Predicting Superconducting Critical Temperatures with Supervised Machine Learning*

K. Kleinasser, Cornell University, Ithaca, NY†

CONTENTS

I. INTRODUCTION

I.1. Superconductors

Superconductors are materials that lose all electrical resistance at low temperatures. These materials have a critical temperature (T_C) at which they lose their resistance. Most have very low critical temperatures, but "unconventional superconductors" can have critical temperatures as high as room temperature under non-atmospheric conditions.

Electrons in superconductors form Cooper Pairs below their critical temperature. These pairs of electrons are held together with phonouns, which are atomic-level collective excitations. Phonouns are similar to photons in that they also have particle-like properties [1].

Unconventional superconductors are still not well understood and remain an open question in Physics. Understanding them could lead to the discovery of superconducting materials stable at room temperature under atmospheric conditions. Such a material would have large implications, such as super efficient electricity transfer and vast efficiency improvements for applications like particle accelerators and power lines.

I.2. Matminer

Most superconducter databases do not include enough information to train an effective machine learning model, but such data can be extracted from the data they do provide. We use matminer to produce our features from the provided material data. Matminer is a python library that generates data from various measured properties of a material [2]. Matminer collects existing calculations into a machine learning friendly python package.

Our database only provides the superconductor composition data. Matminer's featurizers can generate 53 features from the composition of a material. If we had band structure or other data, we could produce more information that we could use in our model.

I.3. Machine Learning

Previous papers have used random forest models to predict critical temperature [citation needed], but this paper will examine eight models before settling on two for further investigation. All models are implemented with Scikit-Learn, with the notable exception of a mlens superlearner [3, 4]. We will also use MAPIE models for uncertainty, discussed in Section II.4.

We started our model search with some linear models. Besides the base Linear Regression model, we used linear (and polynomial) Support Vector Regression (SVR) models. SVR uses decision boundaries, which are lines parallel to the regression line. The model aims to maximize the amount of data within the decision boundaries and has hyperparamters to modify sensitivity to prevent overfitting.[1] We also trialed Elastic Net and Bayesian Ridge models. Elastic Net uses L1 and L2 penalties to stabilize the model, and Bayesian Ridge uses probability distributors instead of point estimates.

Additionally, we trialed Decision Tree and KNeighbors (KNN) models. Decision trees are very interpretable - they break predictions into nodes of the tree, eventually leading to a prediction value. These trees can be represented graphically and show how they produce results, unlike most machine learning models. KNN models are

* This work is supported by the U.S. National Science Foundation under award number NSF PHY-2150125, REU Site: Accelerator Physics and Synchrotron Radiation Science.
† Lycoming College, Williamsport, PA; klekirk@lycoming.edu

[1] Overfitting occurs when a model is trained to be too specific to a particular dataset and is not generalizable.

a little different, they store all the data and predict values based on a similarity measure. The model looks at a specified number of similar neighbors to produce a prediction.

Finally, we tried multiple ensemble models - Random Forest Regression (RFR), Extra Trees, and a superlearner. RFR models use numerous decision trees and subsamples the data with replacement. This means that the model replace data after using it in a subset. Extra Trees is like RFR, but it does not replace the data after use in a subset. The final ensemble model we tested is a superlearner, a model that can combine multiple high-scoring Scikit-Learn model predictions and sometimes improve the performance from the individual models.

II. METHODOLOGY

II.1. Datasets

We chose to use one of the most popular experimental datasets, the supercon database from Japan's National Institute for Materials Science. This datasets contains 16,414 superconductor chemical compositions and their experimentally measured critical temperatures. Unfortunately, the database is not currently available on their website for unspecified reasons, so we obtained the dataset from a github repository that used this data [5].

In this dataset, there are 10,154 samples with a critical temperature below 10K and 6,210 samples above 10K. There are 159 samples with temperatures above 100K and 0 samples above 260K.

II.2. Code Structure

The source code used for this paper is available publicly on github at `https://github.com/sylphrena0/classe2022`. This repository also includes the source files for this latex paper, data files, images, and documentation files.

Our research uses numpy and pandas throughout our code to handle arrays and tabular data [6, 7]. We also use matplotlib and seaborns to generate our graphs [8, 9].

The code is split into multiple python files so processes could be completed in stages and to maintain readability in the code. Most of our testing and final training was completed in juypter notebooks, but some computations were highly computationally expensive and needed to be run remotely. For these jobs, we created simple python files and made bash scripts to run them on Cornell's CLASSE compute farm. We also made several bash aliases and functions to simplify the compute farm workflow, which are also available on the github repository.

Since we used multiple files, we chose to create shared dependencies files where we defined functions to import data, train models, and generate our graphs. These files are then imported in all the relevant scripts to reduce redundancy. More detailed explainations of the purpose of each file is available in the github readme file and documentation within the files.

II.3. Code Evaluation

First, the featurizer script imports the dataset, extracts features from the material compositions, and exports the csv data. This script is one of the most computationally expensive and takes several hours to run on the CLASSE compute farm with 64 dedicated cores.

After the features are exported, our analysis jupyter notebook imports the data with the shared import function and exports histograms and a correlation matrix.

Next, the training_single jupyter notebook or script can train individual models with the shared evaluation functions. This is used to get a landscape of initial performance before optimization. After training, the function plots the actual T_C versus the model prediction, using a heatmap to visualize the difference from the ideal prediction.

The optimizer script then uses a grid of manually defined hyperparameters to optimize models based on R2 score. This allowed significant improvements to baseline models. After optimization, the optimized models can be plotted in our single training notebook. After confirmation that the model is better than the baseline, the models can then be plotted together in a single graph using our bulk training notebook.

We evaluated our models using several metrics - R2 scores (R2) for regression evaluation, Mean Squared Error (MSE) and Mean Absolute Error (MAE) for error evaluation, and MAPIE Effective Mean Widths (MWS) for uncertainty evaluation.

II.4. Uncertainty

Our evaluation functions can produce uncertainty calculations using forestci, mapie, or lolopy [10–12].

Forestci is python implementation of an algorithm from [13] that predicts confidence intervals for random forest models. It is the fastest of the uncertainty methods listed.

The Model Agnostic Prediction Interval Estimator (MAPIE) python library is more recent implementation of jackknife based on [14]. MAPIE uses various resampling methods. Most methods require the use of MAPIE's own MapieRegressor, which accepts an Scikit-Learn regressor and keeps track of uncertainty as the model is trained. MAPIE also has a prefit method, but it is difficult to extract uncertainty bars for individual points from this data - it splits a celebration set off the test set to generate uncertainty, so it can't be easily added to our plot of test set predictions. Thus, we will only compare the normal MAPIE methods with the other libraries. MAPIE trains much slower than other models, particularly on our superlearner, but it is still considerably faster than our final uncertainty model, lolopy.

Lolo is a Scala random forest machine learning library and is not a native python implementation. Lolopy is a python wrapper for lolo, but this implementation is very slow for large datasets.

III. RESULTS

III.1. Model Optimization

Each model's hyperparameters[2] were optimized with various optimization methods. Optimization can be computationally expensive, so we chose to optimize on a randomly selected subset of 2,000 materials, using a numpy random state for reproducibility. To start, we used Scikit-Learn's GridSearchCV, which tests combinations from a grid of hyperparameters and returns the best performing model based on a specified metric.

We also implemented Bayesian optimization using Gaussian Processes methods from the Scikit-Optimize library [15]. Bayesian optimization attempts to optimize models intelligently, instead of randomly testing specified hyperparameters. We provide a range of hyperparamters values for Bayesian optimization to test, and the algorithm uses acquisition functions to decide which specific values to use within the sepcified range. This is different from GridSearchCV, which is simplier but can take much longer to find optimial hyperparameters. We only implemented Bayesian optimization on our top models.

The performance of each optimizer on selected high performance models is show in Table I. Note that the optimial method for each model (shown in bold), was found with a different method each time. The Bayesian optimization was much less computationally expensive than GridSearchCV, however, and is the recommended method for large datasets.

Model	Optimizer	R2	MSE	MAE	MWS
Random Forest	Base Model	0.807	140.482	5.791	53.135
	GridSearchCv	0.804	142.524	5.816	52.416
	Bayesian – PI	0.815	134.509	5.72	50.985
	Bayesian – EI	0.814	135.267	5.763	51.533
	Bayesian – gp_hedge	0.815	134.785	5.747	51.021
Extra Trees	Base Model	0.818	132.395	5.214	48.153
	GridSearchCv	0.818	132.374	5.215	48.231
	Bayesian – PI	0.818	132.104	5.225	48.154
	Bayesian – EI	0.816	133.5	5.257	48.757
	Bayesian – gp_hedge	0.819	131.783	5.202	47.732
KNN	Base Model	0.646	257.108	8.563	75.612
	GridSearchCv	0.703	216.186	7.515	69.758
	Bayesian – PI	0.652	253.137	8.201	74.177
	Bayesian – EI	0.652	253.137	8.201	74.177
	Bayesian – gp_hedge	0.566	315.238	8.159	84.546

TABLE I. Comparison of optimization methods by model.

III.2. Base Models

Our numerical results are shown in Table II.

	Model	R2	MSE	MAE	MWS
Unoptimized	**Extra Trees Regression**	0.818	132.395	5.214	48.153
	Random Forest Regression	0.807	140.482	5.791	53.135
	KNeighbors Regression	0.646	257.108	8.563	75.612
	Decision Tree Regression	0.644	258.945	6.886	76.715
	Bayesian Regression	0.392	441.925	14.52	90.7
	Linear Regression	0.392	442.163	14.506	90.705
	Elastic Net Regression	0.328	488.242	15.603	97.938
	Support Vector Machines	0.084	666.16	15.511	134.628
Optimized	**Extra Trees Regression**	0.819	131.624	5.205	48.088
	Random Forest Regression	0.816	133.87	5.714	50.759
	KNeighbors Regression	0.703	216.186	7.515	69.758
	Decision Tree Regression	0.664	244.095	7.152	74.834
	Elastic Net Regression	0.392	442.133	14.487	90.713
	Bayesian Regression	0.392	441.925	14.52	90.7
	Linear Regression	0.392	442.163	14.506	90.705
	Support Vector Machines	0.325	490.661	14.186	106.946

TABLE II. CV scores of the models using all features and data, comparing optimized and base hyperparameters.

III.3. Cleaned Models

Our intial

III.4. Error Comparison

Our intial

III.5. Feature Importance

Our intial

IV. CONCLUSION

IV.1. Future Work

[2] Hyperparameters are machine learning parameters that change how a model is trained.

[1] J. W. Rohlf, Superconductivity, in *Modern Physics: From Alpha to Z* (Wiley, 1994) Chap. 15.

[2] L. Ward, A. Dunn, A. Faghaninia, N. E. Zimmermann, S. Bajaj, Q. Wang, J. Montoya, J. Chen, K. Bystrom, M. Dylla, K. Chard, M. Asta, K. A. Persson, G. J. Snyder, I. Foster, and A. Jain, Computational Materials Science **152**, 60 (2018).

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, Journal of Machine Learning Research **12**, 2825 (2011).

[4] S. Flennerhag, Ml-ensemble (2017).

[5] vstanev1, Vstanev1/supercon: Data used in "machine learning modeling of superconducting critical temperature" paper (2018).

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, Nature **585**, 357 (2020).

[7] T. pandas development team, pandas-dev/pandas: Pandas (2020).

[8] J. D. Hunter, Computing in Science & Engineering **9**, 90 (2007).

[9] M. L. Waskom, Journal of Open Source Software **6**, 3021 (2021).

[10] K. Polimis, A. Rokem, and B. Hazelton, Journal of Open Source Software **2** (2017).

[11] V. Taquet, G. Martinon, N. Brunel, I. Ibnouhsein, F. Deheeger, R. Adon, A. Papp, A. A. Goumbala, A. Borgohain, T. Morzadec, and et al., Mapie - model agnostic prediction interval estimator (2022).

[12] M. Hutchinson, Citrineinformatics/lolo: A random forest library (2022).

[13] S. Wager, T. Hastie, and B. Efron, Journal of machine learning research : JMLR **15**, 1625 (2014).

[14] R. F. Barber, E. J. Candes, A. Ramdas, and R. J. Tibshirani, Predictive inference with the jackknife+ (2019).

[15] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi, scikit-optimize/scikit-optimize (2021).