Citrine Informatics

Data Science Challenge

PREPARED BY: Greg Loughnane

Submitted: 4/21/2019

Contents

Introduction

Materials informatics is a burgeoning area of research and development aimed at reducing the time-to-market currently required for novel materials to find their way into new products and technologies. The opportunity to make a paradigm shift from a materials-selection way of thinking, to a purpose-driven material-design methodology, has been enabled by several converging factors.

The global materials science and engineering community has been improving multi-scale experimental characterization tools used for collecting large quantities of materials data for decades. At the same time, contributions made by advances in multi-physics simulation tools and virtual prototyping, coupled with improvements in application-specific computer hardware, have also played a role. It should be noted that many of these technologies have been enabled by the foresight of large funding programs like the Materials Genome Initiative, which has allowed for unprecedented integration and alignment between experimental and computational materials scientists and engineers (i.e., ICME/ICMSE). As a result, in the last decade, materials science has made the transition to a largely quantitative field of study.

Concurrently, numerous domains have recently profited from the application of powerful open-source data science and optimization tools, including machine learning and deep learning frameworks, to many of their respective data-driven problems-of-interest. This is because these tools can find complex patterns hidden in vast arrays of existing data much better than any human. It is also in large part because of these tools that open materials-data repositories are finally appealing to audiences beyond just materials researchers. This can be observed indirectly by the success of companies like yours, Citrine Informatics, and through acquisitions being made by companies like ANSYS (i.e., Granta Design). Additionally, new business models are being created for specific application areas within materials design and development; for example, Senvol in the metal 3D printing space.

Since early on in my graduate school career, I have worked to position myself at this multi-disciplinary intersection of experiment and simulation; in other words, at the intersection of materials data. I hope to demonstrate in this work that although I continue to learn the best ways to work with these rapidly improving toolsets, I am as enamored as ever with the possibilities that exist to solve physics-based materials problems like the one given for this challenge in exciting new ways.

Problem Statement

In this work, I use open materials data in combination with data science libraries available in the Python programming language to develop an algorithm that can predict the stable binary compounds that form on mixing two individual elemental constituents.

The features chosen for this application are based on a naïve application of the Material-Agnostics Platform for Informatics and Exploration (magpie) feature set. As provided, the training data included 2572 combinations of element pairs, along with associated electrical, mechanical, physical, and other periodic-table properties. These properties serve as the initial feature set for our machine learning model. For each element within each A/B pair, there are 48 individual features, resulting in 96 overall features for each element pair.

The task is to predict the full stability vector for each element pair, which represents the 1D binary phase diagram of stable compounds. The machine learning problem is visualized in Figure 1.

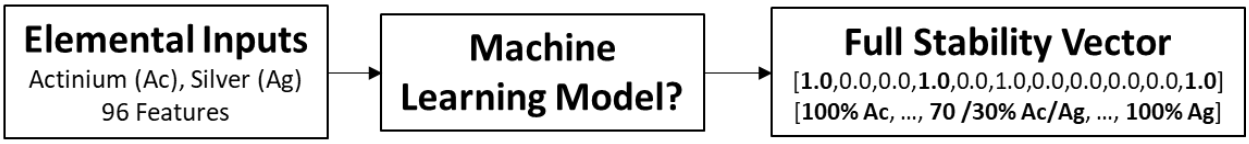

Figure 1: Visualization of the machine learning problem to be solved, including information from the first row of the provided training data.

Technical Approach, Model Construction & Optimization

In this section I will walk through my thought process from start to finish on this problem. Specifically, I will discuss my technical approach associated with each of the following steps:

1. **Splitting the data into training, validation, and test sets**: Here I assign the splits used to perform feature reduction, hyperparameter tuning, and accuracy estimation for blind predictions.
2. **Choice of algorithm**: Discussion of the best ways to cast this problem into an input → black box → output format.
3. **Pre-processing of data and feature engineering**: Determine the optimal amount and type of data to use as input (*I deal with pre-processing and feature engineering separately*).
4. **Hyperparameter tuning of algorithm and model accuracy evaluation**: Quantitatively and iteratively determine the best set of parameters to use for predicting stability vectors.

Since this machine learning problem has a clear problem statement and well-structured data with no holes, many of the most challenging parts associated with getting started are already done.

Additionally, the guidance provided made some pathways more obvious than others to choose. One example of this was that a classical machine learning model was to be used. I generally prefer prototyping a first model in this way, because if I cannot solve the problem using a simple model architecture then it's almost certain that I'll get lost using a deep neural network approach. Another example of a simplification was that precision and recall were specified as the accuracy metrics to be used for model accuracy evaluation and hyperparameter tuning.

Although I will demonstrate just one way of solving this problem, using my best judgement and the available resources I have at my disposal, I have attempted to discuss other salient possibilities that exist within the context of these four overarching steps. Generally, one can do any of the following to improve their machine learning models:

- Put more quality data into the model or take out unwanted data
- Hand-craft additional features or remove unwanted features (i.e., Feature Engineering)
- Tune Hyperparameters
- Try Different Algorithms

I will do my best to discuss these variations within the body of the report.

Splitting the Data into Training, Development, and Test Sets

It is good practice to split data into training, development, and test sets. I chose to go with the simple, classic technique of using 60% of the data for the training set, 10% for a validation, or hold-out cross validation set, and the remaining 30% to use as my simulated blind test set. The figure below describes how data was broken out, and where each part of the data was utilized.

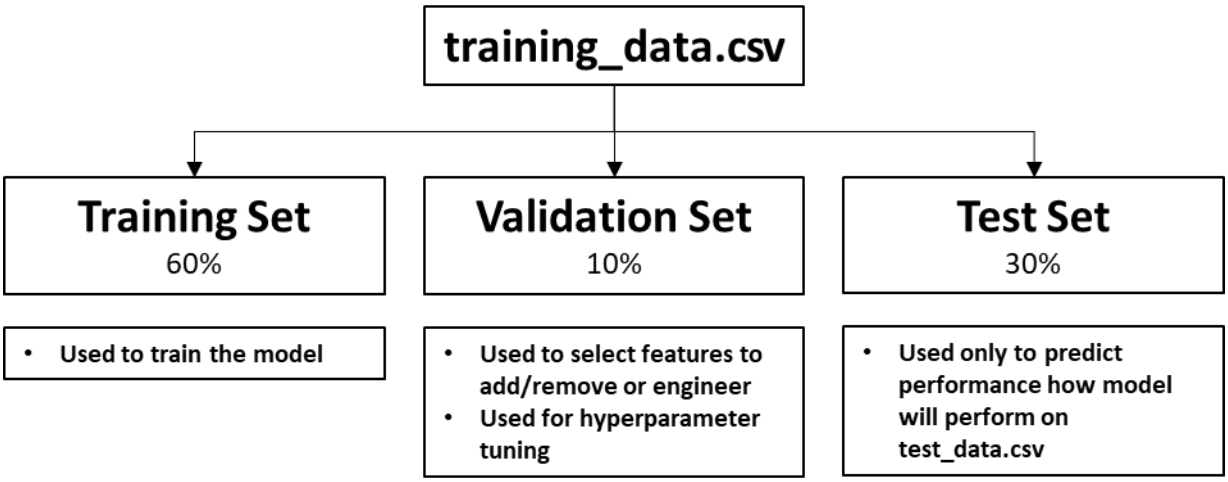

Figure 2: Flow chart showing how data was split up for training, validation, and testing in this work. A classical implementation of a 60/30/10% train/test/validation split was chosen.

I'd like to note here that although k-fold cross validation could have been implemented, I chose to avoid this technique to make interpretation of all my results more straightforward. It could easily be added in the future and is a technique that I've consistently leveraged in my own research.

Choice of Algorithm

The first thing that I noticed about this problem formulation was that it was a multi-class classification problem, made up of 11 distinct classes, each of which were dependent on one another. Further, I do not believe that there is any relationship available to solve for one class and predict another, at least in such a way that it would apply to more than a few select binary compound combinations. Therefore, I wanted to avoid any approach that was solving individual binary classification problems, and then combining their results after the fact. Although this is easier said than done, I also noticed that there were a multitude of physics-based variables that were far from being completely independent of one another. This immediately made me think of using some ensemble method capable of capturing these dependencies, like a random forest, a gradient boosting classifier, or a neural network. Since I wanted to keep it as simple as possible, I decided to go with a Random Forest approach, keeping in mind that overfitting the training data would likely be an issue.

Although I did ultimately go with a random forest approach, I think that given more time it would be interesting to try out a neural network approach, and it would also be interesting to try to implement something similar to the technique used in autonomous vehicle scene identification problems (i.e., independent network/classifier for pedestrian, car, traffic signs, etc.). I am also not sure if there are any algorithms that exist which work well for sequenced output class data, but perhaps investigating Long short-term memory networks associated with natural language processing would be a good place to start looking.

Pre-Processing of Data

The pre-processing of data is extremely important. The first thing that I did was to start playing around with the training data to see what I could glean. After some time with the data, I decided to consider the following data additions, reductions, and transformations.

1. **Every element is stable on its own**: It seemed silly to include the pure elements in my analysis, because they were always going to be stable.
2. **Double the data**: It was mentioned in the initial email that I was given ~5000 element pairs as training data, and after a while it became clear that this would be an easy-to-implement data transformation to improve my statistics.
3. **Mostly metal-on-metal**: It was clear early on that I was given training and test sets with mostly metallic-only binary compounds.

Every Element is Stable on its Own

Since each element in the periodic table is stable on its own, it seemed to be something that I could add after making model predictions. This retroactive output vector formatting is reflected in my code. I also did perform analysis to check if I would get better overall performance across all classes by excluding the ends of the data (see Figure 7). Ultimately, my model was better off without the pure elements.

Double the Data

Simply by reversing the order of the stability vectors for each pair and by replacing A-column information with B-column information the set was doubled. Since there were only 2572 data points, this seemed like a worthwhile thing to do. My intuition is that this transformation perhaps also helped to capture some of the sequencing between elements in output stability vectors that I've already mentioned. Additionally, it would allow me to have larger test, validation, and training sets to work with. It was also clear that this mirroring technique improved some of the class imbalances that existed. Examples, including 20/80% A/B versus 80/20% A/B, or likewise 30/70% versus 70/30% A/B, can be seen in the discrete Probability Density Functions (PDFs) of both distributions shown in Figure 3. It can also be noted here that removal of the pure elements had an extremely large effect on stabilizing class imbalances as well.

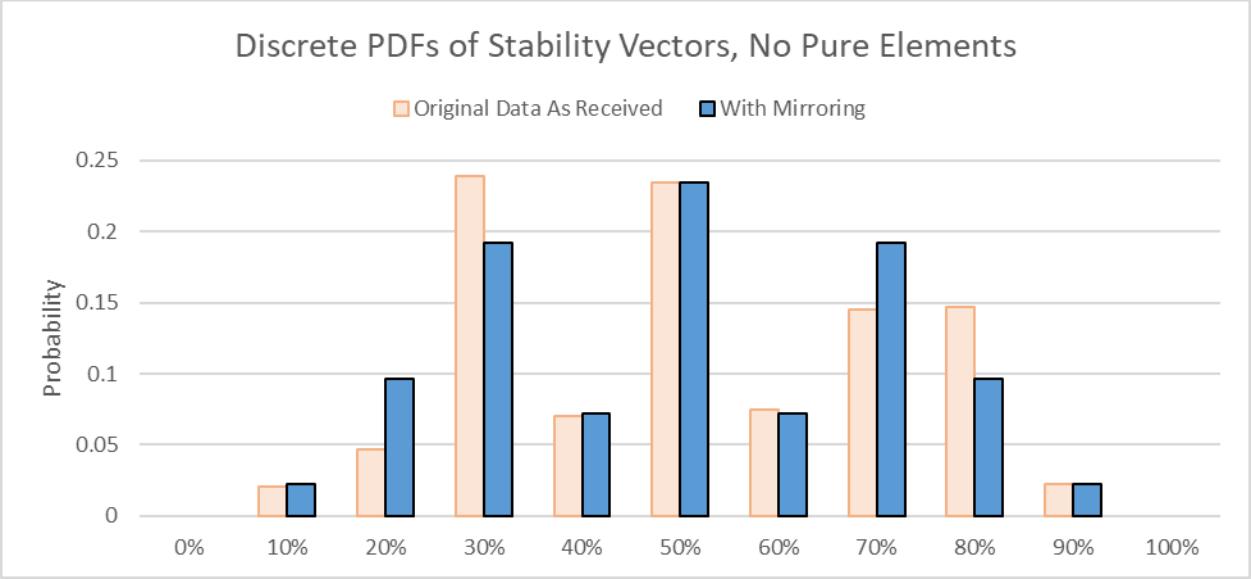

Figure 3: Discrete PDF comparison of initial 2572 element pairs (light orange) and transformed 5144 element pairs (blue). The transformed pairs have improved class imbalances.

Mostly Metallic Compounds

Roughly 2096/2572, or 81.5% of the original elemental data combinations were metal on metal binary compounds. Since it is good practice to choose development and test sets that best reflect the data that you expect to get in the future, and in particular that you want to do well on immediately, I felt that it would not be worthwhile to make a metal-only classifier. This is because the test data that makes up the official challenge had 609/750 metal-on-metal combinations, or 81.2%. Since the training and test data were representative of one another, I left this alone.

If I was working with a customer on this, I might recommend creating a metals-only classifier, a non-metals-only classifier, etc. like the autonomous vehicle approach mentioned above. I might also work with them to scope out exactly what sorts of binary compounds they wanted to be able to use their classifier to predict the stability of in the future. This way, it could be tailored to their specific needs and we could use only the most relevant data. If it was of interest to have a universal classifier, I might also suggest getting more data so that each type of binary compound of interest was well-represented. Lastly, I'd like to note that I'd default to domain Subject Matter Experts as to whether it was even possible to expect that each binary compound of interest could be equally well-represented. It may be true, for example, and seems reasonable to me as a metals guy (but what do I know?), that perhaps metal-on-metal compounds tend to be the most prevalent type of stable binary compounds. Perhaps this would change with ternary compounds and beyond, but it seems that it would be worthwhile to investigate this further.

Feature Engineering

Procurement of the best possible features can be extremely powerful for any machine learning problem. Additionally, the reduction of feature sets to minimally-sufficient combinations of independent features can pay dividends later, during hyperparameter tuning and model optimization. This is especially true if you plan to go through the entire process with multiple types of model architectures.

In trying to come up with the best possible feature set to use as model input, I immediately noticed that all of the features needed to be normalized before being used to create a model, and I chose to do this by subtracting out the mean and dividing by the standard deviation, after testing out a divide-by-max strategy as well.

One of the first things that I did after playing with the data was to perform a light literature review on this problem. This was fruitful and provided me with some direction as to how to expand the existing feature set, as I've detailed in the section below.

In terms of feature reduction, it seemed to me that there were features that intuitively would be less important than others (e.g., IsMetal vs. Atomic Weight). As a first crack at this, I thought of some algorithms that I've used in the past to reduce the number of feature dependencies; namely, sequential forward selection and sequential backward removal. After initially trying to adapt the provided data to my codes, I realized that they consistently resulted in memory errors (stack overflows) from excessively deep recursions. Apparently, I wrote those codes for a lot fewer than 99 features! Instead I ended up leveraging the feature-importance calculations that are built-in to sklearn, which I understand for the RandomForestClassifier are based on the number of times data points flow through individual decision trees. It should be noted here that historically I have not used random forests often and I don't claim to have an extremely firm grasp on how features connect to individual decision trees within forests beyond the high-level descriptions provided by Citrine Informatics[1] and others (see Figure 4 as an example). I look forward to learning more about the ins-and outs of this algorithm, especially since it seems like a favorite of many commercial data scientists, despite often being left out of Massive Open Online Course (MOOC) coursework.

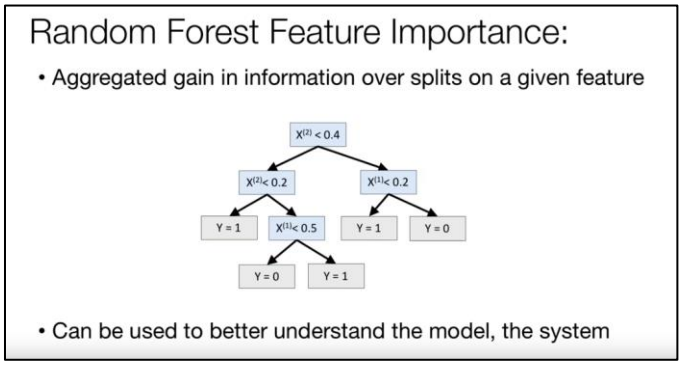

Figure 4: Depicting random forest feature importances, which were leveraged in this work for feature reduction.

So, all in all, related to feature engineering I went through the following steps:

1. **Feature additions, both from the literature and customized**: Unsurprisingly, work by Meredig, et al. [2], was extremely helpful.
2. **Feature reductions towards a minimally-sufficient feature set**: Using a combination of RandomForestClassifier feature importances, intuitions, and verification via Principal Component analysis, I created a minimally-sufficient feature set.

[1] https://www.youtube.com/watch?v=nWk6QlwvXok

[2] Meredig, et al., "Combinatorial screening for new materials in unconstrained optimization space with machine learning," Physical Review B, Issue 89, 2014

Feature Additions, Both from the Literature and Customized

During my literature review, I found that the most useful and applicable information came from some supplemental documentation associated with Meredig, et al.'s paper, "Combinatorial screening for new materials in unconstrained composition space with machine learning." Since the problem being solved in that paper was significantly more challenging than the data science challenge provided here, it was clear that the hand-crafted attributes used to predict ternary compounds would also be useful in predicting binary compounds. So, I went ahead and included 15 of the 17 descriptive attributes that were added for that work into my feature set, including the following:

- Average atomic mass
- Average row and column on periodic table
- Max difference and average atomic number
- Max difference and average electronegativity
- Average number of s, p, d, and f electrons, and associated composition-weighted fractions

The remaining 2 descriptive attributes that were not leveraged were the average and max difference of atomic radii. Here, since I was provided with the covalent and miracle radii, which (after a quick google) appear to be more relevant to binary combinations that the individual atomic radii I decided to take the average and max difference of each of these instead. This worked out extremely well, as can be seen by the right-hand side of both graphs shown in Figure 5 below. All in all, I added 19 additional features for investigation. This brought the total number of features for the investigation to 115 (i.e., 96+19, shown in Figure 5 as 0 to 114).

During this addition of features, I also noticed that only IsDBlock and IsFBlock were originally included in the training data set, and IsPBlock/IsSBlock were left out. Based on the feature reduction performed, I'm sure that I would've removed these as well, but perhaps they'd be important if there were individual classifiers for specific types of elements[3] (e.g., like a metals-only or non-metals-only classifier as mentioned above).

Feature Reductions Towards a Minimally-Sufficient Feature Set

Based on my understanding of the RandomForestClassifier, the most important parameter for tuning is the number of trees, and there are two distinct methods for calculating feature importances, the 'Gini' and 'entropy' methods. First, I wanted to figure out the number of trees that allowed these two feature-importance methods to converge on the features that they were predicting as important; and second, I wanted both methods of feature-importance calculations to converge on one another. To this end, I first investigated Number of Trees = [10, 25, 50, 100, 200, 500, 1000, 2000] for both Gini and entropy methods, while keeping all the other classifier settings as their defaults.

While the model appeared to be overfitting the training data from the first iteration (i.e., 10 trees), it was significantly and undeniably overfitting the training data (i.e., training set accuracy =100%) by the time 100 trees was reached. It also appeared that there was a consensus between each of the feature importance calculations for even low numbers of trees. Figure 5 shows the 2000-tree configurations for both Gini and entropy metrics. It can easily be observed that they came to more-or-less a consensus agreement on feature importances. I defined the threshold for feature

[3] https://en.wikipedia.org/wiki/Block_(periodic_table)

importances based on my own intuition, and also based on the physical nature of the features removed, in addition to the relative magnitude of feature importances that I observed.

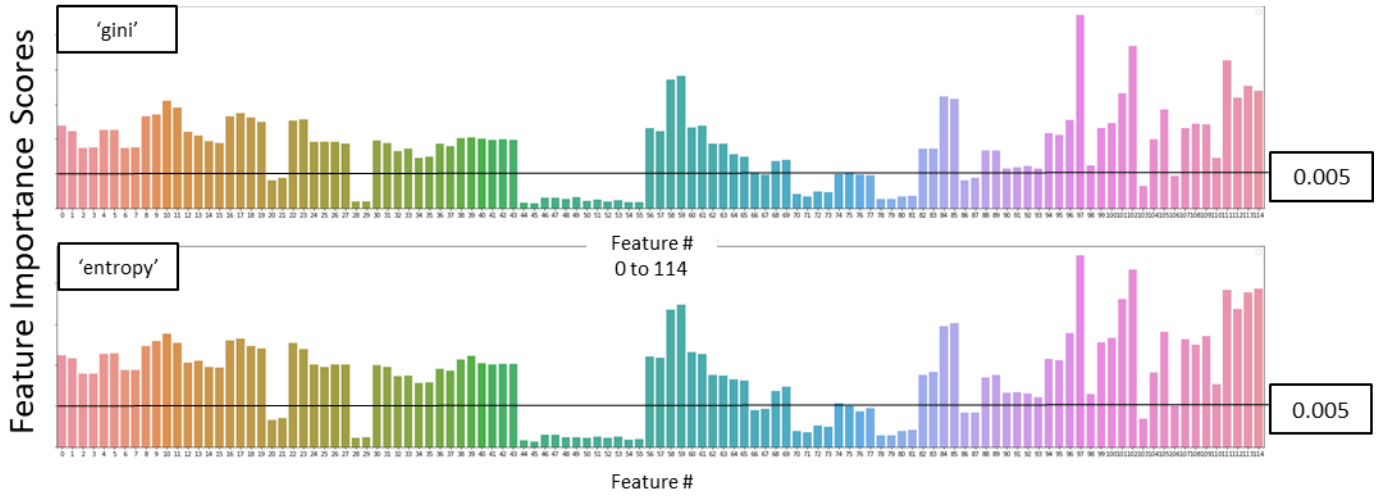

Figure 5: Plots showing the convergence of Gini (above) and Entropy (below) feature importance calculations, for a random forest configuration of 2000 trees that was otherwise set to default parameters. Note that the feature importance can be calculated as the aggregated gain in information over all the splits in the forest on a given feature.

It can also be observed that the engineered features that were added based on the work by Meredig, et al. were identified as extremely important, and similarly, the average and max difference of the Miracle and covalent radii were very successful (see right-most four columns in both plots within Figure 5). Also, it's worth noting is that feature number 97 above, corresponding to the average column of the periodic table for a given element pair, was the most important feature. The second most important feature was feature 102 above, corresponding to the average electronegativity. In fact, the top 6 most important feature were engineered features in this case.

I chose to use a threshold of 0.005 Feature importance score for feature removal. Correspondingly, I removed 29 features based on this threshold. It was also encouraging to see all but one of these features being discarded in A/B pairs. The feature that was not discarded with its corresponding pair was formulaA_elements_NdUnfilled, although its counterpart for element B was not significantly above my threshold. Therefore, I also discarded formulaB_elements_NdUnfilled. In a similar manner, since all of the number of valence and unfilled electrons within each block were removed, I also removed an additional 4 features including A/B NdValence and A/B NpUnfilled that did not actually have feature importances < 0.005. Part of my thinking was that since the added features, especially the average number of s, p, d, and f electrons, and associated composition-weighted fractions, were so successful, and clearly depended on these simpler electron metrics, it was worth removing each of these additional 4 features. Lastly, although engineered features 103 and 106 on the plot, corresponding to the average number of s and f electrons were identified as below threshold, I still kept them in the model, simply because their d- and f-counterparts were not removed. I'm also operating under the assumption that this model may be expanded in the future to more combinations of elements, where these features may have higher relevance.

So, all in all, I removed 34 features from the set, including 17 A/B pairs. These features are summarized in the following list: GSBandgap, Gsmagmom, IsAlkali, IsDBlock, IsFBlock, IsMetal, IsMetalloid, IsNonmetal, NdUnfilled, NdValence, NfUnfilled, NfValence, NpUnfilled,

NpValence, NsUnfilled, NsValence, and Row. This left a total of 81 remaining features, which are pictured in Figure 6 below, shown from 0 to 80.

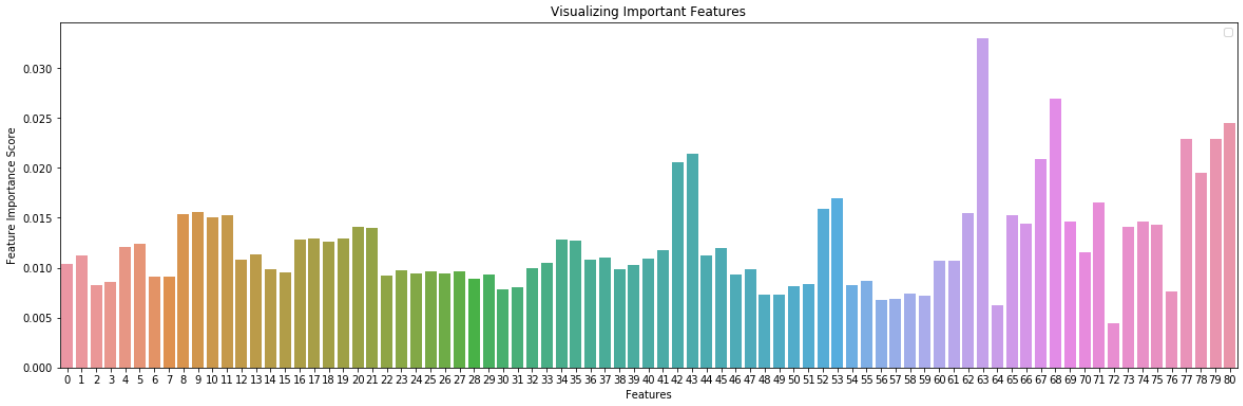

Figure 6: Minimally-sufficient feature set resulting from thresholding based on feature importances < 0.005.

Please note that in the _Supplemental_Info folder included in my submission I've included a folder called HighlightedData_DetailingAdditions&Reductions that has .xlsx files where I've modified the original training and test data .csv files. This is for ease of reviewing my solution.

Lastly, I also performed Principal Component Analysis (PCA) on both the original and reduced feature sets. Both sets produced the exact same "percentage of variance explained" for any given number of principal components, as shown in Table 1.

Table 1: Results from PCA, carried out on both the original expanded and reduced feature sets. The % of variance explained by each principal component was the same for both sets. This tells us that we didn't lose any valuable information during our reduction.

Principal Component	1	2	3	4	5	6	7	8	9	10
% Variance Explained	33.1	65.6	76.6	87.2	91.3	95.3	97	98.7	99.3	99.9

By the time the first 10 principal components were considered, 99.9% of the variance within each feature set was explained. It could be interesting to use this additional data reduction technique to get the number of features down to 10 or less, especially if large hyperparameter spaces or many different algorithms were to be considered. This is, however, beyond the scope of this work.

Hyperparameter Tuning of Algorithm and Model Accuracy Evaluation

One of the final steps in any machine learning problem is to tune the hyperparameters of the selected model architecture for the problem based on some quantifiable metrics. In this case, the metrics to be used are precision and recall, which can be combined into a single number evaluation metric called the F1 score[4].

In addition to the single-number evaluation metric, I also chose to use two satisficing, or "acceptable thresholding" metrics. One is based on time, and the other has to do with one of the downsides of Random Forest; namely; overfitting. I've detailed both of these metrics below:

- Satisficing metric 1 → must run in less than 2 seconds

[4] https://en.wikipedia.org/wiki/F1_score

- Satisficing metric 2 → must not have perfect accuracy on training set and imperfect accuracy on development set (i.e., way overfitting)

Apart from the single-number evaluation metric and the satisficing metrics outlined above, my process for hyperparameter tuning was to first determine which hyperparameters were worth the time of tuning, and then to come up with a systematic approach to knocking them out. While my code can easily be extended to perform either grid- or random-search techniques within extremely large parameter design spaces, this challenge was completed with limited time and resources. As a result, the first parameter I chose to discard from tuning of was the criterion used to measure the quality of a split. Since I saw that the Gini and entropy criterion both performed similarly, I went with the default setting of Gini. Thereafter, I went through the following procedure:

1. **Tuning the approximate number of trees and max_features metric**: Here the aim to was ballpark the number of trees for a further fine-tuning and choose between max_features = 'log2' or 'sqrt'. Additionally, during this step, I wanted to apply each satisficing metric.
2. **Fine-tuning the number of trees**: Here I did another grid search to find the optimal number of trees within a tighter range.
3. **Search a larger space to optimize remaining hyperparameters**: After having optimized the number of trees and the running time, it was much easier to search a larger space to find the remaining optimum hyperparameters.
4. **Estimate model performance on blind test set**: After having optimized the model, I used it to predict performance on the blind test set.

Tuning Approximate Number of Trees and Max Number of Features Metric

The number of trees is the most important hyperparameter in a random forest model, and so I set out to first narrow this number down to something manageable. Since the 2000-tree models were taking about 40 seconds to run, I needed something much faster.

Using my single-number evaluation metric of the F1 score, I could see that in fact, there was really no need to go any higher than 50 trees. This also agreed well with the satisficing metric of having a model that would run in under two seconds. In Figure 7, I've shown the micro- and macro- average F1-scores, for both the situations that include or did not include pure elements. As we can observe, the micro-average with pure elements is wildly inflated, because in micro-averaging we calculate the precision, recall, and F1-score based on the individual true positives, true negatives, false positives, and false negatives of our 11-class model. However, when we macro-average across each individual class, the 11-class model converges on the 9-class model without pure elements by around 25 trees.

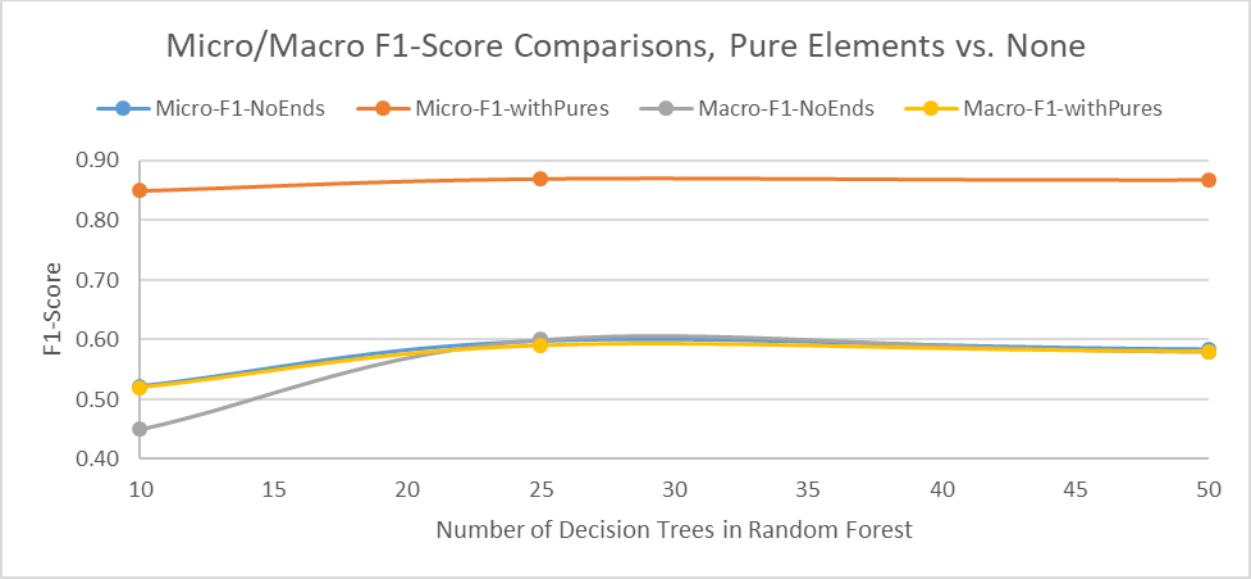

Figure 7: Plot of micro and macro F1-scores for the stability vectors that include pure elements versus those that do not include pure elements. It can be observed that the macro-F1 metrics converge on one another, while the micro-F1 metric is inflated due to the consistently perfect predictions for the stability of pure elements.

This was convincing enough for me to drop the micro-averaging approach altogether for hyperparameter tuning. I decided from there that it would be worthwhile to fine-tune the number of trees by looking closer at the macro-averaged F1 score, for 9 classes, within the 25 to 50 tree range.

Fine-Tuning the Number of Trees

The fine-tuning of the number of trees was straightforward, the results of which can be seen in Figure 8 below. The optimized number of trees within the discrete points that I tried during this fine-tuning was 30, resulting in a macro-averaged F1 score of 0.47.

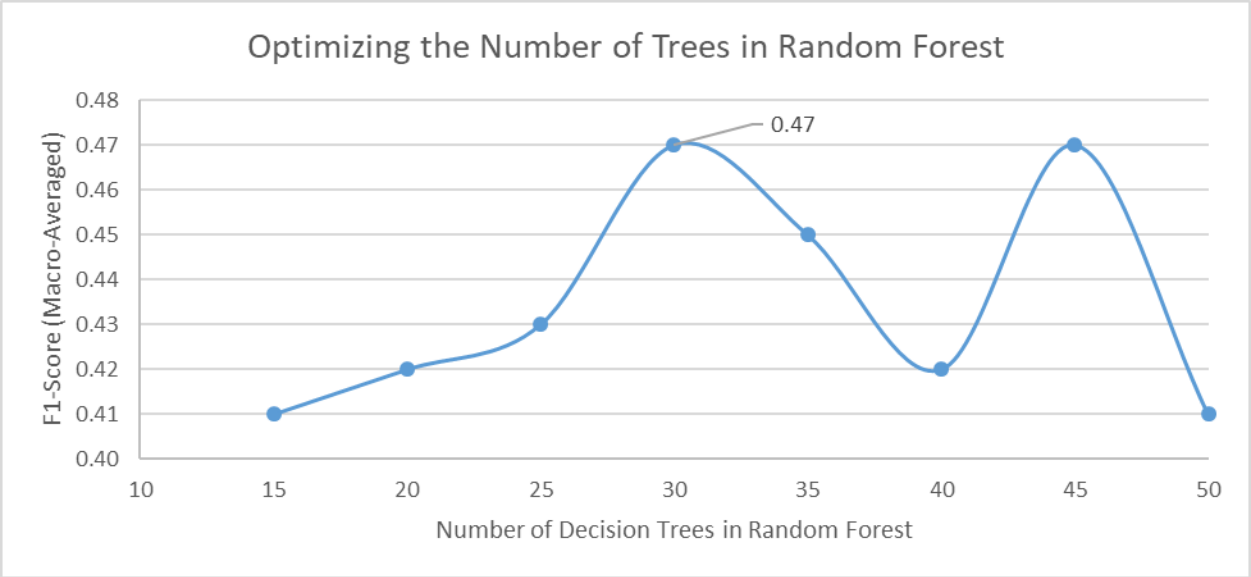

Figure 8: Fine-tuning the number of trees in the random forest resulted in an optimum value of 30 trees.

Search a Larger Space to Optimize Remaining Hyperparameters

Despite random forests having one main parameter of interest, I wanted to demonstrate my custom hyperparameter tuning code on my model that was now down to < 2 seconds per run. I investigated each of the combinations shown in Figure 9.

```
max_features        = ['log2']
max_depth           = [2, 5, 10, None]
n_estimators        = [30]
criterion           = ['gini']
min_samples_leaf    = [1, 2, 4, 8, 16]
min_samples_split   = [2, 4, 8, 16, 32]
bootstrap           = ['False', 'True']
```

Figure 9: Code snippet representing parameter sets investigated during optimization of remaining hyperparameters.

The results of these additional 1400 combinations that I tried out can be observed in Figure 10. Although the same set of parameters was unable to provide the highest macro-F1 score, training set accuracy, and development set accuracy simultaneously, I ultimately decided on the parameter combination 152, which resulted in a training set accuracy of 99.4% (overfit), a development set accuracy of 63.3%, and a macro-averaged F1 score of 0.467. Although I could have gone with a combination that did slightly better on the development set, it would have come at the price of approximately a 9% drop in training set accuracy. Since I'm not an expert at using random forests, it would be interesting to talk to people with significant experience using this algorithm to see if this is typical.

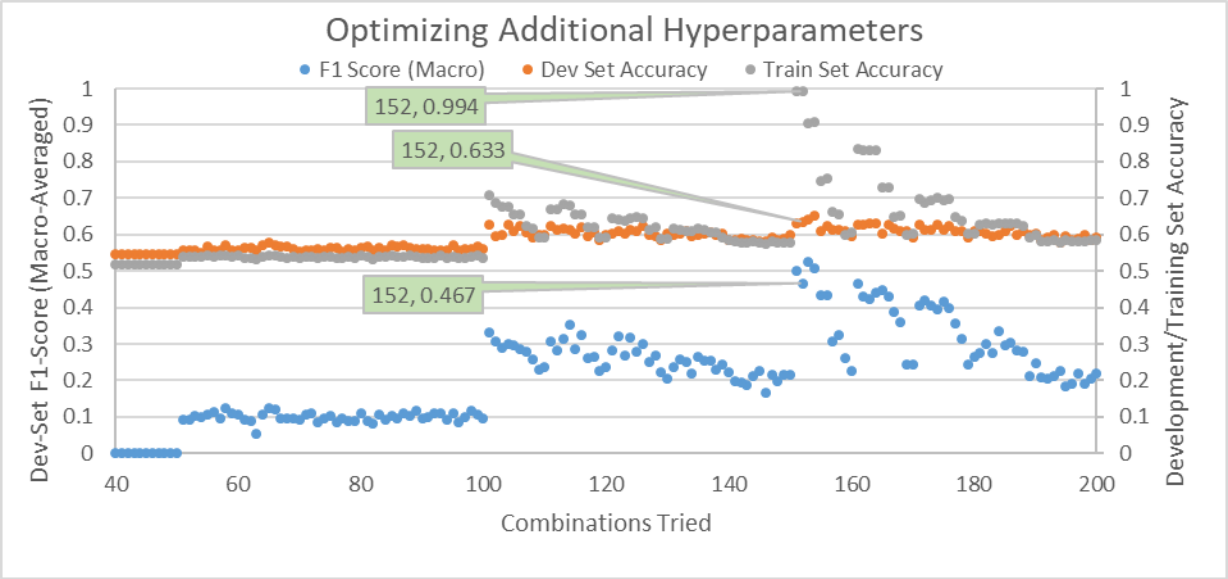

Figure 10: Plot showing all combinations of parameters tried, with data label callouts for each of the metrics associated with the chosen optimal parameter set. Please note that combinations 1 through 50 produced F1 scores of 0 and constant accuracies for both the training and development sets.

Also interesting for this problem was the fact that the macro-averaged F1 score and development set accuracies were consistently higher when using a min_samples_split of 4, rather than the default value of 2 that I ultimately decided on. This intuitively makes sense, because with only 2 samples within the training set required to split an internal node within the random forest, the model was more likely to overfit the training set. Again, here as above, since setting the split to 4

produced only minimal improvements for the development set accuracy, I decided to go with the model that produced the highest training accuracy. Perhaps this could be hashed out with a more systematic and longer process of simultaneous hyperparameter tuning.

Estimate Model Performance on Blind Test Set

With the optimized model now complete, the last step in the challenge was to predict the performance on my predictions for your *test_data.csv* set. My best guess is that my model will perform approximately as well on that data as it was able to consistently perform on the 30% of blind test data that I kept out of model construction and hyperparameter tuning, which is in my estimation somewhere north of 60% and south of 65%.

I can also say that on average, when my model is incorrect, it generally is predicting a 1 when it should predict a 0 (i.e., it thinks that the binary compound composition will be stable when it will not be). It seems to do very well, however, at predicting stability when it decides to, because it seems to do extremely well avoiding false negatives, or Type II errors. In other words, it doesn't seem to miss, it only seems to overshoot! The last thing that I'd like to note is that perhaps this is due to the nature of class imbalances present in the problem. For example, it is simply less likely to get stable compounds using 10%/90% A-B combinations. There just are not that many compounds that like to play together like that. This is something that I'm not sure can be avoided and may also be something that supports the idea of using different models for different output classes, as mentioned in the Choice of Algorithm Section.

Conclusions

The optimum random forest model chosen can be studied directly in the source code and via the Output.html file included with the submission documents. Please see the Appendix for more information on where to dig deeper into any aspects of my thought process that I've attempted to outline for you.

Appendix: Submission Documents List

Each of the documents included in the Submission folder are numbered within the folder and detailed below:

1. **1_test_data-FILLED_IN.csv**: CSV File, as received for the challenge, with entries in the last column of my predictions
2. **2_CitrineInformatics_DataScienceChallengeReport.pdf**: PDF file of this report
3. **3_Code+Outputs**: folder containing the python source code with instructions for running, modified .csv files required to run the code, and associated outputs. File names in reverse alphabetical order are:
    - **training_data_x2-extended&reduced.csv**: the modified training data containing doubling, as well as feature additions and reductions
    - **training_data_stabilityVec_x2-extended&reduced.csv**: the modified stability vector data used for training, doubled with pure elements removed
    - **test_data_extended&reduced.csv**: the modified test_data.csv file required to mimic modified training data
    - **RandomForestClassifier-CustomHyperParameterTuning-FINAL.py**
    - **PredictedStabilityVectors.txt**: the stability vector predictions included in item 1 above
    - **Output.html**: the output that you will get in the IPython console if you run the source code in Spyder via Anaconda according to directions contained within
    - **Output_files**: external output files associated with Output.html
        i. **qt_img118695716192261.png**: see Figure 6.
        ii. **qt_img118807385341957.png:** Binary confusion matrix constructed via mlxtend library, not used in report
4. **4_Supplemental_Info**
    - **HighlightedData_DetailingAdditions&Reductions**: excel files that show the feature additions and reductions to as-received training_data.csv and test_data.csv
        i. **test_data_extended&reduced_Additions&ReductionsHighlighted.xlsx**
        ii. **training_data_x2-extended&reducedAdditions&ReductionsHighlighted.xlsx**
    - **RandomForest-HyperParameterTuning-Steps-FINAL.xlsx**: excel file that shows details from each of the steps that I went through related to hyperparameter tuning