nulls (replacing with zero gave poor results).
- Lomb-Scargle to find the periods. It seems that the hard-to-classify classes are not periodic, so this didn't really help.
- Adjusting the time scale and frequency to account for the redshift. As described above.
- Curve fitting. Slow, and didn't seem to yield a noticable improvement. Possibly not enough data to estimate the parameters accurately.

References

[1] https://www.kaggle.com/michaelapers/the-plasticc-astronomy-starter-kit?scriptVersionId=6040398

[2] https://www.statlect.com/fundamentals-of-statistics/normal-distribution-Bayesian-estimation

[3] https://ned.ipac.caltech.edu/level5/Hogg/Hogg7.html

[4] https://www.kaggle.com/c/PLAsTiCC-2018/discussion/70725#417195

[5] https://www.kaggle.com/c/PLAsTiCC-2018/discussion/69696#410538

[6] https://www.kaggle.com/kyleboone/naive-benchmark-galactic-vs-extragalactic?scriptVersionId=6104036#

[7] https://www.kaggle.com/c/PLAsTiCC-2018/discussion/67194

[8] https://www.kaggle.com/c/PLAsTiCC-2018/discussion/67194#397153

[9] https://www.kaggle.com/c/PLAsTiCC-2018#evaluation

[10] https://www.kaggle.com/mithrillion/know-your-objective?scriptVersionId=7117143#

[11] https://www.ics.uci.edu/ pjsadows/notes.pdf

[12] https://www.kaggle.com/paulorzp/1-line-classes-weighted?scriptVersionId=6152714

[13] https://www.kaggle.com/c/PLAsTiCC-2018/discussion/68943