

1. Who are you (mini-bio) and what do you do professionally?

Recently graduated in Mathematics & Statistics from the University of Toronto. I regularly participate in various online competitions.

2. What motivated you to compete in this challenge?

I was sitting on a few ideas that I knew could perform really well on time-series data based on my theoretical background and experience in other competitions. Moreover, I was motivated by the potential of helping the study of Mars and other interstellar bodies (in-situ or otherwise).

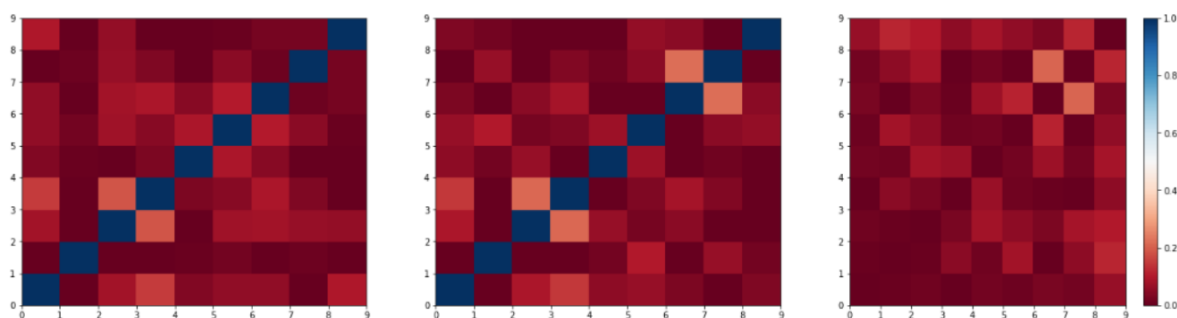
3. High level summary of your approach: what did you do and why?

My solution is an ensemble of ten identical 3-layer Deep Neural Networks (or multilayer perceptrons) where each is trained using a loss with a Label Distribution Learning (LDL) component. LDL works well with data with label noise which I hypothesized existed in the data. I constructed the feature vectors using well-known peak detection techniques alongside n-difference features inspired by the derivative-producing algorithms seen in (older) works in analytical chemistry.

Using these features validating ideas was hard since I observed a relatively high variance validation log-loss, even when averaging runs. This motivated me to use weight averaging which stabilized validation loss. Moreover, my best submissions use a weight-averaging algorithm I developed.

4. Do you have any useful charts, graphs, or visualizations from the process?

Figure 1: The label correlation grids computed over all training labels (left), the validation labels (center), and the absolute difference between them (right).



The above figure motivated me to try Label Distribution Learning techniques (at least initially). Since you can tell that the label distributions, when expressed as pair-wise correlations, are loosely identical.

5. Copy and paste the 3 most impactful parts of your code and explain what each does and how it helped your model.

Below constructs the 'label correlation grid' (lcg) and the corresponding loss function using the lcg computed for both the model outputs and labels; written in tensorflow. This loss is a "Label Distribution Learning" technique that works well with data with label noise and learns to model the pair-wise correlations between the labels. See <https://arxiv.org/pdf/2210.08184.pdf>

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

def self_cross_corr_matrix(x):
    x = tf.cast(x, tf.float32)

    n = tf.shape(x)[0]
    d = x.shape[-1]
    dist = tfd.Normal(loc=0., scale=.5)
    x = x[:, None] + dist.sample([n, d, d])

    y = tf.transpose(x, (0, 2, 1))

    corr = tfp.stats.correlation(x, y, sample_axis=0, event_axis=None)
    return corr

def lcg_loss_func(y, p):
    y = self_cross_corr_matrix(y)
    p = self_cross_corr_matrix(p)

    loss_ = tf.abs(y - p)
    loss_ = tf.reduce_mean(loss_)
    return loss_
```

Below is the core of the weight-averaging algorithm I developed. It continuously trains models on the same data until we find an optimal set of weights that, when averaged, perform well on a validation set.

```
def train_fixed_soup(x_train, x_test, y_train, y_test, f_name,
                    n_iter=40, n_soup=5):
    soup = []
    scores = []
    best_score = 1
    best_soup = []

    for i in range(n_soup):
        w, s = train_model(x_train, x_test, y_train, y_test)
        soup.append(w)
        scores.append(s)

    for j in range(n_iter):
        w, s = train_model(x_train, x_test, y_train, y_test)

        if s < max(scores):
            i = np.argmax(scores)
            del soup[i]
            del scores[i]
            soup.append(w)
            scores.append(s)

        soup_avg = np.mean(soup, axis=0)
        _, val = train_model(x_train, x_test, y_train, y_test,
                             soup_avg, f_name)

        if val < best_score:
            best_score = val
            best_soup = soup_avg
            np.save(str(f_name) + '.npy', best_soup)

    return best_soup, best_score
```

Google colabratory runtime

Below part of the feature engineering pipeline which roughly computes the derivatives of the intensity. Adding more of these features generally improved validation performance but greatly increased the final dimensionality of the vector representation since they are computed across every m/z in the sample.

```
def add_diff_features(df):
    intensity = df.intensity.values

    dx = intensity[1:] - intensity[:-1]
    df['dx'] = [0] + list(dx / 1000)

    dx = intensity[2:] - intensity[:-2]
    df['dxx'] = [0, 0] + list(dx / 1000)

    dx = intensity[1:] - intensity[:-1]
    dxdx = dx[1:] - dx[:-1]
    df['dxdx'] = [0, 0] + list(dxdx / 10)

    return df
```

6. Please provide the machine specs and time you used to run your model.

All the training and inference was done on a Google colaboratory runtime

- CPU (model): Intel(R) Xeon(R) CPU @ 2.20GHz
- GPU (model or N/A): Tesla T4
- Memory (GB): 13
- OS: Linux
- Train duration: ~6 hours
- Inference duration: ~15 mins

7. Anything we should watch out for or be aware of in using your model (e.g. code quirks, memory requirements, numerical stability issues, etc.)?

As mentioned earlier, training the model without weight-averaging can produce a relatively high variation in the validation log-loss. Weight averaging provides much more stable results.

8. Did you use any tools for data preparation or exploratory data analysis that aren't listed in your code submission?

I only selected a subset of the m/z in the final representation. I removed the 10 m/z that accounts for the least amount of variation in the training set.

9. How did you evaluate performance of the model other than the provided metric, if at all?

I only used log-loss.

10. What are some other things you tried that didn't necessarily make it into the final workflow (quick overview)?

Obviously, I tried a similar approach to the first Mars Spectrometry Challenge using large pre-trained convolutional neural nets early on. However, their performance was much worse compared to even low resource MLPs.

I also tried Auto-Sklearn (see <https://automl.github.io/auto-sklearn/master/#>) which tries to find an optimal ensemble of models composed using any of the ML models found on the sklearn package. However, it could only perform well using an ensemble of MLPs,

11. If you were to continue working on this problem for the next year, what methods or techniques might you try in order to build on your work so far? Are there other fields or features you felt would have been very helpful to have?

I definitely want to leverage other Spectrometry datasets for gas chromatography or even liquid chromatography. More data is always a good thing.