Brief Overview

The CHEERS project is a software application that uses machine learning and high-performance computing in order to perform material analysis, specifically in this case, determine the level of sugar and alcohol in red wines.

The ML model used is SVR, which looks to set the largest margin rate and lower the misclassification rate, providing the prediction score for any particular target variable (sugar or alcohol).

Without much details, we are going to take only those points which have the least error rate, thus giving us a better fitting model. In order to achieve this, we need to find the optimal combination of the model's free parameters C and epsilon (leaving a fixed kernel for now), according to
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

We want to perform hyperparameter optimization.

The ground truth data for target variables include: Date, Tank, A420, A520, Alcohol, Sugar, Tannins, Inoculation, Color, Varietal.

The parameters or "features" are: Tank, A420, A520, Alcohol, Sugar, Tannins, Inoculation as seen in
https://github.com/radical-collaboration/FastFingerPrinting/blob/feature/dataread_per_task/src/galloOSIOPT/code/GalloModel-Parallel.py#L155

What are sugar (and alcohol) search levels? number of processes?

R: Search levels are specified by developers to perform the scan on a deeper level, while number of processes is just as it sounds, defines the number of processes to be used when running the code in parallel according to
https://github.com/radical-collaboration/FastFingerPrinting/blob/feature/dataread_per_task/src/galloOSIOPT/code/models.py#L155

Initial Requirements

Functional

1. Which HPO algorithm must be satisfied (grid search, SMBO, K-Means, etc.)?
   - The ML model is well defined

- Training/testing datasets are provided
- Which validation protocol to be used (cross-validation?)
- Parameter search space needs to be defined
- Optimization function needs to be defined (Gaussian process with guided sampling?)

Non-Functional

1. The code must be simple
2. The code must be easy to maintain

Workload Description

It must use the maximum number of cores available on the largest XSEDE machine without significant overhead.

First Approach

We are taking the parameter scan from the code (Chetan's script, specifically getBestSVRMetrics function) and use HyperSpace to convert it to a search since we already have a well defined ML model and provided datasets. We would only need to define the search space according to the free parameters of the model and give it a try.

Solution to be tried using Hyperspace:

- HPO algorithm: SMBO
- ML model is well defined (SVR)
- Training/testing datasets are provided
- Validation protocol to be used is cross-validation
- Hyperparameters search space can be defined (upper and lower bounds can be obtained) from
  https://github.com/radical-collaboration/FastFingerPrinting/blob/feature/dataread_per_task/src/galloOSIOPT/code/models.py#L109
- Optimization function will be Gaussian process with guided sampling

As said before, Chetan's manual grid based search would be a good place to start (https://github.com/radical-collaboration/FastFingerPrinting/blob/feature/dataread_per_task/src/galloOSIOPT/code/models.py#L109). Then, we can also perform a comparison against Andy's Sklearn's GridSearchCV function.

Andy uses the following grid, along with Sklearn's GridSearchCV optimization algorithm:

```
gsc = GridSearchCV(
    estimator=SVR(kernel='rbf'),
    param_grid={
        'C': [0.1, 1, 100, 1000],
        'epsilon': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10],
        'gamma': [0.0001, 0.001, 0.005, 0.1, 1, 3, 5]
    },
    cv=9, scoring='neg_mean_squared_error', verbose=1, n_jobs=-1)
```

It could be worthwhile to expand said grid to add more kernels. Andy mentioned that the linear kernel sometimes outperforms the radial basis function. Also the polynomial and sigmoid could be looked at. Chetan's code already includes these.

UPDATE 05/27/2020:

Questions asked:

1) What is the complexity of the HPO algorithm used, in terms of resources?

R: Speaking solely about resources used (cores), it increases exponentially, with a base of 2, as we increase the number of hyperparameters. This would be something like C = O(2^n) where n is the number of hyperparameters and C would be the number of cores.

2) Can we modify the number of Support Vectors in the SVR model in order to improve performance in terms of accuracy?

R: We definitely can. The Scikit-learn library facilitates this by providing a different ML model, called NuSVR, with a parameter Nu that replaces the epsilon in the "regular" SVR model. This Nu parameter allows us to control the number of support vectors. Therefore, it could potentially affect (for the better or worse) the accuracy of the model. Keep in mind though that the more support vectors that we have, the higher the risk of overfitting, as was well pointed out during the meeting.

So, if you are willing to replace the current ML model, we could play around with it and include the Nu parameter in our optimizations.

UPDATE 06/10/2020:

A parallel Bayesian SMBO program was developed in Python, using EnTK, experimenting with up to 5 parameters (C, epsilon, gamma, kernel and degree) with both NEOSPECTRA dataset and NIRONE 2.5 dataset (provided by Andy). You can find a

comparison and results between this approach and Andy's Grid Search in the following notebook:

https://github.com/radical-collaboration/FastFingerPrinting/blob/feature/starter_tank/phase1/src/galloOSIOPT/hyperparams-opt/code/NIRONE2-5/Andy_comparison_3params.ipynb

We concluded the following:

1. Andy's grid search is simple and effective, works pretty well for small cases like with 3 hyperparameters on a discrete grid. However, we were able to achieve a better prediction score when using a parallel Bayesian SMBO
2. It is more expensive in terms of time for simple cases, bu when scaling this up it has the potential of exponentially reducing time complexity, while searching across many continuous search spaces in parallel
3. We are happy with the progress made so far, and we are working toward showing how this behaves when we scale it up in terms of both the number of hyperparameters and resources utilized (cores), i.e., how far and how fast we can go

UPDATE 06/24/2020:

Question asked:

1) I see there is a library called HyperSpace being used. What role does it play and will we need this going forward?

R: To solve the optimization problem, we are using a library developed in collaboration with the RADICAL team called HyperSpace, which imports Scikit-Learn. We need to continue using HyperSpace unless we reimplement the same functionality inside RCT and build it on top of sklearn directly.

Next steps:

Just to confirm how we plan on doing strong and weak scaling, the package (HyperSpace) that we are using to solve the problem uses MPI. Assuming 4 hyperparameters, the maximum we can go in parallel to solve this is by spawning 16 MPI processes (each one uses 1 core).

We are planning on doing strong scaling (problem size fixed while increasing resources) by showing how it behaves for everything below this. For example, we would measure TTX (Total Time of Execution) for:

16 MPI processes on 1 core
16 MPI processes on 2 cores
16 MPI processes on 4 cores
16 MPI processes on 8 cores
16 MPI processes on 16 cores

We think we can accomplish that using something called “oversubscribing” in MPI, in which we can spawn multiple processes on the same core.

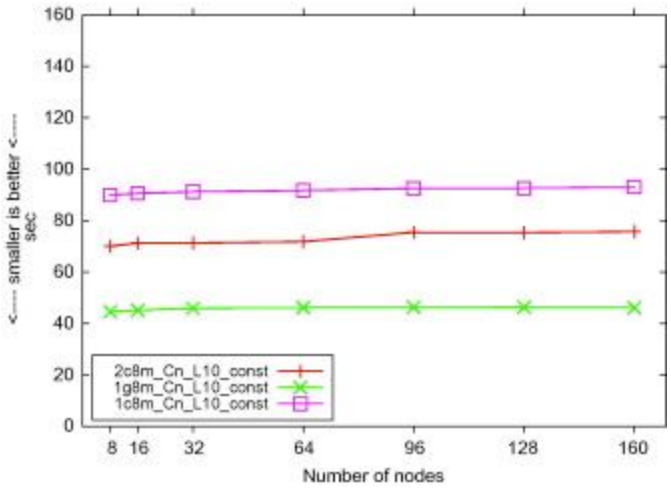
For strong scaling: Hyperspace creates each space based on the available MPI ranks (https://github.com/yngtodd/hyperspace/blob/master/hyperspace/hyperdrive/skopt/hyperdrive.py#L87), so scaling the number of MPI processes (less than what is required) will leave us with an unfinished solution and therefore a bad prediction score. This is a Hyperspace design decision, and the workaround would be to perform the strong scaling experiment the way I suggested previously (fixed number of MPI processes and oversubscribing).

Now, for weak scaling: we would scale it up by increasing both the number of hyperparameters and resources utilized, while keeping the ratio constant in the form of x:2^x. This way, we would measure TTX for:

1 hyperparameter on 2 cores
2 hyperparameter on 4 cores
3 hyperparameter on 8 cores
4 hyperparameter on 16 cores

Keep in mind that the parallel part (lines 1-8 from Hyperspace pseudo-code) could take the same time but what about the non-parallel part (lines 9-11)?

Ideally, we are expecting to see a weak scaling behaviour similar to the below image:



UPDATE 06/26/2020:

After giving it a lot of thought for the past days and after my discussion with Todd Young as well, I have come to the conclusion that just by playing around with the MPI launch command "`mpirun --hostfile hostfile -n 16 ...`" in a way that we let it place the 16 processes on 8, 4, 2 and 1 cores respectively through a hostfile, for example, wouldn't do the trick. We would need to modify the approach taken from Hyperspace itself, and this involves messing around with the library's code at its core. This is doable, but would require investing some time into it.